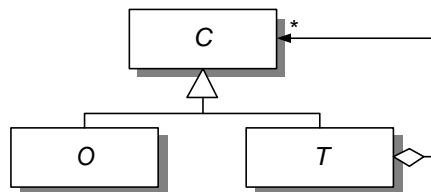


*Case Study of
Maturing and Reusing
a Framework
COT/3-32-V1.0*



Centre for Object Technology

*Centre for
Object Technology*

Revision history: V1.0 25-04-2000 Final version for publication

Author(s): Henrik Bærbak Christensen, University of Aarhus
Henrik Røn, Systematic Software Engineering A/S

Status: Final

Publication: Public

Summary:

This report describes the process of harvesting and maturing a component into a white-box framework, followed by the successful reuse of the framework in another context. The framework can be used to implement search within any domain. The report is a follow-up report on another COT report [COT/3-20], which describes initial domain analysis, design, and implementation of the component. Familiarity with [COT/3-20] is an advantage, but not a necessity.

The report describes the work process and its products, followed by a categorisation of the problems encountered, and ends a summation of key experiences, conclusions, and lessons learned.

© Copyright 2000 Systematic Software Engineering A/S and University of Aarhus

1. INTRODUCTION.....	4
1.1 EXECUTIVE SUMMARY	5
1.1.1 Purpose of the Report.....	5
1.2 MOTIVATION.....	6
1.3 WORK METHOD	6
1.4 TERMINOLOGY	6
1.5 OUTLINE OF THE REPORT	7
2. SIMPLE DESCRIPTION OF THE FRAMEWORK.....	8
3. PROCESS OVERVIEW.....	11
3.1 PROCESS ISSUES: ITERATIVE DEVELOPMENT	12
4. THE HARVESTING PROCESS.....	13
4.1 REMOVING IRIS/MFS DEPENDENCIES.....	13
4.2 DISCUSSION.....	13
5. THE MATURING PROCESS.....	14
5.1 IDENTIFICATION OF DOMAIN INDEPENDENT CHANGES	14
5.2 BEST PRACTICES FOR PHYSICAL STRUCTURE	16
5.3 RETROFITTING IN IRIS/MFS.....	16
5.4 CUSTOMISATION OF THE FRAMEWORK.....	16
6. DOCUMENTING THE FRAMEWORK.....	17
6.1 USAGE DOCUMENTATION	17
6.2 OVERVIEW DOCUMENTATION	18
6.3 INTRODUCTION SEMINAR.....	18
7. THE EWARE FITTING PROCESS	19
7.1 INITIAL FITTING	19
7.2 DESIGN CHANGES RESULTING FROM FITTING PROCESS	20
7.3 FINE-TUNING EWARE IMPLEMENTATION.....	20
7.4 DOCUMENTATION UPDATE	20
7.5 FRAMEWORK USAGE IN IRIS/MFS AND EWARE	21
8. DATA ACQUISITION AND ANALYSIS	21
8.1 STAFF-HOURS LOG.....	22
8.1.1 Cost Avoidance Analysis for EWare Reuse.....	22
8.2 CLASSIFICATION OF EWARE PROBLEMS	24
8.2.1 Analysis.....	25
8.3 SCOPE OF ANALYSIS	26
8.4 SOURCE OF ERRORS	26
9. CONCLUSIONS	27
10. REFERENCES.....	29
A. ABBREVIATIONS AND DEFINITIONS.....	31
B. THE PHYSICAL PROPERTIES OF THE FRAMEWORK	31
C. DETAILED DESCRIPTION OF DESIGN CHANGES.....	32
D. PROBLEMS ENCOUNTERED BY EWARE	38

1. Introduction

This report describes phase three of the pilot project initiated when Systematic Software Engineering A/S, hereafter referred to as SSE, joined the Centre for Object Technology. Phases one and two are described in detail in [COT/3-20] and will only be outlined briefly in the present report. In Figure 1 below is depicted the schedule for the entire pilot project; a detailed schedule for phase three is depicted and the individual phases are described in Section 3. .

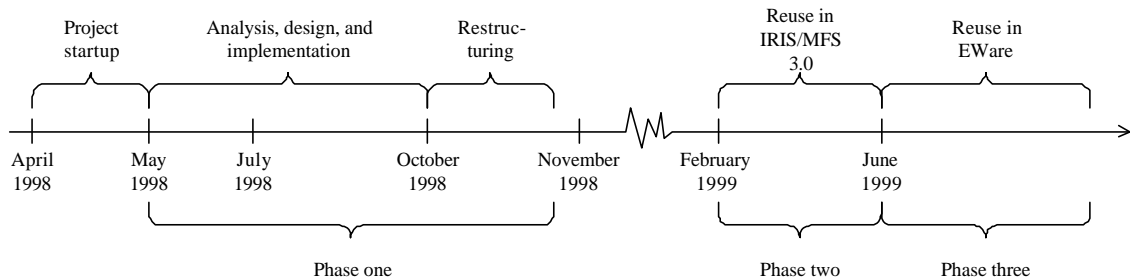


Figure 1 - The Pilot Project Schedule

In phase one of the pilot project, a software component for searching and filtering was developed as part of IRIS/MFS, which is a SSE product. In phase two, it was reused in other part of IRIS/MFS. As this use of the component was not originally envisioned, it represents a first, albeit not particular significant, test of the generality of the component.

In phase three, described in the present report, the component has been harvested and matured further into a white-box¹ framework, i.e., made independent of IRIS/MFS, and reused in a completely new context, the SSE product EWare. The focus of the report is on the issues that have arisen during the harvesting and maturing processes. We also consider it important to give a detailed account of issues pertinent to successful reuse through hands-on experience in a concrete case at SSE. The report classifies all encountered problems during the harvesting and maturing processes and establishes a number of conclusions that highlight problems and benefits with respect to reusing component in different contexts. Finally, the appendices contain a detailed technical account of encountered design issues and problems.

The notation used in the software diagrams throughout the report is UML [Booch et al. 99]. The term “component” is used to denote “a well-defined piece of software intended to be reused”. The term therefore encompasses, but is not restricted to, the currently popular interpretation of the term as meaning a “binary component”, in technologies such as COM, CORBA, or Java Beans.

¹ “A *framework* is a set of classes that embodies an abstract design for solutions to a family of related problems” [Johnson and Foote 88]. “In a *white-box* (inheritance-based) framework, the framework user is supposed to customise the framework behaviour through sub-classing of framework classes” [Bosch et al. 96].

1.1 Executive Summary

This report describes the process of harvesting and maturing a component and reusing it in a new context. The motivation of the project has primarily been to get first-hand experience with issues pertaining to reuse of components in multiple contexts, and thereby relate the theory of reuse to the concrete context of SSE. Secondly, the pilot project has had the working hypothesis that the involvement of potential reusers in the design of a component will increase its reusability. The increased reusability should manifest itself as a reduced cost of using the component in the new context compared to reusing a similar component where potential reusers had not been involved during the design. This is of course speculative as we have not carried out a parallel experiment. This report represents phase three of the pilot project, experiences from phase one and two are documented in [COT/3-20]. Familiarity with [COT/3-20] is an advantage, but it is not a necessity to read and understand this report.

The report also describes the produced work-products. Finally, we analyse the problems encountered during the process, define a classification for them and rank them according to the amount of resources required to fix them.

Based on the data gathered during phase three, the report concludes the validity of the working hypothesis, and shows that there is both a domain cost and a technical cost of reusing a component. By “technical cost” we mean cost resulting from technical issues, and by “domain cost” we mean the cost resulting from issues related from the domain. For our component, the involvement of the reusers in the design made the domain cost low: The provided architecture and functionality was sound and sufficient for their context. However, the technical costs were significant (especially development environment differences caused trouble) and this underlines the necessity of a common development environment for profitable component-based reuse. The term “development environment” is used in a broad sense as it merely covers basic necessities such as a common understanding of code and file layout, how systems are build (e.g. make program and directory structure expected by make), guidelines for user interface style, and so forth. Our use of “development environment” does not encompass choice of compiler, editor, case tool, etc. vendor or version. A number of other conclusions are outlined in Section 9. ; these are more detailed but never-the-less important in the concrete context of a development project.

1.1.1 Purpose of the Report

The purpose of this report is twofold: Its primary purpose is to describe the process of harvesting and maturing the component into a framework and reusing it in the EWare project. Secondly, to reflect on problems, experiences, and lessons learned from the process.

The report is relevant for two audiences within organisations developing software: Developers interested in maturing and generalising components into a framework, and managers interested in reuse, especially the practical problems that can be encountered.

Developers may want to focus on Sections 2. through 1. 7. and 9. , while managers may want to focus on Sections 3. through 5. , 8. and 9.

1.2 Motivation

The primary motivation for the pilot project was:

To get first-hand experience with the practical issues pertaining to harvesting and maturing a software component so it can be reused in a new context.

There are many acclaimed books on the topic of reuse. From a practitioners point of view, however, they necessarily must suffer the problem that they must present their conclusions and recommendations in rather abstract terms that are difficult to apply directly in a given organisation. The pilot study allowed the theory to be related to a concrete context.

Secondly, as a potential client of the search component was known at an early stage, the pilot project adopted the following as its working hypothesis:

Involving a potential reuser in the domain analysis and design of a component would increase its reusability within other contexts.

This is arguably a known fact within the reuse community [Karlsson 95, p. 257-262], however, the exercise was primarily to realise this statement and test it in practice in the context of SSE. A final goal was to try and use state-of-the art techniques for designing and developing software.

1.3 Work Method

The work method used in phase three differed from that of the first two phases. In the first two phases, work was done by a group of developers, see [COT/3-20] for details, whereas in the third phase, work was done by a single developer from IRIS/MFS. This developer did the redesign, implementation, harvesting, and documentation. This was followed by the implementation in EWare by an EWare developer using the framework and documentation developed by the IRIS/MFS developer. The IRIS/MFS developer acted as a consultant for the EWare developer throughout the process.

1.4 Terminology

In the report, references will be made to processes specific to the context of reuse. These are defined as follows:

- **Asset.** An asset is a high-quality product or by-product of the software process that is considered reusable. In other words the product is considered an asset for the organisation owning it. See [Jacobson et al. 97, p. 425] for alternative definitions of asset.
- **Harvesting:** The process of extracting an asset from a context in such a way that the asset

- no longer has any references to entities specific to the context it came from, for example application specific data-types.
- can be made operational in a context different from the original.
- has separate build information (e.g. makefiles), resources and other technical prerequisites for generating the asset in binary form, independent from the original context's technical environment.
- **Maturing:** Denotes a set of sub-processes that are performed on an asset after it has been harvested.
 - Redesigning the asset in order to make it more flexible, general, and adaptable. One example of a redesign activity is the definition of hotspots [Pree 95] (also called hooks) that allow the asset to be parameterised or in other ways adaptable to specific contexts.
 - Documenting the asset.
 - Ensuring that the asset complies with guidelines for code structure (as for example [Lakos 96] or company defined standards). These guidelines go beyond normal code standards, which mostly deal with the appearance and naming conventions of the code, not with its physical layout.
 - Incorporating changes based on observations from the fitting and retrofitting processes.

Maturing is like maintenance; a process that is not terminated until no one uses the asset anymore. However, experience suggests that assets are relatively stable after the third fit.

- **Fitting:** The process of reusing an asset in another context than the one it was originally developed in.
- **Retrofitting:** The process of making an asset operational in its original context after it has been harvested and matured.

1.5 Outline of the Report

A short overview of the developed framework is given in Section 2. . This is followed by an overview of the process in Section 3. . The actual work of harvesting, maturing, and documenting the framework is described in Sections 4. , 5. , and 6. respectively. Then in Section 7. it is described how EWare uses the framework and how this affected IRIS/MFS. Data were gathered during the entire process and Section 8. describes the kind of data gathered and contains two analyses, one of the profitability of the reuse and another of the problems encountered when reusing the framework. Finally Section 9. contains conclusions and references are in Section 10. . Appendices contain detail code metrics and a detailed account of design issues and encountered problems.

2. Simple Description of the Framework

In the following, a short description of the framework will be given. The framework is designed to be part of a larger application within an arbitrary domain. It is used to implement capabilities for constructing expressions describing some data present in the system. The framework provides a GUI for the construction of this expression. The expression is constructed by combining atomic statements for the attributes of the objects in the search domain. The constructed expression can then be used for a purpose suitable for the applications specific needs. In IRIS/MFS messages are matched against the expression during searching or filtering of messages. In EWare the expression is used to generate SQL used for searching the relational database underlying EWare.

Figure 2 shows the graphical user interface of the framework. The picklist marked ① contains the searchable attributes of the domain. To the left of the picklist is an editor (marked ②) for the attribute type that is selected in the picklist, in this case a statement containing dates (or timestamps). The lower left part of the window (marked ③) contains a “natural” language representation of the search expression. The example in ③ consists of four atomic expressions. To the left of the expression window are buttons (marked ④) for adding, substituting, or removing the contents of the “Simple Editor” to the tree and thus to the textual representation in ③.

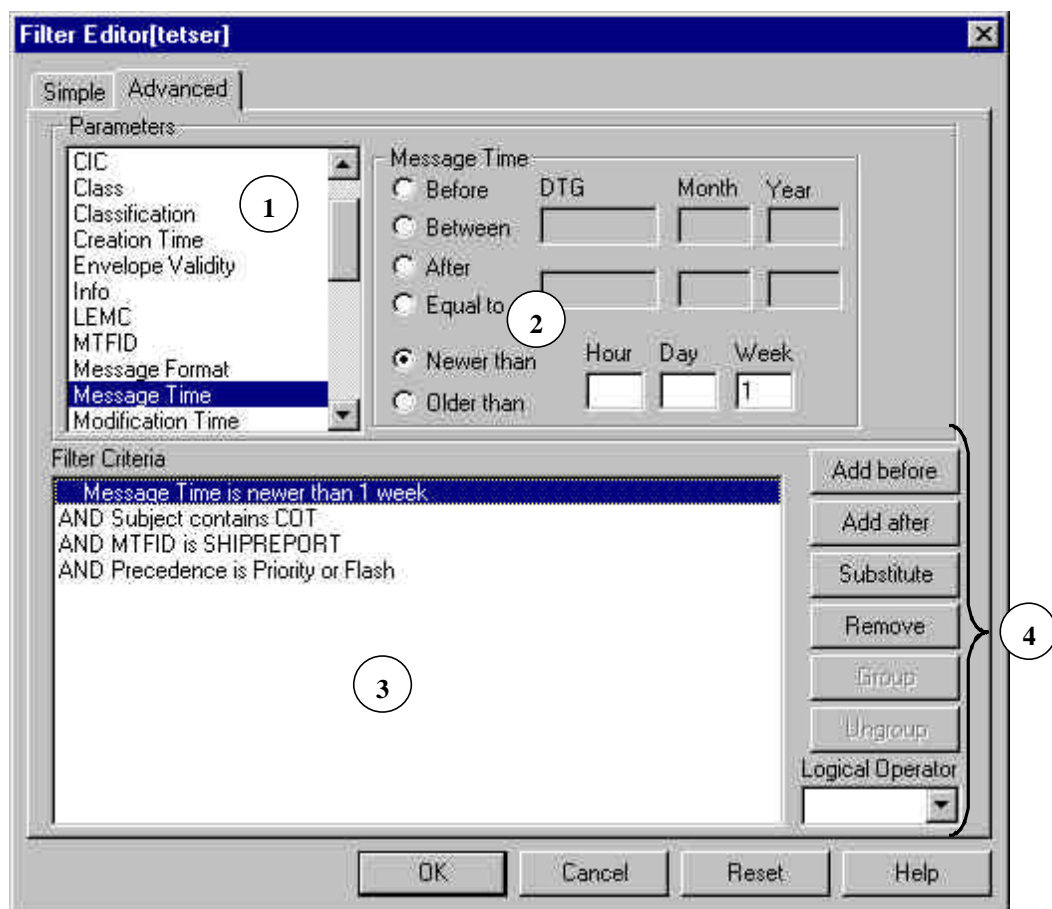


Figure 2 – The user interface for constructing a search expression

The domain independence of the framework is ensured by using a meta-model for an abstract description of the data-model of the domain. The framework operates using the meta-model instead of operating directly on the domain data. The framework returns the query as a tree. Basically the framework has the following flow of data:

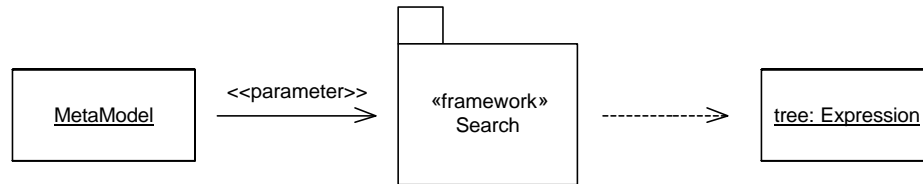


Figure 3 - The basic flow of data in the framework

In the following, it will be explained what a meta-model is and how the tree is structured. A meta-model consists of one or more meta-object(s). For each type of object in the domain model there is a corresponding meta-object. Figure 4 depicts the data model of a fictitious application. This data-model consists of a “Person” class.

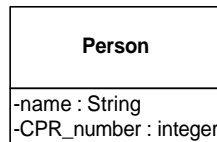


Figure 4 - Domain model

To describe the attributes of objects, meta-objects have meta-attributes. Each meta-attribute has a type, like string or number. The classes that implement meta-objects and meta-attributes are illustrated in Figure 5 below.

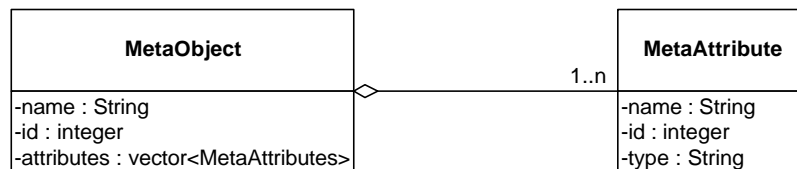


Figure 5 - Class model for meta-information

During run-time, the meta-model for the domain model in Figure 4 will be:

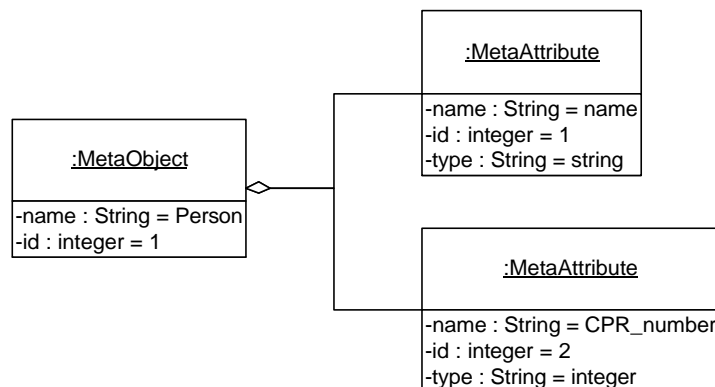


Figure 6 - Run-time snapshot of meta-model for domain model

Internally, the expression is represented as a tree. The tree is structures according to the “Composite” pattern from [Gamma et al. 95] as illustrated in Figure 7. The “ParameterType” is the abstract base class for type nodes. There is a class for each type of meta-attribute. These type nodes are the leafs of the tree and each instance of a type node represents one statement for an attribute of that type, e.g., a “StringNode” for one statement about a string. All leaf nodes combine to the full expression shown in ③. The tree corresponding to the expression in Figure 2, part ③, has four leaf nodes, one “TimeNode” for the first line, a “StringNode” for the second line, and two “SetNodes”, one for each of the two last lines.

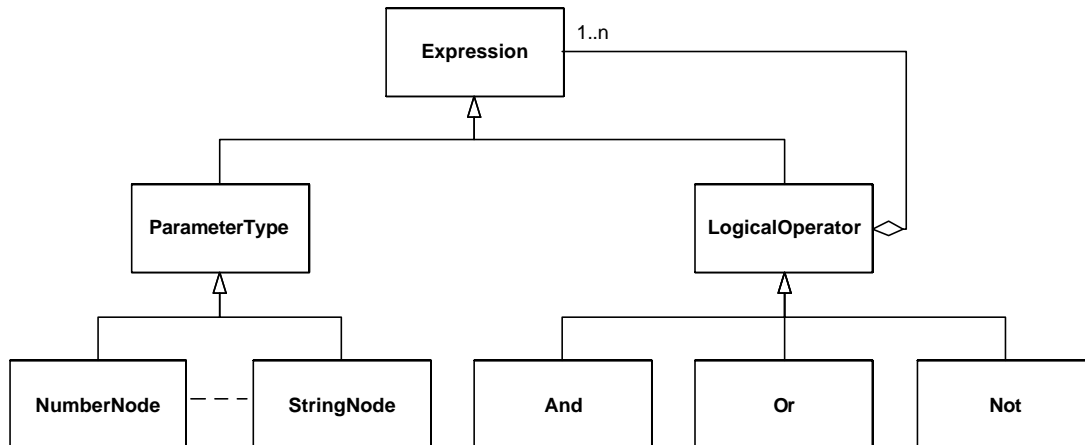


Figure 7 – The classes that form nodes in a search tree

Each type has an editor associated. The editor is used to build a statement for the type, for instance a “StringEditor” for defining a string statement. The “LogicalOperator” class is the abstract base class for the operator nodes in the tree, i.e. “and”, “or”, “not”. The framework provides graphical editors for some build-in types, e.g., integers, strings, dates, and as well as components for constructing and manipulating the data-structure that represents the expression internally in the framework.

The user interface is coupled with the meta-model and the tree in the following way: Each entry in ① has the corresponding “MetaAttribute” associated, e.g., the selected entry “Message Time” has a meta-attribute associated where the attribute “type” has the value “Time”. When an entry is selected, the user-interface determines which editor to present by using the meta-attribute. If the user double-clicks on a line in ③, the leaf node corresponding to the clicked line is used to determine the editor to display in ② again by using a meta-attribute; thereafter the editor is loaded with the data for the statement. When the user presses one of the buttons (④) the entered input is validated by the “MetaAttribute” and if the input is semantically correct, a leaf node is inserted/replaced in the tree.

3. Process Overview

A detailed schedule for phase three is shown in Figure 8 below. The allocated resources (developers) have not worked full-time on the project. The phase is divided into a number of sub-phases:

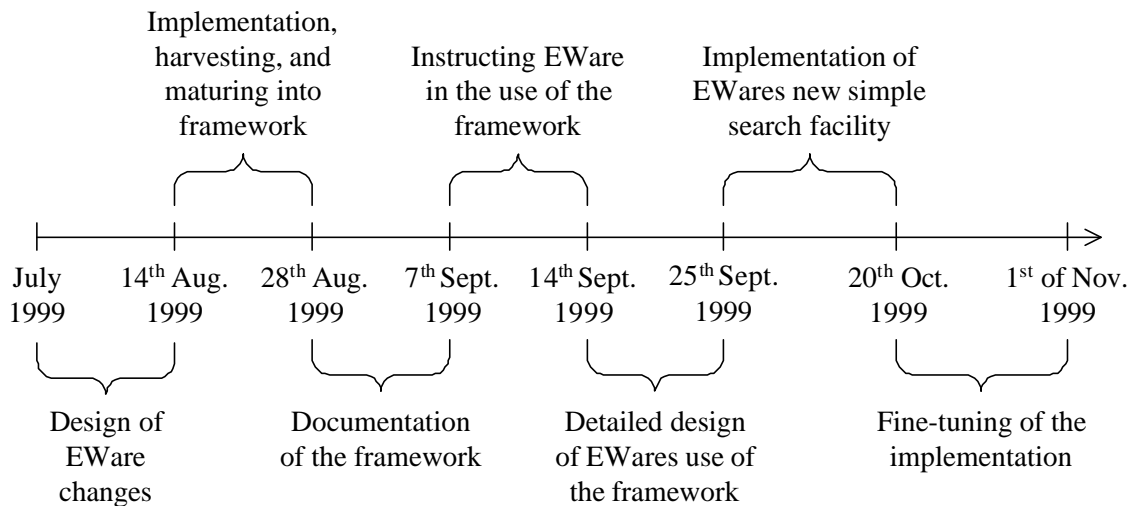


Figure 8 - Schedule of phase three

1. **Design of changes necessary for EWare.** After implementation had been completed in IRIS/MFS, a number of changes necessary to make the component reusable were identified. The changes were on the architectural level as they mainly dealt with the customisation of the framework. That is the changes were related to the instantiation of the framework and how functionality specific to the application domain is added.
2. **Implementation of changes combined with harvesting and maturing.** A number of different tasks were carried out in an iterative process with three main activities, which are interleaved:
 - i. Making the component independent of IRIS/MFS.
 - ii. Separate build information for the framework.
 - iii. Restructuring the physical layout of the files making up the component following guidelines from [Lakos 96].
 - iv. Implementing design of EWare changes.

Activities i and ii correspond to the “Harvesting”, while activities iii and iv, which correspond to the “Maturing” except that documentation was performed subsequently. Activities iii and iv resulted in the transformation of the component into a white-box framework. This framework is customised partly by sub-classing and partly by parameterisation. Both “Harvesting” and “Maturing” are defined in Section 1.4. It is described later in the report, which parts that were customised by sub-classing and which by parameterisation.

3. **Documentation of search framework.** This was done using the “pattern language” technique described in [Johnson 92] combined with a description of the technical properties of the framework, i.e., programming language, compiler, etc.

4. **Instruct EWare in the use of the framework.** Before the EWare developers began their design, there was an introduction meeting. Before the meeting, the developers received excerpts from [Gamma et al. 95] and [Buschmann et al. 96] describing the patterns used in the framework. During the meeting the EWare developers were introduced to the framework, encouraged to ask questions, and the framework developer outlined a possible implementation strategy for EWare.
5. **EWare design of search.** EWare wrote a design document for the new simplified search facility and how it should be integrated in their product. The design document contained description of all new EWare specific types, editors, and functionality. The design included a number of reuse requirements, e.g., that certain parts of the EWare code only could depend on the framework, Motif, and STL in order to make these reusable to others.
6. **EWare implementation of search.** EWare proceeded to implement their design by adding new classes to instantiate the framework. A couple of changes to the framework were also necessary during the implementation. These changes were on a lower level of abstraction compared to the changes described in 1. The changes requested by EWare were on a “detailed design” level of abstraction. After each of these changes the framework was retrofitted in IRIS/MFS.
7. **EWare fine-tuning of the graphical user interface and SQL generation.** The last phase of the implementation was fine-tuning the GUI of the EWare specific “Simple Editors” and the SQL generation from the search expression tree.

3.1 Process Issues: Iterative Development

The process of designing the EWare changes, implementing them, harvesting the component, and maturing it into a framework, was not performed as a waterfall process, but as a highly iterative process, where the IRIS/MFS developer altered between four different activities several times during a day. This is illustrated in Figure 9 below.

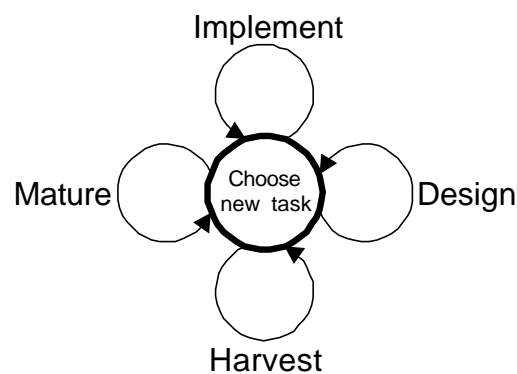


Figure 9 - The iterative development cycle

In our opinion, the iterative process is necessary because it is impossible to foresee all changes and their consequences at once. One design change will necessitate other design changes and so on.

4. The Harvesting Process

In this section, we describe each of the sub-processes of the harvesting process, as defined in Section 1.4.

4.1 Removing IRIS/MFS Dependencies

The component was originally a part of IRIS/MFS and had a number of dependencies on other parts of IRIS/MFS, e.g., data types, modules, and servers. A detailed discussion of the exact nature of these dependencies can be found in Appendix C. The work of making the search component independent of IRIS/MFS consisted of two disjoint tasks:

- **Define the parts that are going to be reusable.**
 - Classifying the original node types as either IRIS/MFS specific or to be supplied by the framework. Examples of reusable node types are “TimeNode” and “StringNode”, but “CodeNode”, which is used for storing three letter codes specific to the IRIS/MFS domain, was not deemed reusable.
 - Restructuring the code so that the IRIS/MFS specific node types and the code that operates upon them is separated from the reusable code.
- **Dependencies on the rest of IRIS/MFS.**
 - Identifying dependencies to the rest of IRIS/MFS.
 - Deciding which of the modules, data types, etc. depended upon should also be harvested and matured.

As the search component originally was developed within IRIS/MFS, a number of IRIS/MFS dependencies had been introduced in the code. These dependencies were:

- **Servers.** The IRIS/MFS servers were used for a number of tasks in the original implementation. However, these server calls make the component dependent on IRIS/MFS and therefore the calls have to be eliminated. This can be done by encapsulating the calls in an abstract class and making this class a customisation parameter. This is basically an application of the Strategy pattern from [Gamma et al. 95]. It is then the responsibility of the reuser to provide a strategy for his/her application.
- **Data-structures.** During development, a number of IRIS/MFS specific data-structures were used. If the code that uses these data-structures is scheduled to be harvested and matured these dependencies must be eliminated; either by replacing the IRIS/MFS specific data-structures with general data-structures with the same functionality or harvesting and maturing the data-structures in question.

4.2 Discussion

The main conclusion on this part of the process is that it should be stated explicitly which parts of a module that are scheduled to be harvested and matured. Much of the

work described in this section could have been avoided if there, during the initial development in IRIS/MFS, had been focus on:

- Which sub-components of the component will be reused?
- How the sub-components are organised in files.

A recommendation is to avoid using project specific data-structures and functionality in these sub-components. There should be guidelines describing what to do if such use of project specifics is unavoidable, e.g., encapsulate the use of project specific functionality in a class, by using e.g. the Strategy pattern from [Gamma et al. 95]. The different users of the framework can then write their own strategies for supplying the functionality.

5. The Maturing Process

The maturing process started by identifying the changes needed by EWare. These changes were identified in a meeting with the EWare developer who had participated in the original analysis and design of the search component. The changes were designed and discussed at a brainstorm meeting.

Only the parts of the IRIS/MFS code needed by EWare were harvested and matured, i.e., a demand-driven strategy. Our argument is that harvesting and maturing only the parts needed avoids “featuritis” [Brooks 96, p. 258]: Components are over-generalised and have too many features that are not needed by the users and actually reduce the reusability of the component. This is also known as the Mini-AntiPattern “Swiss Army Knife” from [Brown et al. 98, p. 197].

As outlined in section 3.1, it is difficult to distinguish sharply between the process of “maturing”, “harvesting”, and “retrofitting/implementing” framework changes. This is reflected in the sections below, which often describes all three processes as a single process.

5.1 Identification of Domain Independent Changes

This subsection contains an outline of the changes needed for EWare. During the brainstorm, the following changes were identified:

- **Ability to register “Simple Editors”.** In the IRIS/MFS implementation there was no mechanism for registering “Simple Editors” in the “GUIFactory”, i.e. specifying which “Simple Editors” that are used for an attribute with a given meta-type, e.g., the “Subject” attribute of an IRIS/MFS message could have the meta-type “String”. In IRIS/MFS it was fixed which “Simple Editors” the framework was instantiated with and which types the “Simple Editors” operated on.
The “GUIFactory” is responsible for returning the “Simple Editor” suited for a certain type of attribute, given the attribute’s type id. See Appendix C.1 for details. After the harvesting the “GUIFactory” has become one of the frameworks instantiation parameters and it is used for customisation of the framework.

- **Validation of data entered in a “Simple Editor”.** EWare would like the ability to validate the data entered in a “Simple editor”. In the IRIS/MFS implementation the validation was located in the “Simple editors” themselves. This was not good for two reasons: Primarily because the validation rules were specific to IRIS/MFS, and secondly they could not be changed without subclassing and rewriting some of the methods.

During the design of the EWare changes, a number of other changes unrelated to EWare were identified. The changes are needed to improve reusability and correct inexpedient design decisions made in the original implementation:

- **Loading and saving in a “Simple Editor”.** In the IRIS/MFS design it is the “Simple Editor” itself that can load (or save) its contents from (or into) a node. This strongly couples a simple editor to a particular type of node, e.g., the “Simple Editor” for numbers could only be used for integers and not reals, as its “Load” and “Save” methods assumed an “NumberNode”, i.e., an integer node². The “Simple Editor” had to be independent of its “Load” and “Save” methods so, e.g., the “NumberEditor” could be used for both integers and reals.
- **Collapse hierarchy for node types.** One of the original design decisions was that not only different types should be represented by a class, but also the different kinds of types, e.g., number between, number less than, number greater than, etc. This is illustrated in Figure 10. This overly elaborate design was also used for time nodes. For both types, the class hierarchy was collapsed, so that there now only is one class representing each of the types.

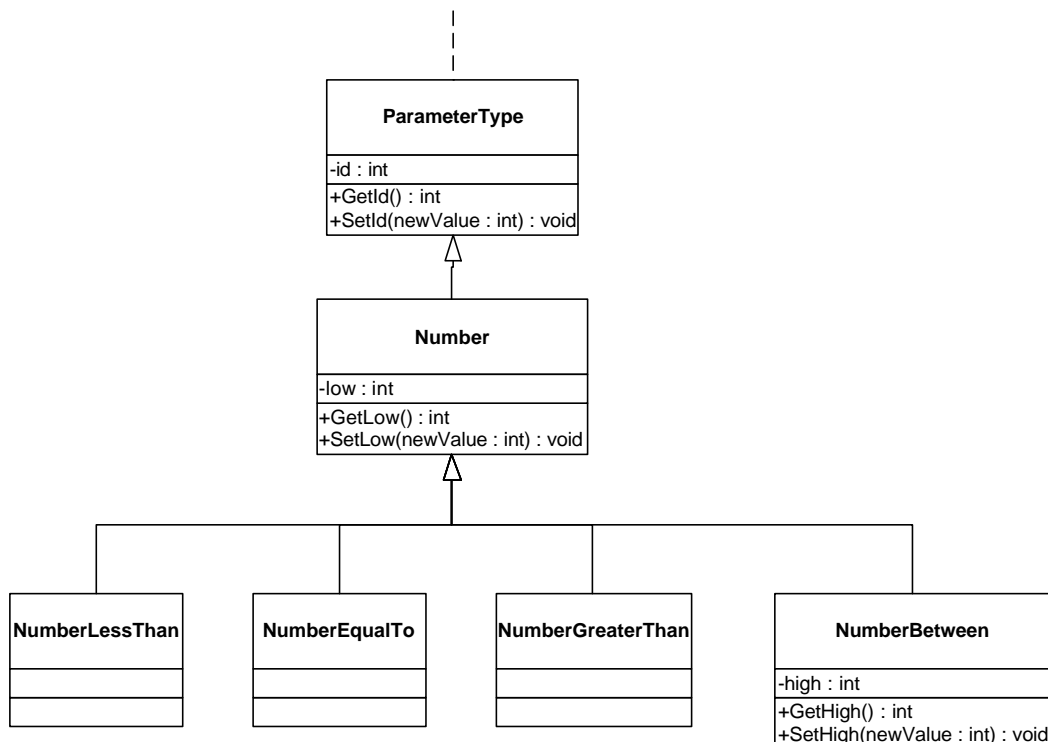


Figure 10 – Example of overworked class hierarchy in IRIS/MFS component

A detailed description of their implementation is found in Appendix C.

² This is an example of unfortunate naming stemming from the IRIS/MFS implementation, as the “NumberNode” class should have been named “IntegerNode” as it can only hold integers.

5.2 Best Practices for Physical Structure

Some of the guidelines from [Lakos 96] for improving the physical structure of C++ programs were followed during the harvesting. These suggestions go beyond the normal coding standard, which mostly concern themselves with the code “looks”, but not how it is structured and organised. The guidelines used were:

- Keep class member variables private and provide accessor methods. This has been recommended practice within the object oriented community for many years.
- Avoid pollution of the global name space. This is done by not using enumerations, typedefs, and constants at file scope. These are instead encapsulated in classes.
- Avoid using pre-processor macros.
- Rules for what may be declared at file scope.
- Place guard around contents of each header file. This has also been recommended practice for many years in order to avoid multiple processing of the same header file.
- Physical design. According to Lakos logical design addresses only architectural design. But there is a need for a “physical design” that addresses organisational issues, e.g., work break-down and division of labour. These guidelines are described in [Lakos 96, chapter 3].

An important point concerning this part of the maturing process is that most of this restructuring could have been avoided if these, from an intellectual point of view, trivial guidelines had been followed originally.

5.3 Retrofitting in IRIS/MFS

The design changes described in Section 5.1 affected the IRIS/MFS implementation as some functionality could now be implemented using the new features. Thus, some of the design changes required by EWare caused the IRIS/MFS implementation to change:

- **Handling default values for meta-attributes.** Previously the default values for some attributes were read into the “Simple Editors” when these were initialised and this strongly coupled the “Simple Editor” to the IRIS/MFS domain. After the “Simple Editors” were separated completely from IRIS/MFS a new way of storing default values was needed. This is described in detail in Appendix C.6.
- **Moving the non-reusable IRIS/MFS code.** Most of the design changes necessitated moving IRIS/MFS specific code to new files or from one existing file to another.

5.4 Customisation of the Framework

The maturing of the component into a framework occurred gradually and unconsciously during the harvesting, maturing, and implementation phase. It was only realised in retrospect that the component had evolved into a white-box framework. In the framework hotspots, i.e. the points of variability, are specified in two ways:

- **Customising by sub-classing.** The user adds new types by sub-classing the “ParameterType” class and makes “Simple Editors” for these new types by sub-classing the abstract “Simple Editors”. Sub-classing is also used to ensure the registration of the right kinds of “Simple Editors” for the new types. This is described in detail in Appendix C.7.
- **Customising by parameterisation.** Some of the classes in the framework can be parameterised with objects that influence their capabilities and behaviour at run-time. These parameters specify which meta-model, pretty printer and “GUIFactory” the framework should use. This is described in greater detail in Appendix C.8.

6. Documenting the Framework

To make good and efficient use of any reusable asset, the reuser must have access to comprehensive and accurate information about the asset.

In the search framework case, this was provided by four sources:

- Usage documentation, in form of a usage pattern organised document, based on an article by Ralph Johnson [Johnson 92] on how to document frameworks.
- Overview documentation, based on a rudimentary asset description template created as part of the COT project case 3.
- Excerpts from two books on patterns, [Gamma et al. 95] and [Buschmann et al. 96], containing all the patterns used in the framework.
- Verbal introduction, given as a three hour seminar which included a lecture on the basic ideas of the framework following the usage documentation, an outline of envisioned tasks facing the EWare developer as identified by the framework developer, and a question/answer session.

6.1 Usage Documentation

The usage documentation was written according to the guidelines in [Johnson 92]. These guidelines basically outline a hierarchical introduction of how to use the asset: The first section is an overview of principles and then references are given to individual sections: “If you need to do X, read section A, if you need to do Y, read section B, etc.” The documentation process facilitated “a check” of the new design and functionality as the IRIS/MFS developer was forced to consider how to solve the tasks described in the documentation in order to supply sample code for it.

The EWare developer used the usage documentation in the beginning mainly to get an overview of the architecture of the component. He soon missed the API reference.

This way to document a framework has so far proved beneficial as the IRIS/MFS developer had to check his implementation of the new design and the EWare developer stated that it provided him with an overview and a starting point for his design and development.

6.2 Overview Documentation

The overview documentation was based on an asset description template that had been written in the COT project case 3. The main purpose of the template was to serve as an overview of the asset and a checklist for potential reusers. It is envisioned that an overview document is written for each reusable asset in a reuse library and these overview documents serve as a browsable index of the library.

Thus, the focus of the template is to 1) shortly describe the existence of the asset and what it does (the overview) and 2) serve as a checklist so potential reusers can verify that they indeed can use the asset (constraints on programming language, compiler versions, dependencies, etc.). It is, however, only the technical properties of the asset that are covered in the template.

As it was already decided that the EWare project should use the asset, the overview document was not prepared until after the EWare developer had begun his work, and it was not used by him.

This leads to a natural conclusion, namely that overview documentation is merely relevant in the situation where potential reusers need to find an asset; not after it has been found.

6.3 Introduction Seminar

The first step in EWare's usage of the framework was an introduction meeting, where the framework developer described the framework and how to use it. Throughout the meeting the EWare developers could ask questions if they wanted anything clarified.

Prior to the meeting, the EWare developers had received material on the patterns used in the framework but no material on the framework itself. This was mainly due to the fact that the documentation described above was not finished at the time.

The presentation proceeded as follows: First the framework was presented the same way it was documented, i.e., an overall description of the framework, how to make a simple instantiation, and how to perform a number of tasks that EWare would have to perform. The presentation was concluded by a short overview of how the IRIS/MFS developer would fulfil the EWare requirements as specified in their requirement specification.

7. The EWare Fitting Process

This section describes the EWare fitting process, i.e., design, implementation, and fine-tuning.

7.1 Initial Fitting

A week after the introduction meeting, the EWare team began working on their solution. Before the design began, however, a feasibility study was undertaken: A simple example program using the framework was constructed in the EWare development environment.

The EWare platform is a Sun Sparc workstation running Solaris 5.5.1 using the SparcWorks compiler version 4.0.1. In section 8.1 the time spent by the IRIS/MFS and EWare developer is divided onto different tasks. As the EWare compiler is a fairly old C++ compiler it did not support STL, which is used by the framework. Therefore a public domain STL implementation had to be located. The compiler, however, could not compile this public domain library, and it was necessary to get a patch for the compiler from the vendor.

When it was tried to compile the test example, a number of trivial compilation errors were encountered because the SparcWorks compiler is stricter than the EGCS³ compiler used by IRIS/MFS. Flaws like these are fairly easy to correct but unfortunate in a situation where several projects already use the framework: The managerial costs of updating, releasing, updating clients, etc. can easily overshadow the cost of the actual code change. It must be avoided by exercising good coding practices during development.

Furthermore, the EWare project uses a different “Make” program and therefore it was necessary to write new makefiles for the framework. All in all 79 man-hours were spent correcting these problems. The vast majority of this could have been avoided if the same basic development environment had been used, i.e., the same compiler, linker, make program, and directory structure as the makefiles assume a certain structure.

When the example program compiled, the EWare developer experimented with the program and made a simple proof-of-concept prototype, where the example application was integrated in EWare. After having done this successfully the design was initiated. Due to other tasks within the team, only one of the two developers worked on the design, but did it almost fulltime. The product of the design phase was a design document of 25 pages. The design document described how the framework would be instantiated in EWare, i.e. which node types and “Simple editors” are needed for EWare, and which application specific code that was needed in order to operate on the output of the framework, i.e. the tree representing the expression. This application specific code mainly dealt with the generation of SQL based on the tree.

The EWare project manager demanded that his developer stated explicitly in his design which parts of EWare specific code mentioned above, i.e. the node types, “Simple

³ EGCS is a public domain extension of the public domain GCC compiler. EGCS is supported by Cygnus.

Editors” and SQL generation code, that were potentially reusable. For the potentially reusable EWare code a number of requirements were listed a) which libraries, other than the framework, the potentially reusable EWare code would depend upon at compile and link time, and b) several quality requirements for the code, e.g. that header files must be self-contained. The argument for this decision was that other framework reusers might be interested in reusing the EWare code and thus it should be clear what it depends upon.

7.2 Design Changes Resulting from Fitting Process

A number of desirable changes were of course identified during the EWare implementation. These changes are described below.

- **Major changes.** Major refers to the amount of resources needed to make the change.
 - *Design conflicted with C++.* The way the pretty printer was implemented was not legal C++ code according to [Ellis and Stroustrup 94]. The exact nature of the problem and the solution are described in Appendices C.5 and D.1.
 - *Redesign of “Simple Editors”.* As a few of the predefined “Simple Editors” conflicted with the EWare GUI style guide⁴, the EWare developer had to develop new editors following their style guide for GUI components. This is described in Appendix D.19.
- **Minor changes.** Most relating to customisation and adaptability. The changes are described in detail in Appendices C. and D.

7.3 Fine-tuning EWare Implementation

The EWare specific “Simple Editors” were coded directly in Motif. The implementation went smoothly and the implemented functionality worked. However, the quality was not up to the desired level. There was undesirable GUI behaviour such as resizing problems and furthermore the SQL code generated from the search expression tree was too slow when it was executed. The problems with user interface behaviour were due to two factors. One factor was too immature UNIX resource files in the component. The other was insufficient knowledge about how to use the framework when “Simple Editors” are developed in pure Motif in contrast to using the XMTMFC asset, which is the approach generally recommended for developing project specific GUI component that are used when instantiating the framework.

7.4 Documentation Update

After the implemented solution had been delivered to the EWare customer, the knowledge gained by the EWare developer was integrated in the usage documentation, which consisted of patterns as suggested by [Johnson 92]. The EWare developer had done two new things compared with the existing patterns: he had implemented the

⁴ The SSE understanding of “style guide” is that it describes the design style (colours, fonts, etc.), layout structure (functionality and placement of reoccurring GUI elements, e.g. “Help” buttons), and the general use of graphical effects.

EWare specific “Simple Editors” in pure Motif instead of using the XMTMFC asset and he had generated SQL from the expression tree. His experiences while doing so were captured in two new usage patterns. He furthermore had suggestions for improvement of the other usage patterns, which were incorporated.

7.5 Framework Usage in IRIS/MFS and EWare

In IRIS/MFS the framework is used to construct a component with a very simple interface consisting of a number of functions that all open the filter component, but have different signatures. This is possible as the same meta-model is used throughout IRIS/MFS.

In EWare a similar approach is used, but the interface to the component is not quite so simple as a different meta-model is used in each of the places from the component is activated. The different meta-models are specified using configuration files.

8. Data Acquisition and Analysis

Data was collected during the maturing, retrofitting, and EWare fitting process in a number of ways. An effort was made to collect high quality quantitative data, but this turned out to be difficult. For instance, it would have been interesting to assess the percentage of time spent on the actual harvesting, maturing, and retrofitting in IRIS/MFS. As these three activities are highly intertwined, and we claim that they cannot be separated in practice, it is unrealistic to get accurate data without burdening the developer with so many data-logging activities that it will influence the developing process negatively.

Thus, the data is primarily taken from qualitative sources. These sources are:

- **Development diary.** The IRIS/MFS and EWare developers kept a diary throughout the process. The diary describes problems and experiences of the developers.
- **Logging staff hour.** At SSE all spent staff hours are registered in a central system.
- **Evaluation interviews.** At regular intervals, about once a week, the EWare developer was interviewed, where he described problems, progress, etc. The interview also involved discussion of technical issues and their solution. After the EWare fitting process was concluded, a final interview was made where a preliminary version of the classification table (see Section 8.2) was discussed.

8.1 Staff-hours Log

Special tasks were made so that the developers could log their effort on these tasks for quantitative tracking.

The IRIS/MFS effort for the development of the original search component, whereof only 50% of the code was reused:

IRIS/MFS tasks	Time spent/man-hours
Design	255,5
Implementation	911,0
Writing test specification, testing and corrections	106,5
Review of documents	8,0
Updating manuals	61,0
Total	1342,0

The tasks for the EWare developer were used from the design onwards.

EWare tasks	Time spent/man-hours
Design	42,5
Implementation	124,5
Writing test specification, test and corrections	25,0
Review of documents	4,0
Updating manuals	3,0
Total	199,0

The harvesting, maturing, documentation, and technical problems were all registered on a separate account.

COT tasks	Time spent/man-hours
Technical problems	79,0
Harvesting + maturing	189,5
Documenting	73,5
Introduce EWare to patterns and framework	6,0
Total	348,0

8.1.1 Cost Avoidance Analysis for EWare Reuse

In [Poulin 97] reuse cost avoidance (RCA) is defined as:

$$RCA = \text{Development Cost Avoidance (DCA)} + \text{Service Cost Avoidance (SCA)}$$

The EWare project manager estimated, that a re-implementation of the EWare search facility would require 260 man-hours of effort. This estimate included analysis, design, implementation, writing test specification and documentation, testing and reviews, i.e., no reuse what so ever from IRIS/MFS. The effort used in IRIS/MFS to analyse, design implement, test and document the search component was 1342 man-hours for about twice the number of lines of codes present in the search framework.

As mentioned in Section 5. only the parts of the component needed by EWare were harvested and matured. We have tried to calculate the resources spent on developing those parts based on data from the SSE time registration database. In the database effort is registered on tasks, e.g. “Update manuals” or “Implement Simple Editors”. We have tried to find all tasks pertaining to the design, implementation, etc. of the code that has been reused and the effort amounted to 850 man-hours. Compared to these 850 hours the EWare estimate of 260 man-hours may be on the small side.

The estimate of 260 hours results in the following EWare DCA:

$$\begin{aligned} \text{EWare DCA} &= \text{Estimate for effort needed} - \text{Effort spent when reusing} \\ &= (260 - 199) \text{ man - hours} \\ &= 61 \text{ man - hours} \end{aligned}$$

However this saving was less then the time spent on the technical problems. The SCA cannot be estimated as there currently is no metric program at SSE measuring information about observations and error reports detailed enough for making an estimate of the saved EWare effort. Furthermore the EWare solution has not been operative at the customer site for very long and therefore any data that can be gathered about observations and error report would be to unreliable to base any conclusions upon.

The amount of resources spent on harvesting, maturing, and documenting the framework was 269 man-hours, which has to be financed by the reusing projects. These expenses include the resource used when retrofitting the framework in IRIS/MFS. However, another expense that is not considered by this metric is the amount of resources have been spent on developing a reusable component instead of a component specific for IRIS/MFS originally. The exact amount of extra development resources is hard to estimate, but nonetheless the IRIS/MFS project manager estimated that design used 50% more resources and the implementation used 25% more resources than if a pure IRIS/MFS implementation had been made. He furthermore estimated that the rest of the activities had a negligible over-head. The design and implementation amounted to 255 and 525 hours out of the 850 hours.

[Poulin 97] states, as a rule of thumb, that the extra effort for developing for reuse is approximately 50%. If harvesting and maturing are included as “development” effort, then the extra effort for developing for is the 50% for design, the 25% for implementation and the harvesting and maturing effort.

$$\text{Extra effort for design} = \frac{50\% \times 255 \text{ man - hours}}{150\%} = 85 \text{ man - hours}$$

$$\text{Extra effort for implementation} = \frac{25\% \times 525 \text{ man - hours}}{125\%} = 105 \text{ man - hours}$$

This amount to the following percentage of the used resources:

$$\begin{aligned}\% \text{ Extra effort for reuse} &= \frac{\text{Extra IRIS/MFS effort} + \text{Harvesting} + \text{Maturing}}{\text{Total IRIS/MFS effort} + \text{Harvesting} + \text{Maturing}} \times 100\% \\ &= \frac{190 + 269}{850 + 269} \times 100\% \approx 41\%\end{aligned}$$

It should be stressed however that this is a very rough estimate with a high uncertainty factor.

8.2 Classification of EWare Problems

Our hypothesis is that involving potential reusers in the design of an asset will result in a more mature asset as most domain related problems have been taken care of up front. To evaluate this hypothesis we registered all the problems encountered by the EWare developer. Table 1 contains a list of the problems encountered during the fitting of the framework into the EWare system. The problems are listed with a coarse-grained classification and estimates of the relative size of the problem in terms of man-hours needed to resolve each problem. A detailed description of each of the problems can be found in Appendix D.

The legend for the table is as follows:

- **Classification** can be “Domain”, “Technical”, “Process”, or “Training”.
 - **Domain** indicates that the problem is related to flaws or lack of flexibility with respect to the domain that the framework is supposed to support, i.e. the construction of queries. These problems are the ones that should have been avoided if one assumes a “perfect” design of the framework.
 - **Technical** refers to problems that are related to the changed technical environment when reusing the framework in EWare.
 - **Process** refers to problems related to constraints on the process that has lead to the framework component, specifically that there was no time to develop all relevant documentation.
 - **Training** refers to cases where the EWare developer did not have knowledge of techniques and processes that were assumed familiar by the framework developer. These techniques and processes are prerequisites for utilising the framework optimally.
- **Change** column describes which project is responsible for correcting the problem and it can have the values
 - **Producer** denotes that the framework developer is responsible for fixing the problem.
 - **Consumer** denotes that the framework consumer (in this case the EWare project) is responsible.
 - **Both** means that both consumer and producer must be involved.
 - **Organisation** means that it is responsibility of the organisation as a whole to handle the source of the problem. The change responsibility classification is discussed in more detail in Appendix D.
- **Cost** column indicates the cost of handling the problem and can have the values “*”, “**”, “***”, “****”
 - * means that the change is inexpensive in terms of man-hours invested.

- ** denotes “somewhere in between”, that is, it is so costly that the problem must be solved for the asset to be easily reusable.
- *** means expensive to the extent that it could compromise the usability of the framework.
- † means that the problem could have been critical because the solution relied on support/products from an outside source, solutions that might not have been available and would therefore have stopped the reuse effort.
- ☰ means that this is “part of the contract” and unavoidable; in almost any case of reuse you gain a lot, but have to pay some price for the gain.

Problem	Classification	Change	Cost
STL compilation	Technical	Consumer	***†
Makefiles	Technical	Organisation	***
Cast-problem	Technical	Producer	**
Missing API documentation	Process	Producer	**
No STL library	Technical	Both	*†
Nameclash on include files	Technical	Both	*
Compiler compatibility	Technical	Both	*
Framework not self-contained	Technical	Producer	*
Library compatibility	Technical	Both?	*
Test program X resources	Technical	Producer	*
X Resources	Training	Consumer	*
Domain model mapping	Domain	Consumer	*☰
Meta attribute identification	Domain	Producer	*
Pretty print name	Domain	Producer	*
SQL Generation	Domain	Consumer	*
GUI Style Conformance	Technical	Organisation	*
XMTMFC	Training	Consumer	☰
STL Learning Curve	Training	Consumer	☰
Framework heavyweight	Technical	Consumer	☰

Table 1 Classification of problems during EWare integration

8.2.1 Analysis

Table 1 has been sorted according to the cost so the most costly problems are listed first. Looking at the classification column, the result of case study is striking: The five most critical and costly problems are classified as technical or process problems. Of the four problems classified as domain related problems, one is viewed as unavoidable (☰), and only two are the responsibility of the framework developer. All four were easy changes.

The costly technical problems stem from a number of different sources. The make file problem (use of different make programs in the two projects) is obvious and underlines the general recommendation that successful reuse relies on having a common development environment [Reuse '99 Notes]. Please note that “common development environment” should be interpreted in a broad sense as “the technical means to allow easy integration of software components in different contexts”. This is typically facilitated by a well-defined and adhered-to way of handling system builds (e.g. common make program and directory structure or the use of a binary component

technology), coding conventions, directory layout, naming conventions, GUI style guides, etc. The cast problem (illegal C++ code in the framework) was a problem that should have been caught by the IRIS/MFS compiler but somehow managed to slip through. The STL problems (STL library not supported by EWare compiler) highlight the necessity of the managers of the reusable assets to provide lists of compatible compilers. Resources must be allocated to help potential clients to verify whether their compiler/environment can take advantage of an asset or not already in the planning phase of a project. The name-clash problem (two header-files with identical names) is difficult to avoid without some kind of company-wide naming convention.

A number of inexpensive problems remain, these are either technical or related to training. However, they should not be neglected. The compatibility problems between what different compilers consider as valid C++ code, especially when templates are used extensively as it is the case in STL, must be handled up-front by the reusable asset developers. This could be through adherence to strict coding practices and test compatibility with components and tools in wide use within the company. The required skill of the reuse developers must be clearly stated as part of the assets; and references to literature, tutorials, or provision for help must be given.

8.3 Scope of Analysis

The processes and analysis presented within the present report has been carried out in a specific setting, that defines the scope of the analysis and thereby the scope of the conclusions presented in the next section.

First of all, the customer base for SSE necessitates development for a number of prevailing platforms, most notably flavours of UNIX, Windows NT and the Win95 generation. Therefore, cross-platform development is a central issue. Upcoming binary component technologies like CORBA, Java Beans, and (D)COM are thus interesting from SSE's point of view. Some of the problems experienced and analysed in this section may have been avoided if such technologies had been employed; and henceforth the conclusions presented may be of less importance in the future. This may be true for many of the technical problems listed in Table 1. However, we think introducing component technologies will introduce other technical problems, especially at the present time when the technologies are still rather immature. For instance, COM is only well-supported on the PC-platform while Microsoft products are not very supportive for CORBA, and Java Beans force all development into the Java platform.

Secondly, some of the problems are specific to C++, even though the language has been standardised [C++ ANSI] it still has undefined semantics in several situations. This leaves a lot of room for idiosyncrasies between the implementation from different compiler vendors and therefore compatibility problems. Still, it does not alter the result that one must be careful that the semantics of a piece of code may differ when different compilers/interpreters are used; the same problem may even occur on binary components, e.g. a Java Bean interpreted by different virtual machines.

8.4 Source of Errors

In Section 9. we state a number of conclusions and lessons learned. These conclusions and lessons learned could be invalid if there are sources of error, if other influences,

within the pilot project or in SSE as an organisation, could lead to the same result. In this subsection, we try to examine potential sources of errors and discuss them.

The working hypothesis is that involvement of potential reusers will eliminate many of the fitting problems related to the domain of the component. As the analysis above shows, few domain related problem were indeed found. However, a similar result would likely be found in the case where the original designers were already well-versed in the domain of the component, here searching and filtering. Then the input from the potential clients would most likely not add significantly to the component developers domain knowledge, and they would be able to design and implement a component with few domain problems even without consulting potential clients.

In the reported case, there were two developers from the IRIS/MFS project group. The primary developer had little experience both within the domain of searching and the IRIS/MFS application and development environment as he recently had been hired by SSE. The other IRIS/MFS participant was a project manager who had been a developer on IRIS/MFS for 5 years. The project manager had implemented the old, very simple and inflexible, IRIS/MFS search facility, which consisted of a window with a limited number of fixed attributed. Thus neither of the two could be considered as experienced within the domain. Thus, we conclude that this source of error is negligible in the presented case study.

9. Conclusions

Based on the analysis in the previous section leads us to a natural (and expected) conclusion on the working hypothesis of the pilot project:

The involvement of the EWare team in the domain analysis and design of the framework ensured that the framework satisfied all major requirements from both the EWare and IRIS/MFS projects.

The working hypothesis is validated through the EWare case study and leads to a general recommendation that potential clients of a component should participate in its requirement specification and design if possible.

However, any given software solution has both a domain- and technical aspect. The experience gained through the EWare case study leads to another important conclusion:

The technical problems when introducing the framework in the EWare project were underestimated and were significant. Especially, the lack of a common development environment is a major obstacle that has to be overcome before systematic reuse seems viable.

These overall conclusions are supplemented by a number of more detailed conclusions. Though the consequences of the detailed conclusions outlined below are perhaps less far-reaching than those above, they must be considered carefully in a given, concrete, context.

The detailed conclusions of the case study are:

1. The processes of harvesting, maturing, retrofitting, and fitting are best carried out as an iterative processes. Especially the harvesting and maturing processes are highly interleaved and cannot easily be separated.
2. Systematic reuse of graphical user interface components requires an organisation wide look-and-feel style guide which all projects must adhere to.
3. Software should be written with potential reuse in mind, by striking a balance between adhering to good software engineering practices and avoid generalisations that may never be used. In our case this would have reduced the amount of resources used during harvesting and maturing if the code had depended on IRIS/MFS code to a very limited extent and had complied with guidelines for code and directory structure, documentation, etc.
4. The EWare developer did not feel that the main benefit of using the framework was faster development, but that the framework provided a ready-to-use and sound architecture for implementing search. In our opinion the developer felt so because he had to write a fairly large amount of code for the framework to be operational, i.e., meta-model, new types, new “Simple Editors”, expand pretty printer, design and implement visitor that generates SQL.
5. It is important to have procedures for ensuring that projects using the component have not been affected by changes made during the fitting for other projects. Every time a change was made for the EWare project the IRIS/MFS project got the same changes. The problem can of course be alleviated by following a release schedule for assets, but all projects, that incorporate the new release, still have to perform their tests again.
6. Introducing reusable assets has an important and highly beneficial side-effect, namely spreading knowledge about new technical skills in the organisation, in our case design patterns, reuse, and general libraries (STL). This increased skill-level is often generally applicable knowledge, i.e., not language or environment dependent. The EWare developer especially valued learning design patterns as he felt that this knowledge would be useful in future projects.
7. Aside from the spreading of knowledge a concrete technical benefit for EWare was that their compiler was patched. They also got confirmed that their present compiler is outdated and they are now considering switching compiler and make environment.
8. Company benefits are increased focus on company-wide development environment and GUI style guide.

10. References

- [Booch et al. 99] Grady Booch, James Rumbaugh, Ivar Jacobson: “*The Unified Modeling Language User Guide*”, 1999, Addison-Wesley, ISBN 0-201-57168-4
- [Bosch et al. 99] Jan Bosch, Peter Molin, Michael Mattsson, PerOlof Bengtsson: “*Object-Oriented Frameworks – Problems & Experiences*” in [Fayad et al. 99]
- [Brooks 95] Frederick P. Brooks: “*The Mythical Man-Month*”, Anniversary Edition, 1995, Addison-Wesley, ISBN 0-201-83595-9
- [Brown et al. 98] William J. Brown, Raphael C. Malveau, Hays W. McCormick III, and Thomas J. Mowbray: “*AntiPatterns – Refactoring Software Architectures, and Projects in Crisis*”, 1998, Wiley Computer Publishing, ISBN 0-471-19713-0
- [Buschmann et al. 96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal: “*Pattern-Oriented Software Architecture – A System of Patterns*”, 1996, John Wiley & Sons, ISBN 0-471-95869-7
- [COT/3-20] Henrik Røn: “*Development of a Component for Searching and Filtering*”, 1999, COT Report COT/3-20, available at <http://www.cit.dk/COT/> under “Reports”
- [C++ ANSI] ISO/IEC 14882, “*Programming Languages – C++*”, 1998
- [Ellis and Stroustrup 94] Margaret A. Ellis and Bjarne Stroustrup: “*The Annotated C++ Reference Manual*”, 1994, Addison-Wesley, ISBN 0-201-51459-1
- [Fayad et al. 99] Mohammad Fayad, Douglas Schmidt, and Ralph Johnson (eds.): “*Building Application Frameworks: Object-Oriented Foundations of Framework Design*”, 1999, Wiley, ISBN
- [Gamma et al. 95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides: “*Design Patterns – Elements of Reusable Object-Oriented Software*”, 1995, Addison-Wesley, ISBN 0-201-63361-2
- [Jacobson et al. 97] Ivar Jacobson, Martin Griss, and Patrick Jonsson: “*Software Reuse – Architecture, Process and Organization for Business Success*”, Addison-Wesley, 1997, ISBN 0-201-92476-5

- [Johnson 92] Ralph Johnson: “*Documenting Frameworks using Patterns*”, Proceedings of the 1992 ACM OOPSLA Conference, pp. 63-76, 1992
- [Johnson and Foote 88] Ralph Johnson and Brian Foote: “*Designing Reusable Classes*”, Journal of Object-Oriented Programming, Vol. 1, No. 2, 1988
- [Karlsson 95] Even-André Karlsson (editor): “*Software Reuse – A Holistic Approach*”, 1995, Wiley, ISBN 0-471-95819-0
- [Lakos 96] John Lakos: “*Large-Scale C++ Software Design*”, 1996, Addison-Wesley, ISBN 0-201-63362-0
- [Poulin 97] Jeffrey S. Poulin: “*Measuring Software Reuse: Principles, Practices, and Economic Models*”, Addison-Wesley, 1997, ISBN 0-201-63413-9
- [Pree 95] Wolfgang Pree: “*Design Patterns for Object-Oriented Software Development*”, 1995, Addison-Wesley, ISBN 0-201-42294-8
- [Reuse '99 Notes] Henrik Bærbak Christensen: “Summary of REUSE'99 Conference”, 1999,
<http://www.daimi.au.dk/~hbc/conference/reuse99.html>

A. Abbreviations and Definitions

- DCADevelopment Cost Avoidance.
 EWare..... A SSE product for registering and managing electronic warfare information.
 GUI.....Graphical User Interface.
 IRIS/MFSIris Message Formatting System. Iris is the Greek goddess for the rainbow and messenger for the Gods.
 LOC.....Lines Of Code.
 RCA.....Reuse Cost Avoidance.
 SCAService Cost Avoidance.
 SQLStructured Query Language.
 SSE.....Systematic Software Engineering A/S.
 STL.....Standard Template Library. A container library that is part of the C++ standard.
 XMTMFC.....An internal SSE product that acts as a thin wrapper around Motif, so that GUI code can be compiled on both UNIX and Wintel machines. The XMTMFC classes are wrappers around Motif providing an MFC interface to Motif.

B. The Physical Properties of the Framework

The size of the framework in terms of classes. We distinguish between the classes that are the actual framework and classes that are supplied to make the job easier for potential reusers, i.e. predefined types and “Simple Editors” for these types.

Entity	# of classes
Entire original IRIS/MFS component	49
Essential framework	19
Predefined types, editors, etc.	10
IRIS/MFS component based on framework including IRIS/MFS specific types, editors, etc.	19

The original IRIS/MFS implementation of the search component had the following physical properties in lines of code⁵ (LOC).

Entity	# files	# .cpp files	# .h files	Size in LOC
Original IRIS/MFS component	39	19	20	10510
Framework	28	18	10	5593
IRIS/MFS component based on framework	37	16	21	5926

The IRIS/MFS component based on the framework also contains the code from the original IRIS/MFS component that was not harvested and matured. The amount of code (and files) in the framework and the framework based IRIS/MFS component combined

⁵ This includes comments and blank lines.

is larger than the original IRIS/MFS component for a number of reasons. Firstly, because of the split new IRIS/MFS component, which is based on the framework, needs to subclass quite a number of the framework classes. Secondly there has been introduced some new functionality in the framework. Thirdly comments have been added to some of the .h files in longer files. Fourthly a lot of restructuring and movement of code has been done, which resulted in more files.

The search framework depends on a number of other potential assets, which also have been harvested and matured.

Entity	# files	# .cpp files	# .h files	Size in LOC
Expression tree in IRIS/MFS before harvesting	2	1	1	1109
Expression tree asset	4	2	2	814
IRIS/MFS extensions to the expression tree	2	1	1	166
XMTMFC wrapper	22	18	4	6266
String formatting utility	2	1	1	104

The XMTMFC wrapper depended on a number of other Motif GUI widgets that also had to be included:

Widget	# files	Size in LOC
Treelist	3	2644
Combobox	4	3201
Tabs	8	3402
Multi column list	5	4237

Because of the harvesting and maturing a new abstract data-type representing a date had to be added.

Entity	# files	Size in LOC
Date	2	359

The total amount of code needed to use the search framework is:

Entity	# files	Size in LOC
Framework + all dependencies	78	26620

C. Detailed Description of Design Changes

This appendix contains detailed descriptions of the design changes made during the harvesting and maturing processes. They are listed in the same order as they are listed in Section 4. and 5. .

C.1 Ability to Register “Simple Editors”

The “GUIFactory” is basically an application of the “Prototype” pattern from [Gamma et al. 95]. The change to the “GUIFactory” itself was minimal. It amounted to making

the prototype manager [Gamma et al. 95, p. 121] dynamically extensible by adding a method for registering a “Simple Editor”. In the original IRIS/MFS implementation this registration was handled in the “GUIFactory’s” constructor.

C.2 Validation of Data Entered in a “Simple Editor”

The ability to customise validation was added by given the meta-attribute the responsibility for validating input. A method “Validate” was added to the “MetaAttribute” class. The method returns a boolean and the default implementation is to return true. A user of the framework can make special validation by sub-classing the “MetaAttribute” class and specialising the “Validate” function.

The chosen design imposes a number of restrictions:

- **Validation is a “batch” process.** The validation is not performed until either the “Add Before”, “Add After” or “Substitute” button in the GUI from Figure 2 is pressed. A more flexible approach could be validation after every key press or similar.
- **More return values.** There are only two possible return values of a validation: true or false. A more differentiated and flexible approach where error codes are returned instead might be preferable.

C.3 Loading and Saving in “Simple Editors”

In the original IRIS/MFS implementation the “Simple Editors” and the node that the “Simple Editor” worked on were tightly coupled. The new design to alleviate this problem was to separate “Load” and “Save” by applying the “Strategy” and “Builder” patterns respectively. Both patterns are from [Gamma et al. 95].

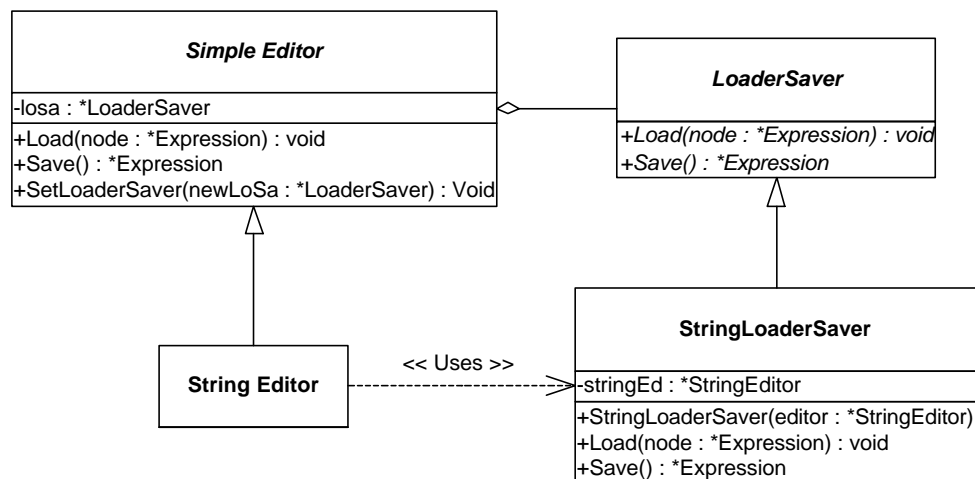


Figure 11 - Implementing Load and Save using the Strategy and Builder Patterns

In our opinion we have used the “Strategy” pattern for the “Load” operation, i.e., the “LoaderSaver” encapsulates the exact “algorithm” for displaying a node in a “Simple Editor”. The “Builder” pattern is used for the “Save” method, e.g. the “StringEditor” only knows that it’s “Save” operation results in an “Expression” node, it is the “Save” operation of the “StringLoaderSaver” that results in the “StringNode”. To quote

[Gamma et al. 95] on the intent of the “Builder” pattern: “Separate the construction of a complex object from its representation so that the same construction process can create different representation”. In our case this maps to “Save” (construction) and node types “representation”.

C.4 Reducing the Node Type Class Hierarchy

In the new design the extra classes are eliminated by adding an attribute for describing, which kind of expression it is. This is illustrated in Figure 12.

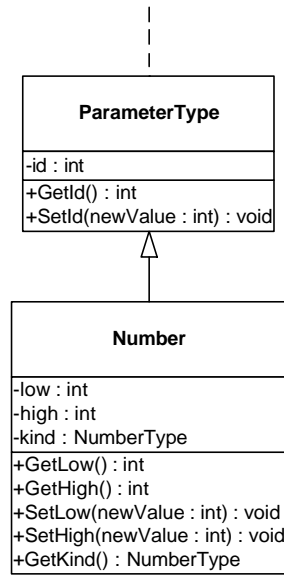


Figure 12 - The new structure after re-design

This reduced the complexity both in terms of the number of classes in the “Expression” hierarchy, but also in the number of “Visit” methods in the visitors. The number of classes was reduced from 21 to 11 and the same reduction in the number of “Visit” methods.

C.5 Redesign of the Pretty Printer

The “Pretty Printer” now aggregates a Visitor, which it uses to traverse the tree. The Visitor then calls back to the “Visit” methods of the “PrettyPrinter” class, which contain the implementation. This is illustrated in Figure 13 and Figure 14 below. First the general design of the “PrettyPrinter” aggregating a Visitor, followed by a sequence diagram depicting how this would work when pretty printing a “StringNode”.

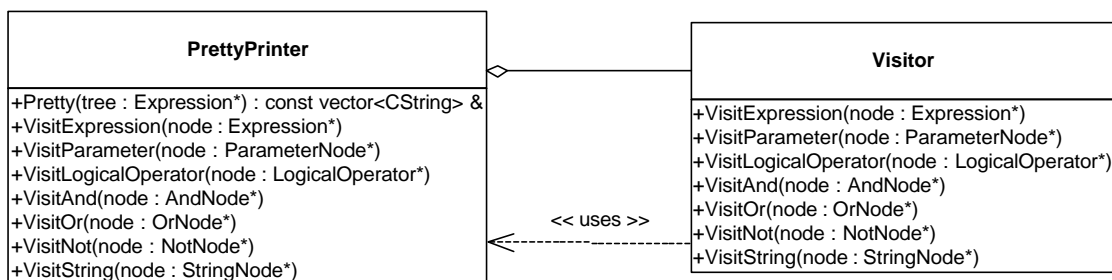


Figure 13 - The Pretty Printer aggregates a Visitor instead of sub-classing one

The calling sequence for visiting a “StringNode”:

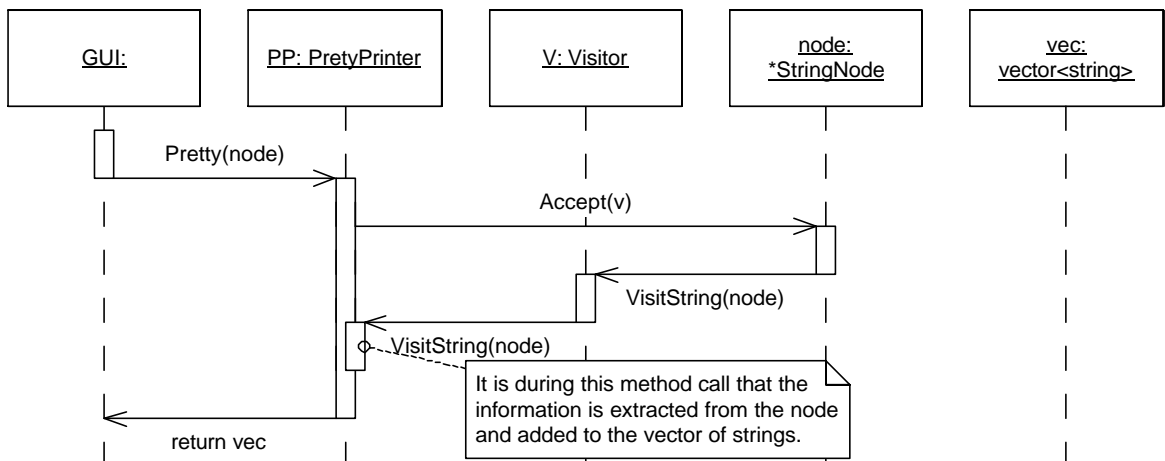


Figure 14 - The calling sequence in the new design

In EWare terms the class diagram for a “PrettyPrinter” would look like this:

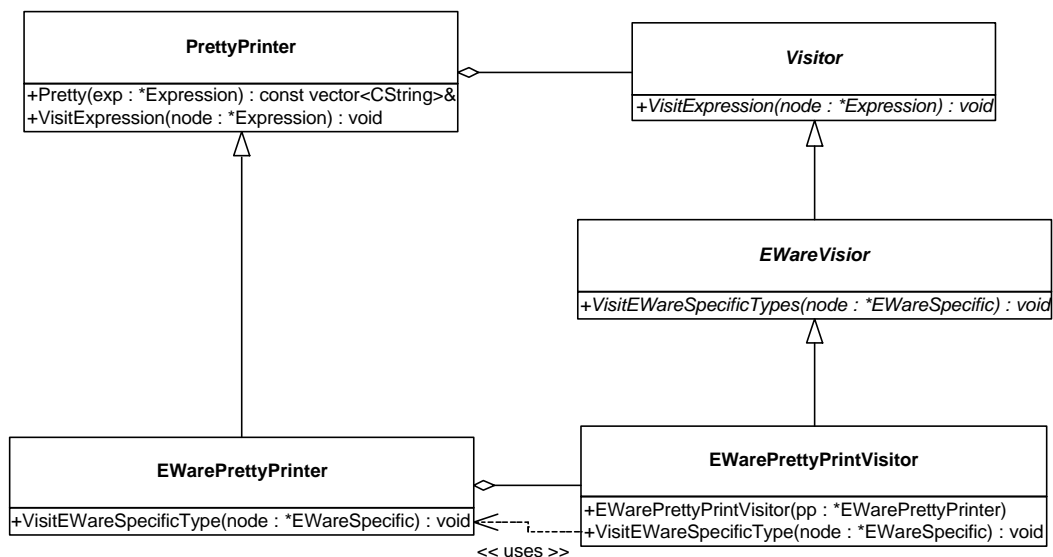


Figure 15 - Implementing a Pretty Printer in EWare

The “EWarePrettyPrintVisitor” calls back to the “EWarePrettyPrinter” that contains the implementation for the actual visitation of the node as shown in Figure 14.

The problem is partly due to C++, see Appendix D.1 for a detailed description of the problem. A language that supports multiple inheritance of interfaces, like Java, would use interfaces instead of abstract classes, and keep actual, shared, code in separate classes to be delegated to. This is basically the same solution as the one adopted, but the intention can be stated clearer through the language constructs. It may also be the case that the new `dynamic_cast` feature of C++ can make the casting without problems. This has not been studied in detail, however, the dynamic cast feature is a fairly recent addition to C++ and not all compilers support it correctly. Therefore the feature is

therefore not feasible for a reusable asset at the moment given the number of platforms the asset must be available on.

C.6 Default Values for Meta-attributes

Legal values were added to the meta-attributes by sub-classing the original meta-attribute.

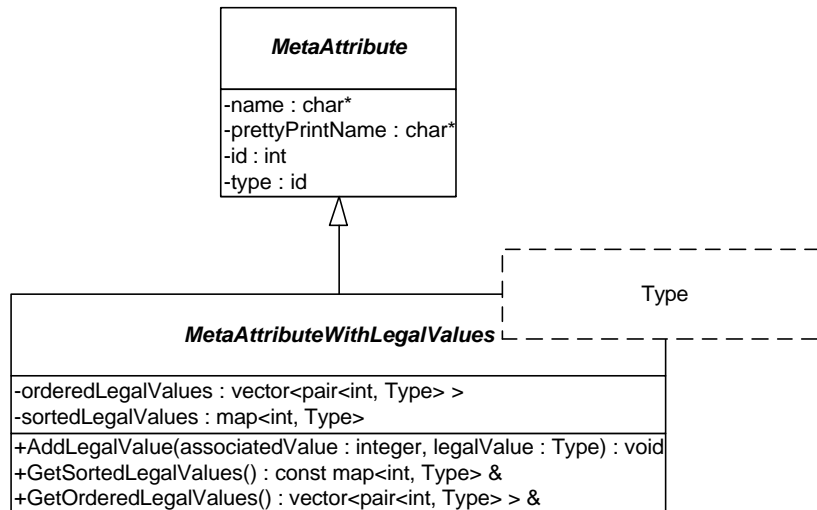


Figure 16 - Adding legal values to a meta-attribute

C.7 Customisation by Sub-classing

For each type of domain attribute there must be a “Simple Editor”. As described in Sections C.1 the editors are registered in the “GUIFactory”. Because all widgets in Motif must have a parent, the “Simple Editors” cannot be instantiated before their parent, in this case the “Advanced Page”, has been instantiated fully, i.e., it is not possible to do this in the constructor of the “Advanced Page”. Therefore the “GUIFactoryCreator” was introduced. The user of the framework writes the code that creates the “GUIFactory” and then creates and registers the needed “Simple Editors” in the “CreateGUIFactory” method that return a “GUIFactory” when called. This “GUIFactoryCreator” method is then called by the “Advanced Page” at an appropriate time.

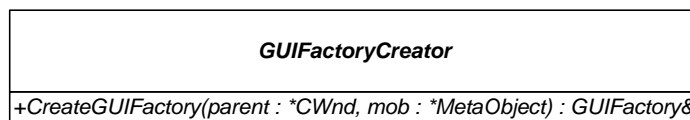


Figure 17 - The “GUIFactoryCreator” class

The user of the framework makes a subclass of “GUIFactoryCreator”, specialises the “CreateGUIFactory” method and in the method body instantiates and registers all the “Simple Editors” needed. This subclass is given to the “Advanced Page” as a parameter in the constructor, and is used later by the “Advanced Page” as illustrated in Figure 18.

An example of using a “GUIFactoryCreator” is shown in the Figure below.

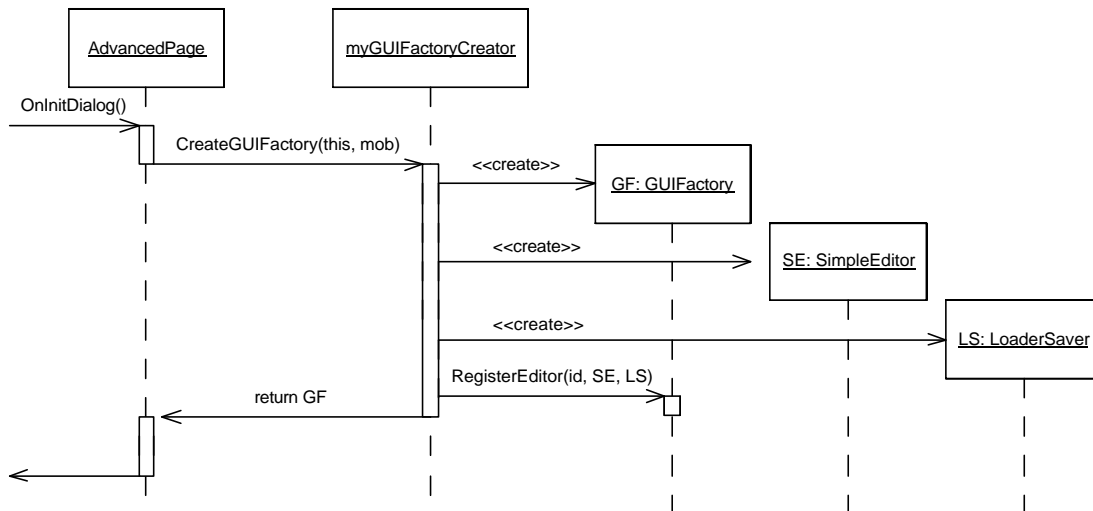


Figure 18 - The “Advanced Page” uses the “GUIFactoryCreator”

C.8 Customisation by Parameterisation

In the “Advanced Page” there are a number of parameters in the interface for opening the page:

```
AdvancedPage(MetaObject &mob, GUIFactoryCreator *gfc,
             PrettyPrinter *pretty=0,
             Controller *contr=0, bool readOnly=false);
```

The parameters are used for parameterisation in the following way:

- **MetaObject.** This parameter describes the data-model of the application domain and decouples the framework from the data-model of the application domain because the framework only uses the meta-model’s abstract types.
- **GUIFactoryCreator.** As described above it is the responsibility of the “GUIFactoryCreator” to create a “GUIFactory” containing the “Simple Editors” needed.
- **PrettyPrinter.** The pretty printer is used to print a textual representation of the filter tree, which is shown to the user in the GUI, marked \mathfrak{R} in Figure 2. If the user of the framework wants to add extra types to the framework, then the pretty printer must be expanded to handle these new types. Therefore it may be necessary for the user to be able to supply her/his own pretty printer instance.
- **Controller.** This is the object manipulating the tree based on the user’s interaction with the GUI. The users of the framework will usually not write their own Controller, but use the default one. The main reason for having such a parameter is when there are multiple filter pages like in IRIS/MFS where there are two filter pages, “Simple” and “Advanced”, and both pages use the same controller.
- **ReadOnly.** This parameter controls whether the page is read-only or not. This is useful when a search query can only be used, but not modified.

The “Simple Editors” can also be parameterised by giving a “LoaderSaver” object, i.e., the strategy for loading nodes into the editor and saving the contents of the editor into a node. This enables the use of the same editor for different tasks. The parameterisation is done when the “Simple Editor” is registered in the “GUIFactory” in the “GUIFactoryCreator” as it is registered along with a type-id and a “LoaderSaver” object matching that type-id.

D. Problems Encountered by EWare

This appendix contains a description of the problems encountered during EWare’s use of the search framework.

D.1 Cast-problem

Redesign and reimplement of the pretty printer. As the pretty printer code developed in IRIS/MFS contained some logistics on how to extract information from the tree for the pretty printing and it is highly desirable to reuse this code as it is not trivial. Therefore the original design was:

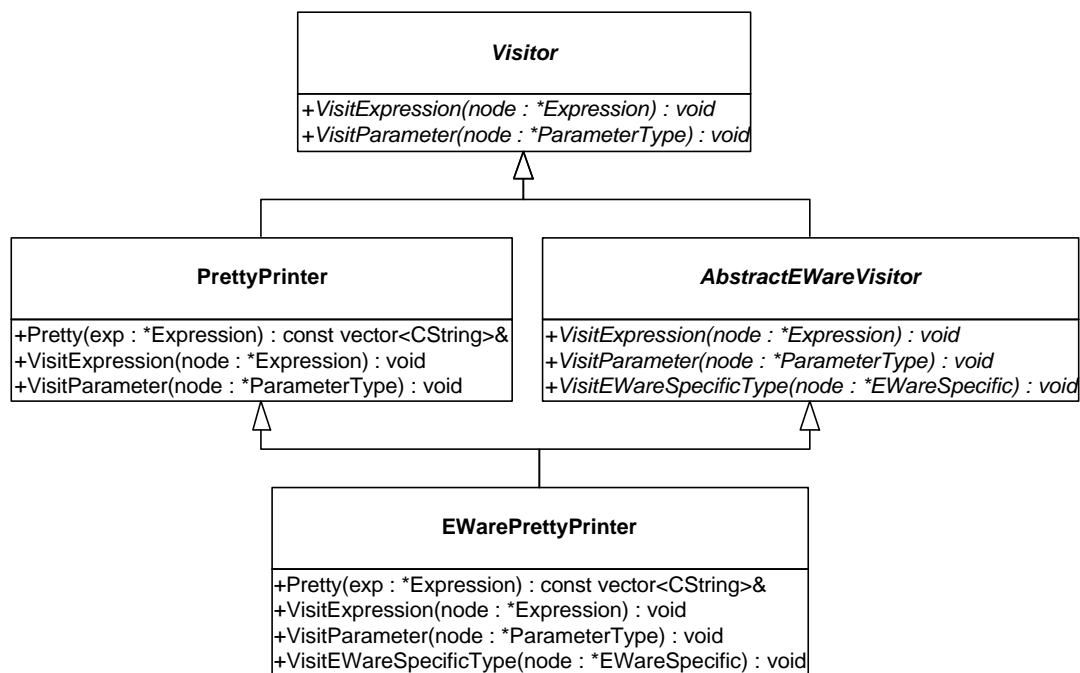


Figure 19 - The old design of the pretty printer

The above figure is an example from EWare. An interface for all EWare Visitors “AbstractEWareVisitor”, which contains visit methods for the EWare specific types, is defined. A pretty printer for EWare, “EWarePrettyPrinter”, inherits from “PrettyPrinter”, for the logistics and interface for the predefined types, and from “AbstractEWareVisitor”, for the interface for EWare specific types. The problem arises in the Accept method, which is part of the Visitor pattern from [Gamma et al. 95], in the class for the “EWareSpecific” node.

```
void EwareSpecific::Accept(Visitor *v)
{
```

```
(AbstractEWareVisitor*)v->VisitEWareSpecificType(this);  
}
```

This is not allowed in C++ according to the “Annotated C++ Reference Manual” [Ellis and Stroustrup 94, Section 10.5]. However the EGCS compiler accepted this, but the SparcWorks compiler did not. Therefore a redesign was needed. This redesign is described in Appendix C.5.

D.2 Makefiles

There is a tendency to view a component as merely a set of source code files. However, the build scripts are just as important and must be viewed as an integral part of a component. This size of the technical problems concerning build scripts facing the reusers of the framework was underestimated.

The make file structure for the framework, which adheres to the conventions of IRIS/MFS, had to be changed substantially to fit the file hierarchy, Make program, and compiler used in EWare. This was necessary as there is no standard development environment (here meaning: Agreed upon common structuring of makefiles, physical layout of files and directories, compiler conventions, etc.) agreed upon within SSE across departments and projects. This was one of the most time-consuming activities during the process.

D.3 No STL Library

The STL collection class library became part of the C++ standard in the draft from 1995, which was not ratified until 1998. Thus older releases of C++ development environments rarely have STL. This was also true in the case of the EWare environment. EWare is thus based on RogueWave’s collection class library. As the framework is STL based, a STL library implementation had to be located. A public domain STL library implementation that could function with the EWare compiler was located by searching the Internet.

However, the STL library was quite complexity to set-up. Obviously, a major investment had been made into the library as it was reported to work correctly on a very large number of compilers and platforms. The price was that the library was heavily parameterised, as a large number of #defines control the actual compilation of the library. Thus, a large effort must potentially be invested in understanding the parameters and their interrelationships in order to get STL operational. Fortunately the default parameter setting worked for the EWare compiler.

D.4 STL Compilation

The EWare compiler was unable to compile the STL library. Through the thorough a documentation and troubleshooting section of the STL library homepage, it was learned that a patch, released by SUN, was necessary to make the EWare compiler translate the library. The problem was that the compiler did not handle templates correctly.

D.5 Nameclash of Include Files

The STL library had a header file that had the same name as an include file within the framework. This was a problem during compilation as the STL header file was found first and it should have been the framework header file. This was fixed by moving the STL library header file directory to the end of the compilation search path.

D.6 Domain Model Mapping

The domain model used in EWare is very complicated compared the "flat" view provided by the framework. Therefore, a mapping between the EWare domain and the framework meta-model is necessary. As a simpler search facility was the user requirement to the EWare project, which demands a flat view model, this work had to be done anyway and is thus not related to the use of the framework.

D.7 Meta-attribute Identification

A meta-attribute in the meta-model is identified by an integer value, generated by the framework. The EWare project, however, had two, decoupled, locations within the design/code that needed to access the type definition of the same attribute. As the locations were decoupled in the design, there was no way to transfer the identity of the integer from one location to the other. Of course, the framework cannot help either, as it can only generate new integer values or acknowledge existing. The remedy is to introduce an indirection to the integer value; the EWare developer created a wrapper that held a mapping of string to integer values. The meta-attribute can thus be identified by an identical string-value (more readable anyway) in the two locations and the mapping make sure that either the related integer value is generated or just looked up.

As this problem can be expected to occur in many other contexts using the framework, it seems wise to include the mapping facility directly into the framework itself.

D.8 Pretty Print Name

As shown in Section 2. a meta-attribute is given a single name, like "Position". This name is used in two different contexts in the graphical user interface: In the picklist containing the list of attributes to edit, and in the search expression (panes denoted ← and → in Figure 2 on page 8). However, in EWare a long attribute name is wanted in the picklist and a shorter one in the search expression for clarity. The framework did not support this.

D.9 SQL Generation

As the underlying domain model of EWare is an entity-relation (E-R) model, not a flat model, it created a problem concerning SQL generation from the search-tree generated by the framework. The problem is that creation of the meta-model, information is lost. Specifically, the E-R model contains two different paths using table-joins between two table entries. This information, especially concerning the optimal of these two paths, is lost during the translation to the flat model. Thus, when the SQL is going to be

generated the information, which would allow an optimal SQL expression to be generated, has been lost. In practice, however, it turned out that the potential performance problems were not a problem.

D.10 Missing API Documentation

Due to lack of time an API documentation for the editor interface was not produced by the framework developer. Therefore the EWare developer really had no alternative but to reverse-engineer the exact sequence of API calls in order to write the code for “Simple Editors” for EWare specific types correctly. This was done by instrumenting the framework code (as the source was delivered) with output-statements in each method, and run the example code to infer a sequence diagram.

D.11 Compiler Compatibility

The EWare compiler was a lot stricter in terms of the C++ code that it would accept compared to the IRIS/MFS compiler. The framework code contained some spurious declarations like arrays of size 0, and some template functions assumed some default parameters which was therefore not written explicitly in the code; acceptable to the IRIS/MFS compiler but not to the EWare compiler.

D.12 Framework not Self-contained

One of the header files in the framework would not compile in the EWare environment as there was a missing include directive. This was not detected in the IRIS/MFS use of the framework as there was an indirect inclusion of the file from another source file. This problem was easily corrected, but nonetheless annoying.

D.13 Library Compatibility

As mentioned before, the EWare project generally uses RogueWave as collection class library, not STL. The EWare developer found that actually these two libraries do not function correctly when used within the same module. Therefore a strict separation had to be made. Fortunately only very low complexity data has to be moved between the design entities that uses STL and those that rely on RogueWave. However, in other projects this may be a large obstacle.

D.14 Test Program X-resources

The X resource file (graphical layout description) of the test program did not clearly identify the entities that were just part of the test program and those that was necessary for the framework.

D.15 X-resources

The EWare developer was used to hardcoding graphical layout from the source code, not to specifying this in X resource files. Thus a certain amount of training was required in order to make the layout for the filter dialog in EWare.

This problem is difficult to avoid in general, as many programming tasks today can be done in many different ways. However, the documentation should clarify what kind of training is assumed by the framework developers.

D.16 Framework Heavyweight

The EWare developer complained that the framework required quite a lot of steps before anything was up and running. This is one of the costs of flexibility and in general very difficult to avoid except for the most trivial reusable assets.

D.17 The IRIS/MFS Motif Wrapper

The framework relies on the XMTMFC library developed at SSE. Basically this is an adapter pattern that makes the graphical library X resemble the Win32 API quite closely (actually it is a bridge (strategy?) pattern making it possible to run applications on both platforms). This is of course great if you are a proficient Win32 programmer but do not know X. However, in the case of the EWare developer it was quite opposite: He was well versed in writing X code but has no experience with Win32. Therefore the adapter seems like a costly detour. Again, this seems like a cost that is difficult to avoid.

D.18 STL Learning Curve

STL is a large complex library. Therefore there is quite a steep learning curve before developers can use it proficiently. Again a cost that is difficult to avoid, but on the other hand introducing STL through a concrete reusable asset that teaches developers its usage through concrete examples may be a rather efficient way of teaching.

D.19 GUI Style Conformance

The look-and-feel of the framework is the same as that of IRIS/MFS. As the EWare style is somewhat different, the developer actually had to re-implement some of the supplied “Simple Editors” to conform to the EWare style. Thus, parts of the reusable asset were actually not reusable. This is classified as an organisational problem: A decision must be made either to strive for maximal reuse which require that all projects conform to a standard style guide, or less reuse potential. The first choice has the added benefit that developers are more easily reallocated across projects as style guidelines are known and the same for everybody.