

# A Case Study of Framework Design for Horizontal Reuse

Henrik Bærbak Christensen and Henrik Røn  
University of Aarhus  
Centre for Experimental Computer Science  
DK-8200 Aarhus N, Denmark  
Tel.: +45 8942 3188  
E-mail: {hbc,roen}@daimi.au.dk

## Abstract

*In this experience report we present a case study in designing, implementing, and reusing an object-oriented framework for constructing and editing complex search expressions over arbitrary data domains. We present a framework design that achieves a large degree of business domain decoupling through the application of design patterns. We outline the reuse process and analyse and classify the problems encountered during the first instance of framework reuse. The major lessons learned are that while design patterns are well-known for providing decoupling solutions at the code level, the lack of similar decoupling techniques at the non-code level may give rise to technical mismatch problems between the framework and the client systems; that such technical mismatch problems can be costly; and that a reusable framework may beneficially provide a solution template when it can not provide actual functionality.*

## 1. Introduction

In this paper we outline the design of a reusable framework for editing complex search expressions over arbitrary data domains. The framework has been successfully implemented and is operational in two commercial products focused on two distinct business domains. Reuse across business domains, horizontal reuse, is intrinsically difficult: as there is no business domain given a-priori, there is no overall software architecture to define the relations and interfaces between the reusable software assets. This often (or rather usually) leads to *architectural mismatch* between the reusable asset and the client applications, making it anything from just tedious to very difficult to make them work together [12, 19]. Another problem of horizontal reuse is to strike the right balance between making few assumptions while still providing enough functionality to make the investment in understanding the framework worthwhile.

We outline the process of maturing and reusing the framework. The problems encountered are analysed and classified. Finally, we present the main lessons learned from the case study.

## 2. Background

The framework is a graphical editor for constructing and editing search expressions over an arbitrary data domain. As an example, it is possible to instantiate it to handle the domain of e-mail messages; this would allow a user to construct a search expression like (Subject='COT' OR

Subject='CIT') AND Receiver='hbc' AND Date after '01-01-2000'. The asset is implemented in C++ as an object-oriented framework.

The motivation for the case study was twofold: to get first hand experience with systematic reuse issues in a concrete industrial context [20]; and to meet an actual need for similar functionality within two different projects.

The first project, IRIS/MFS, is responsible for the continued development of a mature product for handling structured electronic messages, i.e., a specialised e-mail tool. The second project, EWare, develops a system for maintaining large databases of radar signatures. In IRIS/MFS searching is used to find particular structured messages; in EWare searching is used to define subsets of a database through filtering.

The technical setting for the case study was that both projects used C++ but used different compilers and development environments (i.e. understood as conventions for organising source files, "make" programs, etc.).

### 3. The Challenge

The challenge was to construct a reusable framework that would be operational within both projects. A closer analysis revealed five major areas in which the framework had to be decoupled from the specific client applications:

1. *Domain data.* The two application domains have very different data models. The electronic mail domain has only one entity type, message, with a fixed number of attributes. The radar signature domain has an unlimited number of entity types and the number of attributes of each entity may vary from customer to customer. Thus, the client of the framework must be able to specify the entities and attributes of the domain data.
2. *Expression execution.* A consequence of the need to handle arbitrary domain data is that the asset can not assume anything about how the domain data is searched. Still, it must produce an expression that facilitates the actual execution of the search by the client. Thus, the asset must be decoupled from the actual search algorithm.
3. *Representation of expression.* A consequence of the fact that the search expression must be executed by the client is that search expressions must be passed between the instantiated framework and the client. Thus, the representation of the search expression must be independent of the business domain.
4. *Graphical user interface.* The search expression is edited by a user through a traditional graphical user interface dialog. The project applications run on both UNIX and Windows NT. Thus, the asset must be decoupled from the specific graphical user interface toolkit.
5. *Custom attribute editors.* The framework must support any domain and it must thus be possible to edit values of types of attributes unknown to the framework. Thus, the client must be able to define graphical user interface editors for the particular attributes of its domain.

These challenges are typical for horizontal reuse problems: at face value the needs of the two applications appear rather identical, namely searching, but a closer analysis reveals that there are many domain issues that may influence the design in incompatible ways. For instance the EWare project uses a SQL database while IRIS/MFS uses a pointer-based navigational network database; designing for one without considering the other quickly leads to a solution that is in semantical mismatch with the way the other domain must execute the search.

At the other end of the spectrum is the pit-fall of providing a “shallow solution” in the sense that it does not really provide any functionality that could not be build from scratch with less effort than spent on understanding the framework. Our analysis revealed that it is not possible to let the framework execute the search due to the different data storage schemes; but the framework still addresses the search execution issue by defining a template for doing so, as will be outlined shortly. This stands as an important difference from making it the full responsibility of the client to solve the problem.

## 4. User Interface

The user of a client application constructs and edits search expressions using the framework’s dialog, shown in fig. 1. Pane ① is a list-box listing all attributes of the client application’s domain. The user constructs/edits an atomic expression by clicking on any of the attributes in the list. This action results in an attribute editor appropriate for the type of the attribute to appear in pane②; i.e. if the user clicks “Subject” an editor for editing string expressions will appear, or if the user clicks “Message Time” an editor for time expressions will appear (as shown in fig. 1).

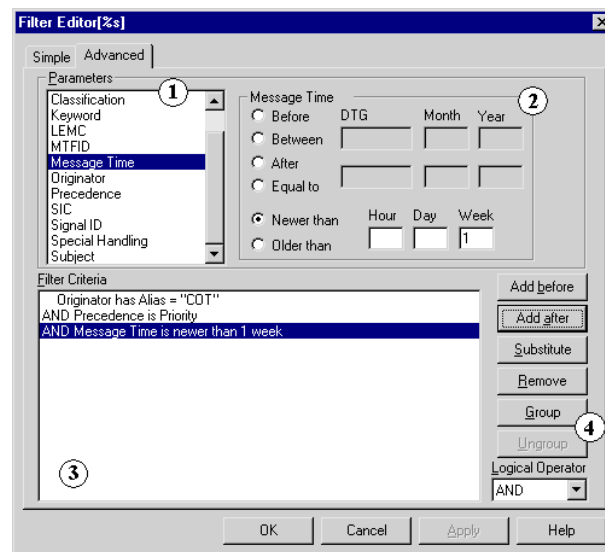


Figure 1. The expression editor.

The lower two panes, ③ and ④, allow the user to compose atomic expressions into the composite search expression. The buttons in pane ④ allow the edited atomic expression in pane ② to be inserted into the composed expression using AND, OR, and NOT, or to be removed. The constructed search expression is shown in pane ③. Clicking any line in pane ③ loads the atomic expression editor, pane ②, with the proper values for editing.

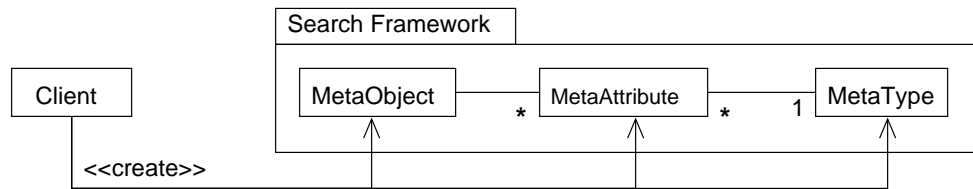
## 5. Design

In this section the design of the framework is outlined with emphasis on the adopted solutions to the challenges described in Section 3. The design relies heavily on design patterns to achieve the

decoupling. The design will be presented by partial UML class diagrams focusing on the decoupling solution in question. A more thorough description is given in [17].

### 5.1. Domain Data Modelling

Figure 2 shows an UML class diagram that outlines the way decoupling of the domain data is achieved. The *Reflection* [5, p. 193] pattern is used to define a meta-model of the domain specific entities and attributes. That is, each entity in the domain is modelled by an instance of *MetaObject* and the entity’s attributes are modelled by instances of the *MetaAttribute* class. For instance the e-mail application must define a *MetaObject* instance to model the basic domain entity, message, and associate it with *MetaAttribute* instances modelling message attributes like subject, sender, body, etc. Each *MetaAttribute* instance is associated with a *MetaType* that defines the type of the attribute.



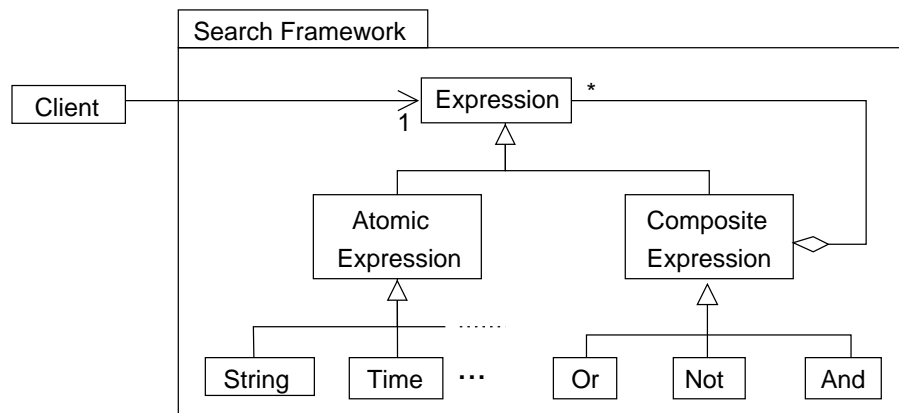
**Figure 2. Domain data decoupling.**

The framework supports a small number of build-in *MetaTypes* representing string, time, integer, etc. The client application can define its own *MetaType* instances; for instance the EWare project needs a geographical position type.

The instances defining the meta model are given to the framework when it is instantiated.

### 5.2. Expression Representation

A search expression is defined as an hierarchical expression tree using the *Composite* [11, p. 163] pattern, as illustrated in fig. 3. Leaf nodes represent atomic expressions over a single attribute, e.g. Subject=‘COT’, while non-leaf nodes represent sub-expression composition using AND/OR/NOT. When the user ends an editing session, the root node of the expression tree is returned to the client application.

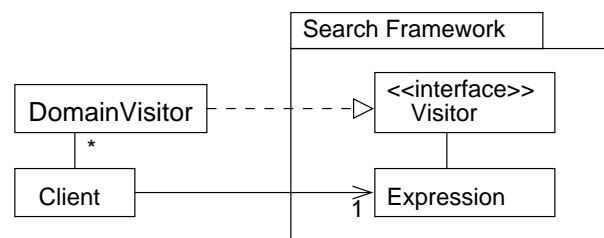


**Figure 3. Expression representation.**

Again, the framework supplies a number of build-in types of nodes such as string, integer, time, etc. If the meta-model has been extended with new types (MetaType instances) these must be mirrored by new node classes representing these types by subclassing the AtomicExpression class. Instances of node classes representing client defined types are created by the type's corresponding attribute editors (see section 5.5 below).

### 5.3. Expression Execution

The framework defines a *Visitor* [11, p. 331] pattern interface for traversing a search expression and the client must define a class that implements this to perform the actual expression execution, that is the search, in the domain data of the client (fig. 4). The execution is thus the responsibility of the client but the framework defines a rigid, yet flexible, template for the structure of the client's implementation of the search code.

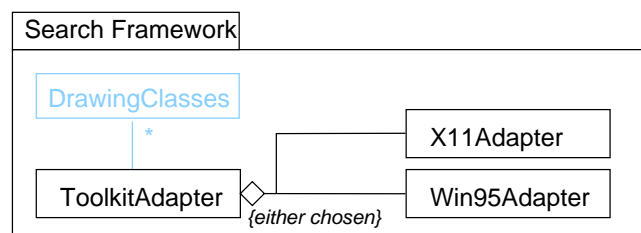


**Figure 4. Expression execution.**

The string representing the composite expression in pane ③ (fig. 1) is produced by a build-in “pretty-print” visitor instance.

### 5.4. Graphical Toolbox

The *Adapter* [11, p. 139] pattern is used to define an abstract, platform-independent, graphical user interface toolkit that the framework depends upon for standard graphical widget drawing, see fig. 5. The framework provides backends for X11 and Windows at present. The toolkit adapter is

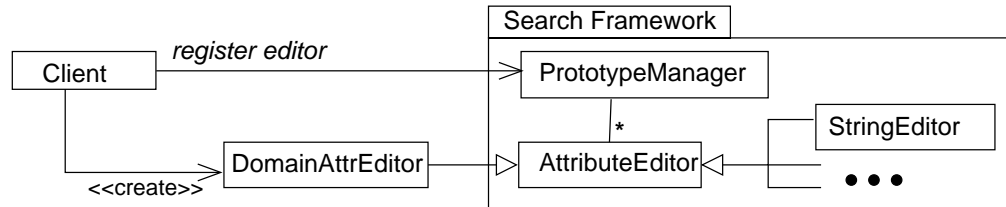


**Figure 5. Toolkit adapter.**

lightweight in the sense that it does not provide complete decoupling from the underlying graphical toolkit. It relies on platform dependent resource files, that is, it requires a Windows resource file for running on the Windows platform, and an X11 resource file for the X11 platform. Resource files defining the graphical layout of the framework dialog for both platforms are part of the asset.

## 5.5. Attribute Editors

The *Prototype* [11, p. 117] pattern is used to provide attribute editors, one for each type of attribute, as outlined in fig. 6. The framework contains a prototype manager in which the client may register instances of `AttributeEditor` subclasses. The prototype manager is basically an associative array that maps from meta-types to corresponding graphical editor instances. Thus, when the user selects an attribute to edit, the framework asks the prototype manager to provide an appropriate editor for that type. The framework of course provides editors for the build-in types, but these can be substituted by registering alternatives if a different graphical interface for editing built-in types is wanted. The attribute editor is responsible for creating an instance of the proper type of node



**Figure 6. Prototype manager for attribute editors.**

class to be entered into the composite expression.

## 5.6. Framework Customisation

To summarise, the client has to go through the following steps to instantiate the framework; some of them may be optional depending on whether the framework has to be extended with new attribute types:

1. *Define meta-model.* The client defines a meta-model by describing domain objects and attributes as instances of `MetaObject` and `MetaAttribute` classes. If the data domain has types beyond those provided by the framework, these are specified as `MetaType` instances. The resulting meta-model is passed as parameter to the framework when it is activated.
2. *Implement node classes and attribute editors for domain specific types (optional).* For domain specific types, the client must define new node types to handle these as well as graphical editors for constructing/editing instances of these new node types.
3. *Implement visitors.* After an editing session the framework returns the composite expression as a tree structure. The client must define at least one visitor, namely the one that can transform the expression tree into an executable search in the domain. E.g. in the case of `EWare` this means writing a visitor that can produce SQL to be executed on the underlying database. If new types of attributes have been added then the client must also implement a pretty printer visitor to produce proper text for the graphical view in pane③ in fig. 1.

## 6. The Process

The framework was first developed in the `IRIS/MFS` project and then later reused in the `EWare` project. The development and deployment processes are sketched in this section, a more thorough presentation is given in [8]. A detailed account of the design and implementation phases is provided in [17] and for the harvesting, maturing and reuse phases in [7].

## 6.1. Design Process

A number of analysis and design meetings produced an initial overall design for the asset. Developers from the IRIS/MFS project (asset producer), the EWare project (asset reuser), and researchers from university participated in these meetings. This approach takes the general reuse recommendation, to consider the requirements of potential future reusers [14], one step further as the reusers were directly involved in the design. This additional cost for the EWare project amounted to about 3–4 man-days.

## 6.2. First Development

The framework was developed in the IRIS/MFS project. It was originally planned that an IRIS/MFS developer who had participated in the design meetings should implement the asset, i.e., be lead framework developer. But due to a rescheduling of the project to meet an earlier delivery date, the task was staffed with four additional developers: three had a master degree in computer science and about 2–3 years of industrial experience with object-oriented programming while the fourth had a master degree in engineering and only about one year of experience with non object-oriented programming. None of these developers had experience with design patterns and because of the severe time-pressure no coordinated training was carried out. In this changed context the use of design patterns became a problem. The overall design, heavily relying on interrelated patterns, proved difficult to understand for the new developers. These difficulties obviously lead to implementations that inadvertently broke the intent (and thus benefit) of the patterns. The difference in the developers' experience showed up clearly; while the experienced OO programmers did come to an understanding after some add-hoc mentoring during the implementation process, the inexperienced developer never really comprehended the underlying ideas.

## 6.3. Harvesting and Maturing

During the harvesting and maturing process the framework was extracted from the IRIS/MFS project, made operational in an isolated test environment, generalised and documented. Due to the inexperience of the additional developers some of the code, though working, did not follow the intent of the underlying design and compromised the intended decoupling. This required quite a bit of code rewriting.

During this phase, the lead framework developer wrote the main documentation. It consisted of a number of implementation scenarios and a small demonstration program showing how to use the framework. The implementation scenarios were based on the concepts of *pattern language* by Johnson [13] and *motifs* by Lajoie and Keller [15]. The documentation consists of a description of when and how the framework is applicable, an overview of the concepts in the framework and finally references to a number of implementation scenarios describing in detail (a) how to solve certain tasks using the framework, and (b) how to customise the framework.

## 6.4. EWare Reuse

The framework was handed over to the EWare project for reuse. The framework was introduced to the EWare developer in a design walkthrough meeting followed by the lead framework developer's suggestions on how to best customise the framework to satisfy the requirements of the EWare customer. The meeting agenda was inspired and driven by the implementation scenarios of the documentation. After the meeting the EWare developer wrote a design document that was reviewed by

the lead framework developer. The implementation process embodied: (a) getting the framework up running in the EWare software context; (b) defining the meta-model appropriate for EWare; and (c) defining the visitor instance that must produce SQL for the underlying relational database.

## 7. Analysis

The case study process was tracked by a research group from University. The primary sources were qualitative sources: development diaries kept by the lead framework developer and the EWare reuse developer, and interviews of the EWare reuse developer during the reuse process. The logged staff-hours were also recorded. A detailed account is given in [7].

### 7.1. Problem Analysis

During the reuse process 19 problems were encountered. Only four of these were classified as domain analysis problems, that is, design- and implementation errors due to an incorrect analysis of the requirements of the two projects. None of these four errors were costly to correct. In contrast 10 problems were classified as technical problems, problems stemming from the technical environment: compiler, preprocessor, makefiles, graphical resource files, libraries depended upon, etc. Three of these were costly to correct in terms of spent staff-hours. The last five problems were primarily attributed to lack of training and were not costly.

Table 1 gives a short description of the 10 technical problems. The mismatch column classifies the mismatch between the framework and the EWare client either as code- or non-code mismatches. Code mismatch means that the problem source was related to the source code of the asset, and non-code mismatch means that the problem source was related to other artifacts that are part of the asset

Most of these problems were relatively easily solved. However, the problems concerning the STL library, makefile adaption, and C++ cast problem were costly in terms of spent staff-hours. The STL library problem was also critical as it would have made the framework useless if a suitable STL library and a patch for the EWare compiler had not been found. The makefile adaption was trivial but tedious; the two make programs were semantically similar but they required different syntax.

One feature of the list is that the number of mismatches due to non-code artifacts equals the number due to source code artifacts. The non-code artifact mismatch problems are due to makefiles, physical file structure, resource files, GUI style conventions, and preprocessing. Though mismatch problems with non-code artifacts have received little attention, they are still important to consider. From an theoretical point of view, we can not claim the framework to be a black-box reusable asset: black-box reuse means reuse as-is, and this should include all aspects of the asset, not just the source code. From a practical point of view the manual adaption of build scripts and resource files for each reuse is error-prone and cost extra time and training. Finally, small problems accumulate and leave developers with an impression of low quality. This may endanger the acceptance of a reuse program within an organisation.

Design patterns and other software engineering techniques allow decoupling at the source-code level, and the source code level decoupling problem has rightly received a lot of attention. However, less attention has seemingly been focussed upon techniques for decoupling and modularisation of non-code artifacts.

**Table 1. Outline of technical problems.**

Problem	Description	Mismatch
	Solution	
Missing STL library	EWare compiler did not provide the STL library required by the framework.	Code
	A public domain implementation was found.	
STL compilation	EWare compiler was unable to compile the found STL library	Code.
	A patch of the compiler was found.	
Cast problem	The framework contained a cast to a subclass in a multiple-fork join inheritance hierarchy that the EWare compiler would not accept.	Code
	The framework was rewritten to use delegation instead of multiple inheritance.	
Compiler incompatibility	The framework contained C++ constructs that were not accepted by the EWare compiler.	Code
	The constructs were rewritten.	
Library incompatibility	The EWare project used collection classes from the RougeWave library. The EWare developer found that modules using both RougeWave and STL did not function correctly.	Code
	Modules were rewritten to use only one of the libraries.	
Makefile adaption	The two projects used different Make programs.	Non-code
	This required an almost complete rewrite of the makefile.	
Include file name clash	The framework and STL library included files with similar names.	Non-code
	Rearrangement of include file search sequence.	
Framework not self-contained	A required header file was not included by the framework itself.	Non-code
	Include directive introduced.	
X11 resource modularisation	The resource file supplied with the demonstration program did not clearly separate resources necessary for the framework from those relevant for the demonstration program.	Non-code
	Lead framework developer acted as consultant.	
GUI style conformance	The look-and-feel of EWare was different from that of IRIS/MFS.	Non-code
	Some of the build-in attribute editors had to be replaced by functionally identical but differently looking editors.	

## 7.2. Reuse Metrics

The EWare project originally made an estimate of 260 staff-hours to implement a search facility from scratch. As the corresponding estimate made in the IRIS/MFS was considerably higher this number may be a bit optimistic.

The actual effort invested by EWare in reusing the framework amounts to 199 staff-hours spent on training, designing, implementing, and testing a solution using the framework, and finally documenting the solution. Additional 79 staff-hours were spent on the outlined technical problems, mainly by the lead framework developer. When one considers the additional 263 staff-hours invested by IRIS/MFS on extracting, maturing, and documenting the framework, it supports accepted wisdom that a reusable asset can not become profitable before it has been reused at least three times. It also underlines that technical problems may be relatively costly even though some of the outlined problems will not reappear in future instances of reuse.

## 8. Lessons Learned

Our experience with design patterns supports that of Beck et. al. [2] and Beck and Johnson [3]. Design patterns were actively used during design to generate the architecture. The points of variability of the framework, the hotspots, were generally expressed in terms of patterns as they provided convenient ways of decoupling the asset from the clients. They served as a medium for high level communication, a common vocabulary, for the architects.

Finally, design patterns served to specify requirements and guiding constraints on the client implementation in areas in which no direct functionality could be provided. In the framework, visitor instances defines the code template for searching the domain data. This solution has been strong enough to handle two highly different methods of searching/filtering. The result is twofold. First, the provided solution is comprehensive. All issues concerning searching are considered by the framework; either directly through providing functionality or indirectly through providing templates and thereby guidance. Next, it provides better support for developers with little knowledge of the domain of searching. They do not have to “reinvent the wheel” but can rely on a well-proven template. This issue can be further strengthened by providing good documentation and examples for common situations such as SQL generation.

The reuse of the framework had another, less obvious, benefit, namely that it facilitated transferring of software engineering knowledge in the organisation. The reuse developer had no previous experience with design patterns but the initial design walkthrough meeting followed by the concrete work of reusing the search framework made him become intimately acquainted with a small set of them. To many developers applying a new technique and being rewarded immediately by a working program is a more fun way of being introduced to a new subject than an abstract introduction such as reading a book. The reuse developer reported design pattern knowledge as his main benefit of the exercise.

Our case study, however, also underlines that the use of design patterns must be accompanied by appropriate training and mentoring. The influx of new developers that had no experience with design patterns during the initial implementation resulted in wasted effort. The newcomers were unable to produce code that followed the intent of the patterns and much code therefore had to be rewritten.

An reusable asset embodies more than just the source code and architectural mismatches between assets can stem from other than syntactic and semantic sources at the source code level. In our case, effort had to be invested in adapting the makefiles and user interface resource files, a process somewhat similar to the “recycling of non-code artifacts” reported by Allen et. al. [1]. Instances of such *technical mismatch* may lower reusability as required, non-code, artifacts have to be adapted for every (re)use—essentially depriving the asset of being a black-box asset. One obstacle is the “make” program itself [10], which is inherently non-modular, making it difficult to decouple the build descriptions from the particular physical domain. The same applies for graphical resource files.

Technical mismatch problems seems to have received little attention compared to other mismatch problems such as packaging mismatch. In the paper by Garlan et al. [12] the term “architectural mismatch” covers problems of incompatible build scripts, but later papers focus on the problems of mismatch in a more narrow sense as differences in interfaces, data formats, push/pull models of control flow, etc. [4, 9, 19]. The standard “solution” for technical mismatch problems is that of *avoidance*: “Standardise tools and environments”. However, various stake-holder influences such as customer requirements, legacy systems, multi-platform support, etc., may inhibit standardisation. We find that there is a need for technical patterns or idioms for decoupling non-code artifacts such

as makefiles, resource files, etc. Binary component technology may be the solution in the future, but at present the competing technologies themselves present a considerable mismatch problem.

The EWare developer reported that reusing the framework was a success. The framework provided all the functionality required by the EWare customer and provided a sound architecture for solving the search problem. He did not find it difficult to understand nor customize; this should be seen in the light that he was given a detailed introduction to the framework and that the lead framework developer was available as consultant. He reported, however, that it was heavy-weight in the sense that many issues had to be understood and resolved before feedback was given by a running program. We interpret this as the framework has reached its limits of generality; as Mili points out generality and immediacy are contravariant properties of assets [16]. Thus making the framework even more general will make it less immediate to use—to the point that it would become “reuseless” as the investment required to understand and use it will become prohibitive.

## 9. Summary

We have presented a case study of designing, implementing, and reusing a framework for searching arbitrary data domains. We have presented the challenges faced by the framework, a design that addresses the challenges, and described and analysed the problems encountered in the reuse phase with emphasis on technical problems. A key observation is that technical problems should not be underestimated and that mismatch problems between a reusable asset and client applications stem both from problems at the source code level as well as at the non-code artifact level. While the software engineering community has rightly invested much effort in techniques for decoupling at the source code level, we find there is also a need for decoupling techniques at the non-code level—a need that has received considerably less attention.

## Acknowledgements

This research has been funded by the *Centre for Object Technology* [6]. The Centre is financially supported by The Danish National Centre for IT-Research (CIT) and the Danish Ministry of Industry.

## References

- [1] Jeffrey Allen and Larry Latour. If you Can't Reuse, Recycle: A Case Study of a Platform to Platform Port. In *Eighth Annual Workshop on Institutionalizing Software Reuse*, 1997. URL: <http://www.umcs.maine.edu/~ftp/wisr/wisr8/wisr8.html>.
- [2] Kent Beck, James O. Coplien, Ron Crocker, Lutz Dominick, Gerard Meszarus, Frances Paulisch, and John Vlissides. Industrial Experience with Design Patterns. In *Proceedings of the 18th International Conference on Software Engineering*, pages 103–114. IEEE Computer Society Press, 1996.
- [3] Kent Beck and Ralph E. Johnson. Patterns Generate Architectures. In *European Conference on Object-Oriented Programming, ECOOP*, pages 139–149, 1994.
- [4] Francois Bronsard, Douglas Bryan, W. (Voytek) Kozaczynski, Edy S. Liongosari, Jim Q. Ning, Asgeir Olafsson, and John W. Wetterstrand. Towards software plug-and-play. In *Proceedings of the 1997 Symposium on software reusability SSR'97*, pages 19–29, Boston, United States, May 1997.
- [5] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-oriented Software Architecture*. John Wiley and Sons, 1996.
- [6] Centre for Object Technology. *COT home page*. <http://www.cit.dk/COT>.

- [7] Henrik Bærbak Christensen and Henrik Røn. Case Study of Maturing and Reusing a Framework. Technical Report 3-32, Centre for Object Technology, April 2000. URL: <http://www.cit.dk/cot>.
- [8] Henrik Brbak Christensen and Henrik Rn. A Case Study of Horizontal Reuse in a Project-Driven Organisation. In *Proceedings of 7th Asia-Pacific Software Engineering Conference APSEC 2000*. IEEE Computer Society Press, 2000. To appear.
- [9] Robert DeLine. A Catalog of Techniques for Resolving Packaging Mismatch. In *Proceedings of the Fifth Symposium on Software Reusability*, pages 44–53, Los Angeles, CA, USA, May 1999. ACM SIGSOFT, ACM Press.
- [10] Stuart I. Feldman. Make – A Program for Maintaining Computer Programs. *Software – Practice and Experience*, April 1979.
- [11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reuseable Object-Oriented Software*. Addison-Wesley, 1994.
- [12] David Garlan, Robert Allen, and John Ockerbloom. Architectural Mismatch, or, Why it’s hard to build systems out of existing parts. In *Proceedings of the 17th International Conference on Software Engineering*, pages 179–185, Seattle, Washington, April 1995.
- [13] Ralph Johnson. Documenting frameworks using patterns. In *OOPSLA 92*, pages 63–76. ACM, 1992.
- [14] Even-Andre Karlsson. *Software Reuse – A Holistic Approach*. John Wiley and Sons, 1995.
- [15] Richard Lajoie and Rudolf K. Keller. Design and Reuse in Object-Oriented Frameworks: Patterns, Contracts, and Motifs in Concert. In *Proceedings of the 62nd Congress of the Association Canadienne Francaise pour l’Avancement des Sciences (ACFAS)*, Montreal, Canada, May 1994. Colloquium on Object Orientation in Databases and Software Engineering.
- [16] Ali Mili, Sherif Yacoub, Edward Addy, and Hafedh Mili. Toward an Engineering Discipline of Software Reuse. *IEEE Software*, 1999.
- [17] Henrik Røn. Development of a Component for Searching and Filtering. Technical Report 3-20, Centre for Object Technology, May 1999. URL: <http://www.cit.dk/cot>.
- [18] Mansur Samadzadeh and Mansour Zand, editors. *Proceedings of the 1995 Symposium on software reusability SSR’95*, Software Engineering Notes, Seattle, Washington, Aug 1995. ACM, ACM Press.
- [19] Mary Shaw. Architectural Issues in Software Reuse: It’s Not Just the Functionality, It’s the Packaging. In Samadzadeh and Zand [18], pages 3–6.
- [20] Systematic Software Engineering A/S. *Additional product information on IRIS/MFS and EWare*. <http://www.systematic.dk/products/products.html>.