Exporting MOOSE Models to Rational Rose UML

A **Computer Science Project**
of the faculty of Natural Science
at the University of Berne

submitted by

Daniel Schweizer

led by

Prof. Dr. Oscar Nierstrasz
Dr. Stéphane Ducasse

Software Composition Group
Institute of Computer Science and Applied Mathematics

April 2000

Further information about this work, the used tools and an *online* version of this document can be found at:
`http://www.iam.unibe.ch/~scg/`

The address of the author:

Daniel Schweizer
Effingerstrasse 9
CH-3011 Berne

or

Software Composition Group
University of Berne
Institute of Computer Science and Applied Mathematics
Neubrückstrasse 10
CH-3012 Berne
`dschwzr@iam.unibe.ch`
`http://www.iam.unibe.ch/~dschwzr/`

Abstract

In software re-engineering projects very often you have the source code of an application but you miss its programmer, the design and the documentation. In order to understand these systems you need reverse engineering tools.

UMLDesignExtractor is the prototype of a reverse engineering tool generating UML class diagrams from object-oriented code. *UMLDesignExtractor* is built on top of MOOSE and is written in SMALLTALK. For the graphical output it uses the API of Rational Rose, a professional UML modeler.

In the following chapters you find a survey on the involved technologies and architectures as well as a detailed description of the application *UMLDesignExtractor*, illustrated with four case studies.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

First of all I would like to introduce the environment of this project. I describe the given problem, and I give also a overall picture of the solution and the project's contributors.

1.1 Problem Description

In the past few years, the Software Composition Group (SCG) at the University of Berne researched a lot in software engineering - , and especially re-engineering techniques. As one practical result of the FAMOOS project[1], the SCG developed MOOSE - the implementation of a language independent meta model for object-oriented systems. Several integrated tools for problem detection and analysis helped MOOSE to grow to a powerful re-engineering framework.

In order to understand an object-oriented legacy system, it is necessary to capture its design and architecture. So far, MOOSE was lacking a tool for graphically illustrating its models. The classical reverse-engineering capabilities should be added to MOOSE, also but not only to document the architecture of MOOSE itself.

Since the Unified Modeling Language (UML) seems to be the future standard notation for object-oriented design, a graphical UML generator could fulfill all the requirements.

1.2 Requirements

- The application should be written in the same environment as MOOSE. It should be an independent and optional add-on.
- Generally there is a need for a platform and language independent solution. The idea was to use existing technologies.

[1] See chapter 2.2 for further information about the FAMOOS project

- The product should provide full UML compatibility.
- Additionally, there is a need for a flexible mechanism that enables filtering entities (classes, methods, properties, inheritance definitions). This enables the generation of diagrams containing relevant information only (In the sense of understanding the system in its external behaviour).
- The architecture of the application is considered to be strongly object-oriented, and thus flexible, dynamic, and extensible.

An informal evaluation demonstrated the accessibility of Rational Rose via its API through COM.

1.3 Environment

Here is an summary of the environment, in which the future application should be integrated:

- **VisualWorks 3.0**. The SMALLTALK environment from ObjectShare is installed as commercial version on the Sun Solaris workstations at the university, and as non-commercial version on the PCs at the university and at home.
- **MOOSE**. The implementation of the object meta model FAMIX is written in VisualWorks 3.0 SMALLTALK.
- **COM Connect**. The libraries from ObjectShare that enable an interaction with Active-X objects through Microsoft's Component Object Model (COM) are available for Windows family operating systems only.
- **Rational Rose 98**. This commercial and professional UML modeler is installed on the PC at the university and as non-commercial version on my PC at home.

Documentation and Application were written on the following platforms:

- **UNIX**. Sun Solaris (SCG)
- **PC**. Microsoft Windows NT 4.0 (SCG / at home)

All functionalities related to COM can only be run on platforms which allow the installation of VisualWorks 3.0 **and** the COM Connect libraries, such as:

- Microsoft Windows NT 4.0
- Microsoft Windows 95
- Microsoft Windows 98
- Microsoft Windows 2000

This documentation is written in **LaTeX**.

1.4 Solution

The result of this work is *UMLDesignExtractor*. A reverse engineering tool written in SMALLTALK. Figure 1.1 shows the general concept of *UMLDesignExtractor*.

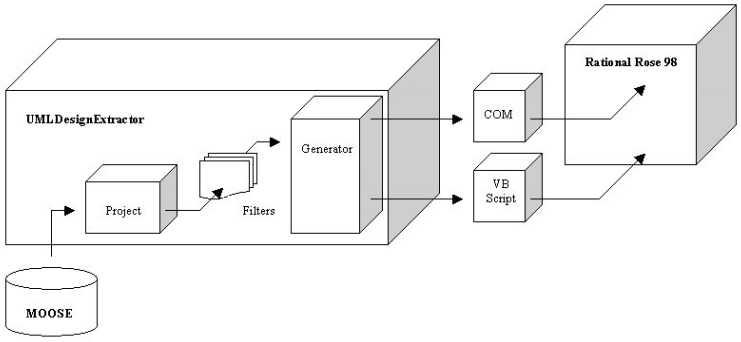

Figure 1.1: Conception of *UMLDesignExtractor*

MOOSE. The MOOSE meta model acts as base to our application.

Project. *UMLDesignExtractor* supports the definition of projects providing support for re-use and iteration when generating graphical representations of permanently changing models or changing focuses on the same model. These projects consist of information on the included models (or parts of models) and associated filters. Projects can be stored in project files, explained in Appendix A.

Filters. *UMLDesignExtractor* provides filter mechanisms to hide non-relevant entities. Possible filters are:

1. Attributes
2. Methods
3. Protected Methods
4. Private Entities
5. Inheritance Definitions

Any combination of the above or self defined filters is appropriate. Filters can either have global or local (class) scope. Potential filters are stored in a filter library. The file format of filter libraries is introduced in Appendix A.

Generator. A model can be generated and even stored without launching a Rose Application Window. Currently the following entities can be generated in a Rational Rose class diagram:

1. Classes
2. Attributes
3. Methods
4. Inheritance Definitions

Properties. *UMLDesignExtractor* provides several enhanced parameters for entity details, such as:

- Access control qualifiers (private, protected, public) for methods and attributes.
- Return types for methods
- Init values and types for attributes

COM. *UMLDesignExtractor* may communicate with the API of Rational Rose via COM. Rational Rose is launched as ActiveX object in a backgroud process.

VB Script. *UMLDesignExtractor* can alternatively generate a VisualBasic (VB) script and physically store them in a file. Theses scripts then can be imported and run from the Rational Rose Application itself.

Download an *online* version of this document, as well as *UMLDesignExtractor* as SMALLTALK code file (.st) on:

`http://www.iam.unibe.ch/~dschwzr/`

1.5 Contributors

This project was led by, and would not have been possible without, the continuous and spontaneous help of Prof. Dr. Oscar Nierstrasz, Dr. Stéphane Ducasse and Dr. Serge Demeyer. Spiritual father of this project is Dr. Stéphane Ducasse. He guided me through the crucial decisions, and powered me with plenty of ingenious ideas and a lot of know-how. Thanks also to rest of the FAMOOS team who prepared the base of my work, the SSUG and its protagonists to ameliorate my knowledge of Smalltalk, Cincom (former ObjectShare) and Rational Software Corporation to allow me to work at home on my PC thanks to their free non-commercial versions of the software. Special thanks to the whole SCG team to support me with all its experience and feed-backs.

Chapter 2

Context

Before I present my own work I would like to introduce all the involved techniques and architecture. I begin with a survey on software engineering, go ahead with MOOSE and the FAMOOS project and end up with an introduction to Microsoft's Component Object Model (COM) and a section dedicated to the Unified Modeling Language (UML).

2.1 Software Engineering

After repeating the classical view to software engineering I try to classify the terms *Re-engineering* and *Reverse Engineering*. The sections are followed by the history from procedural to object-oriented software development.

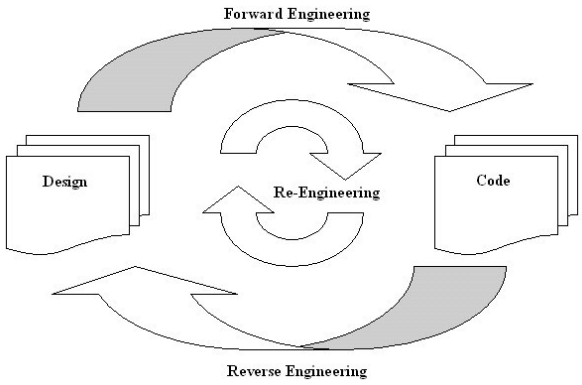

Figure 2.1: Software Engineering Lifecycle

2.1.1 Waterfall

The classical process of software development may be split in the following subprocesses:

1. Detection of requirements
2. Analysis of environment
3. High-level abstractions
4. Logical, implementation-independent design
5. Physical implementation of the system
6. Error testing
7. Documentation.

This traditional process is also called “forward engineering” and in depth described by *Waterfall*.

2.1.2 Re-engineering

In the real world whenever a piece of software is said to be complete its out of date. Changing environments and requirements ask for a continuous adaption. Software re-engineering includes:

1. Detection of changed environments
2. Analysis of concerned subsets of the software
3. Redesign and test case specification
4. Implementation of the design
5. Error and compatibility testing
6. Redocumentation.

Software re-engineering may involve refactoring and restructuring.

2.1.3 Reverse Engineering

In most re-engineering projects there is a need of reverse engineering, the process of analysing a subject system

1. to identify the system’s components and their interrelationships, and
2. to create representations of the system in another form or at a higher level of abstraction (e.g. metrics or visualizations).

It’s important to understand that without reverse engineering, the re-engineering of a system is unthinkable: changing a large and complex system without sufficient knowledge of its inner structure, will almost certainly trigger unwanted side effects which could make the system inoperable.

2.1.4 Object-Oriented Software Engineering

Object-oriented programming emphasizes the concept of an object. An object is an identified unit which has state and behavior. The state is stored in instance variables. The behavior of an object is effected through operations. When an operation of an object is called, the code to be executed is a procedure called a method. The class defines the instance variables, and the methods which are to be executed for each of the operations on instances of the class. The prime features that give object-oriented languages their power are called polymorphism, encapsulation, and inheritance.

Since procedural languages could not ease coping with permanently changing, and therefore in complexity growing systems, in the last two decades object-oriented languages have become the main force in software development. Object-oriented engineering includes the effort to support:

1. Higher-level programming languages (C++, Java, SMALLTALK)
2. Reusability of structure and behaviour
3. Tools for re- and reverse engineering
4. Round-trip engineering
5. Standardized notation to represent object architecture (UML)

Thus object-oriented systems are easier to re-engineer, which implies better maintainability, higher quality, lower costs and faster development.

2.2 MOOSE and the FAMOOS Project

2.2.1 The FAMOOS Project

The need for object-oriented re-engineering technology has been recognised by two of the leading European companies, namely Daimler-Benz and Nokia. Together with the University of Berne, Forschungszentrum Informatik, SEMA Spain and Take5 they started a research project named FAMOOS [1] to investigate tools and techniques for dealing with object-oriented legacy systems.

FAMOOS is a name referring to the ESPRIT Project 21975. FAMOOS is an acronym for **F**ramework-based **A**pproach for **M**astering **O**bject-**O**riented **S**oftware Evolution.

The "FAMOOS Object-Oriented Re-engineering Handbook"[2] is one of the main results of the FAMOOS project. It collects techniques and knowledge on the problem of software evolution with a special emphasis on object-oriented software. Most of the subject matter is not "new" in the sense that it represents new discoveries. Rather the handbook regroups much of the knowledge about redesign, metrics and heuristics into a single work that is focused on object-oriented reengineering.

2.2.2 FAMIX

FAMIX (**FAM**oos **I**nformation E**X**change model, see [DEME 99]).

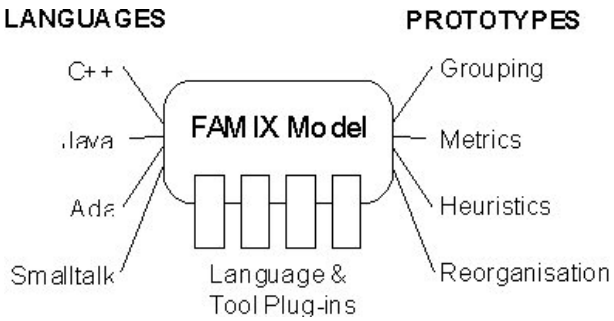

Figure 2.2: Conception of the FAMIX Model

The FAMIX model provides a language-independent representation of object-oriented source code and is used by the FAMOOS tool prototypes as a basis for exchanging information about object-oriented software systems.

[1] If you want to read more about the FAMOOS project and its results, I suggest to browse the web-sites offered by the respective project partners: http://dis.sema.es/projects/famoos/; http://www.iam.unibe.ch/∼famoos/; http://www.fzi.de/prost/

[2] [FAMO 99]

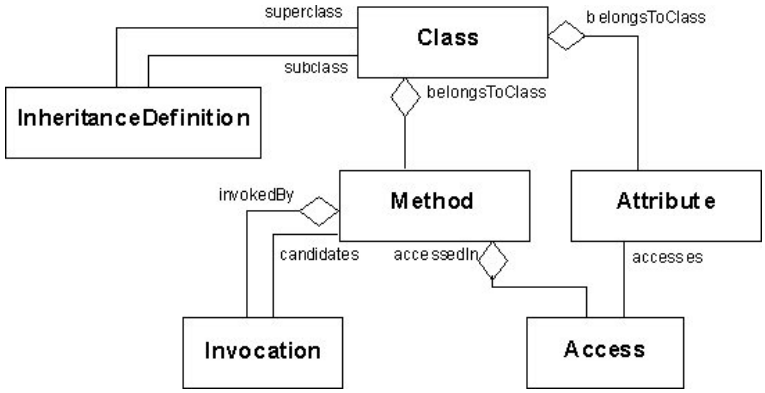

Figure 2.3: FAMIX Core Model

CDIF Transfer Format. CDIF was adopted as the basis for the information exchange of information in the FAMOOS exchange model. CDIF is an industrial standard for transferring models created with different tools. The main reasons for adopting CDIF are, that firstly it is an industry standard, and secondly it has a standard plain text encoding which tackles the requirements of convenient querying and human readability. Next to that the CDIF framework supports the extensibility needed to define the model and language plug-ins. More information concerning the CDIF standard can be found at http://www.eigroup.org/cdif/index.html.

A possible alternative for CDIF is XMI. However, XMI was considered too premature at the beginning of the FAMOOS project in 1996. Still, XMI is considered as a promising way to exchange FAMIX-based information.

Why not UML? The unified Modeling Language (UML) is rapidly becoming the standard modeling language for object-oriented software, even in industry. So, UML is a viable candidate for serving as the data model behind the exchange format. Nevertheless, UML does not include internal dependencies such as method invocations and variable accesses. Those dependencies are necessary in the problem detection and reorganisation phases of the re-engineering life cycle. Thus, choosing UML would violate the requirement of being a sufficient basis for re-engineering operations.

However, the terminology in FAMIX relies heavily on UML in the terminology and naming conventions applied in the model to become independent of the implementation language. For example, FAMIX has attributes instead of members (C++) or instance variables (SMALLTALK) and classes instead of types (Ada).

Why not CORBA/IDL? CORBA is receiving widespread attention as interoperability standard between different object-oriented implementation languages. CORBA's Interface Description Language (IDL) is used to specify the external interface of a software component and there are tools that extract IDL from source code. As such, CORBA/IDL is a viable candidate to serve as exchange format.

However, CORBA/IDL only describes the interface of a software component, and, like UML, not the internal dependencies such as method invocations and variable accesses. Thus, also CORBA/IDL would violate the requirement of being a sufficient basis for reengineering operations.

2.2.3 MOOSE

Mastering **O**bject-**O**riented **S**oftware **E**volution (MOOSE) is the implementation of FAMIX, written in SMALLTALK.

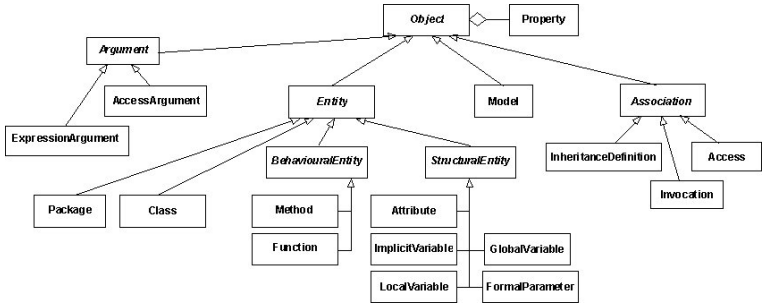

Figure 2.4: MOOSE Complete Model

The architecture of MOOSE, as an environment to aid in the program understanding and problem detection, corresponds to the basic re-engineering tool architecture: an information base is generated using parsers and semantic analysers and this information base is used to extract new views of the code using queries, graph viewers, etc. The MOOSE information base explicitly represents the concepts that are present in the code, e.g. classes, methods, instances, method invocation, etc. This explicit representation allows manipulating the code at a higher level than textual editing and allows to formulate hypotheses based on queries about the concepts present in the code.

MOOSE is an attempt to integrate third-party tools into a coherent whole. As an example, MOOSE imports information from the symbol tables maintained in the Sniff+2.2.1 and Concerto/Audit-CC++ environments and exports to public domain graph lay-out tools (XVCG). It allows also experimenting with a wide range of analysis tools perl-scripts, spreadsheets, query languages, prolog inference engines, graphical displayers to test their applicability in re-engineering.

2.3 COM - The Component Object Model

The following section contains information obtained from Microsoft's Developer Network Library Visual Studio 98. For more information about COM see their website:

`http://www.microsoft.com/`

To learn more about the use of COM with VisualWorks read *"COM Connect - User's Guide"*[3] or visit the VisualWorks Cookbook online:

`http://www.objectshare.com/doc/`

2.3.1 VisualWorks COM Connect

VisualWorks "COM Connect" provides support in SMALLTALK for COM and related fundamental technologies based on COM such as:

- Call-out and call-in of interface functions
- Distributed COM (DCOM)
- COM Object Server
- COM structured storage
- COM clipboard data transfer
- COM Automation (former OLE Automation)
- COM events

2.3.2 Microsoft's COM Architecture

COM is a system object model that enables modular system construction and reliable application integration. COM is increasingly used as the basis of new features in the Windows family of operating systems and is the foundation of a number of technologies. Microsoft is also working to make the COM architecture an open industry standard, with implementations on platforms other than Windows.

COM provides functions that enable you to build components that are distributed, and reusable. COM also enables cross-platform support, programming language independence and transparent remoting. COM objects (clients and servers) can be developed and written in different programming languages (Visual Basic, Visual C++, SMALLTALK).

Automation is built on top of COM to enable scripting tools and applications to manipulate objects that are exposed on Web pages or in other applications. Other technologies derived from COM include ActiveDirectory, OLE Messaging, Active Controls, Active Data Objects, ActiveX Scripting, Web Browsing.

[3] [OBJE 98a]

Objects and Interfaces. In COM, an object supports one or more interfaces. Each interface is a collection of functions that provide a related set of services to clients of the object. An interface is a collection of typed function signatures, representing a contract between a client and a server.

A number of standard interfaces are defined for common services and COM object implementors are encouraged to support existing interfaces where appropriate. COM object implementors can also define new interfaces as needed to publish the services of their server objects.
An interface is uniquely identified by an interface ID, or IID, which clients use to obtain an interface from a COM object. Interfaces are also referred to by a common name, which by convention is prefixed by the uppercase "I" to denote an interface. For example, the QueryInterface function in the standard IUnknown interface is referred to by the IUnknown::QueryInterface notation.

COM objects can only be manipulated by clients by referencing its available interfaces. Clients only obtain interfaces, never direct references to an object, so a COM object is entirely encapsulated. In COM only interfaces are real.

Publishing COM Objects. A COM object is published by registring information about its object class with COM. Published COM objects are identified by a class ID, commonly referred to as the CLSID. A COM object class can also be identified by its program ID, or PROGID, which is a short string name that identifies the application in the registry.

A published COM object class is supported by a class factory. A class factory is an object used by COM or the client to create new instances of the published object class. The IClassFactory interface contains a CreateInstance function, which allows clients to manufacture new objects.

Clients use COM objects by obtaining interfaces and invoking functions, then releasing interfaces when they are done using their services.

COM Applications. COM Applications are either clients or servers of COM objects, or both. COM server applications create and maintain objects. COM client applications are consumers of these objects. Many COM applications have both roles, in that they both use COM objects provided by other applications and implement COM objects themselves.

2.3.3 Distributed COM

DCOM extends the existing architecture by providing network communication capabilities from the existing model to enabled distributed object applications. DCOM is a direct competitor to the Common Object Request Broker Architecture (CORBA).

2.3.4 COM Automation

Automation (formerly called OLE Automation) is a technology that allows software packages to expose their unique features to scripting tools and other applications. Au-

ActiveX object	*Methods*	*Properties*
Application	Help	ActiveDocument
	Quit	Application
	Save	Caption
	Repeat	DefaultFilePath
	Undo	Documents
	...	Height
		...

Table 2.1: Microsoft Excel Application as ActiveX Object

ActiveX object	*Methods*	*Properties*
Document	Activate	Application
	Close	Author
	NewWindow	Comments
	Print	FullName
	PrintPreview	Keywords
	RevertToSaved	Name
	Save	Name
	SaveAs	Parent
	...	Path
		ReadOnly
		...

Table 2.2: Microsoft Excel Document as ActiveX Object

tomation uses COM, but can be implemented independently of other COM-based technologies, such as the OLE container architecture or ActiveX controls. The objects an application or programming tool exposes are called COM or ActiveX objects. Applications that access those objects are called COM or ActiveX clients. COM and ActiveX components are physical files (for example .exe and .dll files) that contain classes, which are definitions of objects. Type information describes the exposed objects, and can be used by COM and ActiveX components at either compile or runtime.

ActiveX object example. An ActiveX object is an instance of a class that exposes properties, methods, and events to ActiveX clients. For example, Microsoft Excel exposes many objects that you can use to create new applications and programming tools. See Table 2.2 and Table 2.2 for examples of ActiveX objects and a selection of their exposed methods and properties.

Accessing ActiveX objects. An Automation interface is a group of related functions that provide a service. All ActiveX objects must implement the IUnknown interface because it manages all of the other interfaces that are supported by the object. The IDispatch interface, which derives from the IUnknow interface, consists of functions that allow access to the methods and properties of ActiveX objects. The client must first create the object, and then query the object's IUnknown interface for a pointer to its IDispatch interface.

Although programmers might know objects, methods, and properties by name, IDispatch keeps track of them internally with a number called the dipatch identifier

(DISPID). Before an ActiveX client can access a property or method, it must have the DISPID that maps to the name of the member.

With the DISPID, a client can call the member IDispatch::Invoke to access the property or invoke the method, and then package the parameters for the property or method into one of the IDispatch::Invoke parameters. DISPIDs are available at runtime (late binding).

Basic Automation Invocations: (illustrated with examples from *UMLDesignExtractor*)

Creating an ActiveX Object. How to create a Rational Rose ActiveX object:

```
aDispatchDriver := COMDispatchDriver createObject: 'Rose.Application'.
```

Alternatively one can get an ActiveX object referring to its path or program id:

```
aDispatchDriver := COMDispatchDriver pathName: aFileName.
aDispatchDriver := COMDispatchDriver onActiveObject: aProgID.
```

Setting a Property of an ActiveX Object. I set an Attributes AccessControlQualifier to 'Private'.

```
aExportControlProperty setProperty: 'Name' value: 'Private'.
```

When there is more than one value to give as argument, you can provide the invocation with an array of values:

```
aDispatchDriver setProperty: name value: #(value1 value2).
```

Getting a Property of an ActiveX Object. I assign an Attribute's AccessControlQualifier to a local variable *aExportControlProperty*.

```
aExportControlProperty := aRoseAttribute getProperty: 'ExportControl'.
```

Calling a method of an ActiveX Object. The last invocation during a session is always:

```
aDispatchDriver invokeMethod: 'Quit'.
```

Methods may also be called parameterized with an array of values, either explicitly named or not:

```
arguments := Array with: 'Gary' with: #('Los Angeles' '33').
aDispatchDriver invokeMethod: 'SubmitAdress'
    withArguments: arguments.

namedArguments := Dictionary new
    at: 'Name' put: 'Gary';
    at: 'City' put: 'Los Angeles';
    yourself.
aDispatchDriver invokeMethod: 'SubmitAdress' withNamedArguments:
    namedArguments.
```

For a more detailed view on some code fragments from *UMLDesignExtractor* see Chapter 3 (Design).

2.3.5 COM Data Types

Automation Data Type	*Smalltalk Class*
VT_14	Integer
VT_UI1	Integer
VT_I2	Integer
VT_R4	Float
VT_R8	Double
VT_BOOL	Boolean
VT_ERROR	Integer
VT_CY	FixedPoint with a scale of 4
VT_DATE	Timestamp
VT_BSTR	String
VT_UNKNOWN	IUnknown
VT_DISPATCH	COMDispatchDriver
VT_ARRAY	Array (of Smalltalk objects)

Table 2.3: Matching COM Data Types to Smalltalk Classes

2.3.6 Links

COM Specification:

`http://msdn.microsoft.com/`

ActiveX Working Group:

`http://www.activex.org/`

Comparing DCOM and CORBA/IDL:

`http://www.bell-labs.com/~emerald/dcom_corba/Paper.html`

2.4 UML - The Unified Modeling Language

Although object-oriented anlaysis and design (OOA&D) was known since the late 1980s, there was a need of standardization to help demystify the process of software system modeling. A really recommendable introduction into UML and its history is "UML Distilled" written by Martin Fowler [FOWL 97].

2.4.1 The Initiators

There were a few key methodologists and their techniques between 1988 and 1992:

- Sally Shlaer and Steve Mellor. Wrote a pair of books [SHLA 89], [SHLA 91] on analysis and design; the material in these books evolved into their Recursive Design approach [SHLA 97].
- Peter Coad and Ed Yourdon. Wrote books that developed Coad's lightweight and prototype-oriented approach to methods. See [COAD 91a], [COAD 91b], [COAD 93], [COAD 95].
- The SMALLTALK community in Portland, Oregon. Came up with Responsability-Driven Design [WIRF 90] and Class-Responsability-Collaboration (CRC) cards [BECK 89].
- Jim Odell. Based his books (written with James Martin) on his long experience with business information systems and Information Engineering. The result was the most conceptual of these books [MART 94], [MART 96].

And of course there were the legendary Three Amigos:

- Grady Booch. Worked for Rational Software Corporation developing Ada systems. His books featured several examples and were famous for its great cartoons. Booch focused most on analysis and design methods [BOOC 94], [BOOC 95].
  Key question: How to abstract a real situation?
- Ivar Jacobson. Worked for Ericsson programming telephone switches. Jacobson was the first to introduce the concept of use cases. His field of interest was in notation and semantics [JACO 94], [JACO 95].
  Key question: How to represent an abstract situation?
- Jim Rumbaugh. Led a team at the research labs at General Electrics. Rumbaugh came out with a book about a method called OMT (Object Modeling Technique). His research was more process-oriented [RUMB 91], [RUMB 96].
  Key question: How to use a modeling language?

2.4.2 Unification

After a period of competition between the different notations that caused a lot of confusion several efforts for standardization were made with no success. Finally in 1994, Jim Rumbaugh left General Electrics to join Grady Booch at Rational Software, with

the intention of merging their methods. Grady and Jim proclaimed *the methods war is over - we won"*.

In 1995 they presented version 0.8 of the *Unified Method* and announced that Ivar Jacobson would join the Unified Team.

During 1996 Rational Rose and its Three Amigos worked on their method, under its new name: The Unified Modeling Language (UML). In the meanwhile the Object Management Group (OMG) formed a (from Rational Software) independent task force to do standardization in the methods area.

In 1997, UML was added to the list of OMG adopted technologies, and has become the industry standard for modeling objects and components.

The UML Revision Task Force (RTF) is responsible for generating minor revisions to the UML specification. The first UML RTF completed its revision work in June 1999, when it recommended its final draft of the UML 1.3 specification for adoption and submitted its final report. The members of the second UML RTF are now working on the next minor revision (UML 1.4).

The next major revision to UML, will be UML 2.0. The UML 2.0 Workgroup is responsible for drafting the UML 2.0 Request for Information (RFI) and the UML 2.0 Request for Proposals (RFP)[4].

2.4.3 Specification

UML in its current state, defines a notation and a meta-model. The **notation** is the graphical stuff you can see in models; it is the syntax of the modeling language. The **meta-model** is a diagram that defines the notation.

The syntax in UML is pretty intuitive and it's not the intention of this documentation to give a full enumeration of the elements in a diagram. However the most important components are:

1. Class (Object, Instance)
2. Attribute (Property)
3. Operation (Method)
4. Genaralization

Of course there are also some advanced concepts integrated in UML as:

- Stereotypes
- Multiple and Dynamic Classification
- Aggregation and Composition
- Derived Associations and Attributes
- Interfaces and Abstract Classes

[4]For the definitive UML reference visit: http://www.omg.org

- Immutability
- Visibility

Table 2.4 gives an view on the techniques and diagrams supported by UML.

Technique	*Purpose*
Activity Diagram	Shows behavior with control structure. Can show many objects over many uses, many objects in single use case, or implementation of method. Encourages parallel behavior.
Class Diagram	Shows static structure of concepts, types, and classes. Concepts show how users think about the world; types show interfaces of software components; classes show implementation of software components.
CRC Cards	Help getting the essence of class's purpose. Good for exploring how to implement use cases. Use if getting bogged down with details or if learning object approach to design.
Deployment Diagram	Shows physical layout of components on hardware nodes.
Design by Contract	Provides rigorous definition of operation's purpose and class's legal state. Encode these in class to enhance debugging.
Interaction Diagram	Shows how several objects collaborate in single use case.
Package Diagram	Offers useful bits of analysis, design, and coding techniques. Good examples to learn from; starting point for designs.
Refactoring	Helps in making changes to working program to improve structure. Use when code is getting in the way of good design.
State Diagram	Shows how single object behaves across many use cases.
Use Case	Elicits requirements from users in meaningful chunks. Construction planning is built around delivering some use cases in each iteration. Basis for system testing.

Table 2.4: UML Techniques and their Uses

The definitive specification guide on UML is: *"The UML Reference"* by the Three Amigos [RUMB 99].

2.4.4 Terminology

Still there is a big confusion about different terms meaning the same or people using the same term in a different sense. Table 2.5 shows the components of a standard UML class diagram and all its acronyms in comparison.

UML (verbose)	**Class**	**Association**	**Generalization**	**Aggregation**
UML (graphical)	□	□ – □	□ → □	□◇ → □
Booch	Class	Uses	Inherits	Containing
Coad	Class & Object	Instance Connection	Gen-Spec	Part-Whole
Jacobson	Object	Acquaintance Association	Inherits	Consists of
Odell	Object Type	Relationship	Subtype	Composition
Rumbaugh	Class	Association	Generalization	Aggregation
Shlaer / Mellor	Object	Relationship	Subtype	N/A

Table 2.5: UML Class Diagram Terminology

2.4.5 Rational Rose

Why Rational Rose? Rational Rose was chosen for the following reasons:

1. Its popularity (as market leader)
2. The availability of the API
3. The accessiblility of this API via the COM interface

Consider that there are many other UML modeling tools in industry like MagicDrawUML, COOL:Jex, GDPro, Visual Modeler (from Microsoft), Objecteering (with graet support for repositories), Together (round trip engineering), Argo/UML (public domain) etc. matching or even exceeding the requirements. Because of the modular architecture of *UMLDesignExtractor*, the necessary extensions for being able to communicate with another modeler, if ever desired, should be easy (assumed, that the modeler exposes its functionality as ActiveX object via COM).

Techique	*Rational Rose*
Activity Diagram	?
Class Diagram	Class Diagram
CRC Cards	?
Deployment Diagram	Deployment / Component Diagram
Interaction Diagrams (Sequence & Collaboration)	Interaction / Scenario Diagrams (Sequence & Collaboration)
Pachage Diagram	Component - & Class Package Diagram
Patterns (Repository)	?
State Diagram	State Diagram
Use Case	Use Case Diagram

Table 2.6: UML Techiques supported by Rational Rose

Rose models could be stored in a platform independent ASCII representation known as *"Petal"* (.ptl). The specification of petal is Rational Software Corporation proprietary and not publically available. Therefore I did not pay any further attention to Petal.

Rational Rose Extensibility. The concept of Rational Rose provides several ways to customize its capabilities. You can:

1. Customize menus
2. Automate manual functions with Rose Scripts
3. Execute functions from within another application by using the Rose Automation object (RoseApp)
4. Access classes, properties and methods right within your software development environment by including the Rose Extensibility Type Library in your environment
5. Activate add-ins using the add-in manager

The Rose Application is itself component-based, and is defined in the Rose Extensibility Interface (REI) Model. The REI Model is essentially a meta-model of a Rose model, exposing the packages, classes, properties and methods that define and control the Rose application and all of its functions. For a detailed description of the REI calls see the *"Rose Extensibility Reference Manual"*[5].

Important in our context are the two capabilities Scripting and Automation. *UMLDesignExtractor* supports both interaction types - that means that we can generate **Rose Scripts** and run them later on a destination machine, that is not connected to the source machine. On the other hand we are able to communicate directly to Rose within its **Automation** (COM) interface. If Rose is not installed on the same computer (but the two computers are connected) we can communicate via the DCOM interface.

Compatibility; UML vs. MOOSE. Although the terminology between the UML implemented in Rational Rose differ sometimes from the terminology used in the meta-model of MOOSE, the concept and the components as well as its roles in the two models match very well. One nonsignificant example of an exception: Rational Rose supports the C++ specific *"implementation visibility"* property for attributes and methods which is not known to MOOSE.

2.4.6 Classification

The availability of a standard modeling language will encourage more developers to model their software systems before building them. The benefits of doing so are well-known to the developer community. While before the key question was to build right the system today's focus is set on building the right system.

The need of a standardized modeling language is out of question. Nevertheless one should be aware that this mostly is not more than a common way of representation for exchanging object models or to gain a very general view on a system's classes and its relationships. UML defines a notation and its semantics but not the process of modeling. There is a book about the process of using UML on object-oriented projects: [JACO 99].

[5] [RATI 98a]

Chapter 3

Design

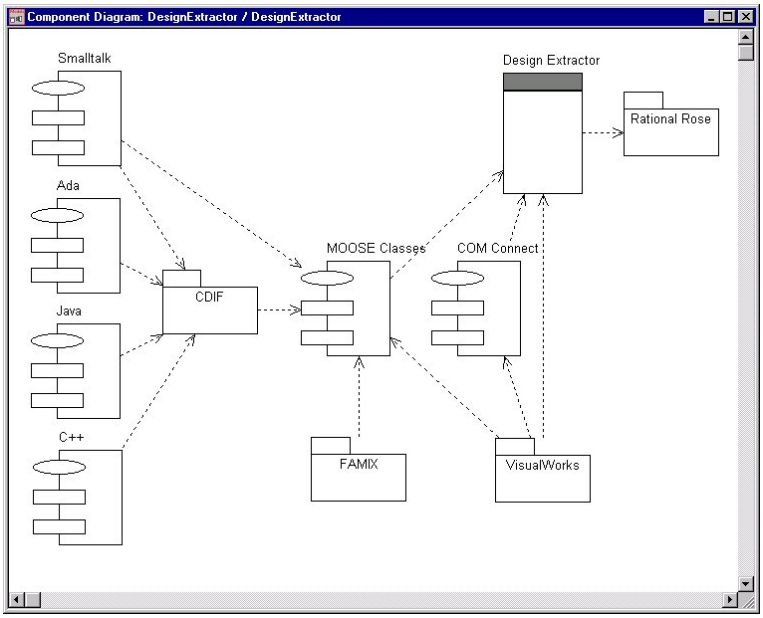

Figure 3.1: *UMLDesignExtractor* Framework

Figure 3.1 shows the overall deployment. Object-oriented source code, represented in the CDIF format, or directly taken out of the VisualWorks SMALLTALK system, is the input of *UMLDesignExtractor*. MOOSE, the implementation of the meta-model FAMIX, produces a SMALLTALK representation of this model, that then can be read by *UMLDesignExtractor*; thereby all desired entities of the model can be generated on a Rational Rose UML class diagram. Rational Rose is represented by a ActiveX object available to the system via COM.

3.1 Components and Classes

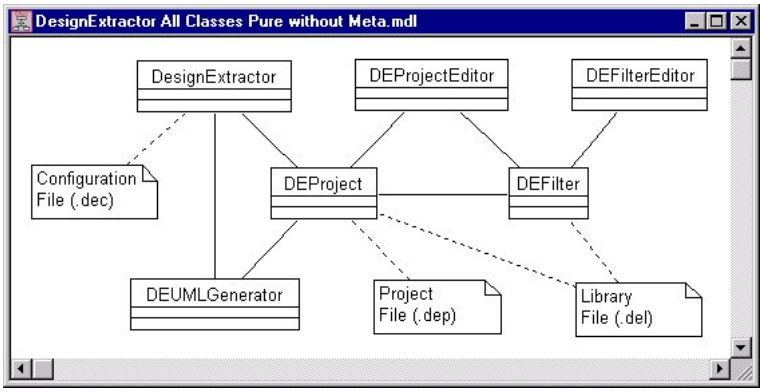

Figure 3.2: *UMLDesignExtractor* Classes

DesignExtractor. The core application class, implemented as Singleton[1]. Responsibility: This class acts as control board for all the provided functionality such as loading and editing a model, modifying and storing the application's configuration, editing a project and the filter library, starting the UML generator, quiting the application. DesignExtractor inherits from *ApplicationModel*.

DEFilter. A Filter is an object that disallows some entities to be generated. Responsibility: A filter contains a block, which, applied to a entity, is evaluated to true or false. DEFilter inherits from MSEAbstractRoot; (for more specific information about the filter mechanisms refer to the next section).

DEFilterEditor. An editor for filters. A filter has a name, a block, and a type. The type represents the specific entity type (class, method, attribute, entity) to which a filter is applicable.

DEProject. A project includes the configuration of filters and can be stored in a file (.dep). For a detailed description of the file formats see Appendix A.

DEProjectEditor. An editor for a project (for detailed information see tutorial in section Graphical User Interface)

DEUMLGenerator. This class effectively produces the UML model. Communicating with the API of Rational Rose via the COM interface, DEUMLGenerator acts as

[1]For more about the Singleton design pattern refer to [ALPE 98]

remote control for Rational Rose. To understand what exactly happens the following fragments of code, that generate all (not filtered) attributes of one class, might be helpful:

```
DEUMLGenerator >> prepareGeneration
    aDispatchDriver := COMDispatchDriver
        createObject: 'Rose.Application'
        serverName: destinationServer.
    roseModel := aDispatchDriver invokeMethod: 'NewModel'
    roseCategory := roseModel invokeMethod: 'RootCategory'.

DEUMLGenerator >> generateAttributesFrom: aMooseClass in: aRose-
Class
    aMooseClass attributesDo:
        [:attr |
        showAttribute := project filter: attr.
        showAttribute ifTrue:
            [self DEUMLGenerator >> generateAttribute: attr in: aRose-
Class]]

DEUMLGenerator >> generateAttribute: aMooseAttribute in: aClass
    arguments := Array
        with: (aMooseAttribute name)
        with: (self attrType: aMooseAttribute)
        with: (self attrInitValue: aMooseAttribute).
    aRoseAttribute := aClass
        invokeMethod: 'AddAttribute' withArguments: arguments.
    aExportControlProperty := aRoseAttribute
        getProperty: 'ExportControl'.
    aExportControlProperty setProperty: 'Name'
        value: (self attrAccessControlQualifier: aMooseAttribute).
    aRoseAttribute setProperty: 'Static'
        value: (self attrStatic: aMooseAttribute).
    aRoseAttribute setProperty: 'Derived'
        value: (self attrDerived: aMooseAttribute).

DEUMLGenerator >> finalizeGeneration
    roseClassDiagram := roseCategory
        invokeMethod: 'AddClassDiagram' with: 'DesignExtractor'.
    allClasses := self getAllClasses: roseCategory.
    numberOfClasses := self getNumberOfClasses: allClasses.
    1 to: numberOfClasses do:
        [:indexOfClass |
        aClass := self getClassAtPos: indexOfClass from: allClasses.
        self addClass: aClass to: aClassDiagram]
```

```
arguments := Array with: destinationModel with: true.
globalStream invokeMethod: 'SaveAs' withArguments: arguments.
```

3.2 Filter Mechanisms

What is a filter? There are plenty of filters that make sense:

1. hide all private entities
2. hide all public methods
3. hide all protected entities with a name like "my*"...

Finally there are many combinations that may be very interesting in some cases and totally unnecessary in other situations. Therefore "Filter Libraries" may be defined. One single filter is a piece of SMALLTALK code (a block like e.g. of the form):

```
[:each | each isPrivate]
```

With this implementation I achieve a maximum of flexibility and scalability. Any valid combination of SMALLTALK blocks can be a filter. The application then prevents the generation of the entities that match one of the selected filters in a so called "Filter Project". You can find the definition of the file formats ("Filter Libraries" and "Filter Project") in Appendix A.
Attention! Consider you want to define a composed filter that hides abstract classes like this:

```
(each isAbstract) & (each isClass)
```

You may run into problems because not every entity knows the message "isAbstract". So avoid these problems defining filters as follows:

```
(each isClass) and: [each isAbstract]
```

Where filters are applied? Filters are evaluated in one single place in the system. Take a look at the following code fragments to understand where exactly the filters are applied.

```
DEUMLGenerator >> model allClassesDo:
    [:aClass |
    aClass allMethodsDo: [:anEntity | filter: anEntity ]
    aClass allAttributesDo: [:anEntity | filter: anEntity ]]
model allInheritanceDefinitionsDo: [:anEntity | filter: anEntity ]
```

```
DEUMLGenerator >> filter: anEntity
    project allFiltersDo detect:
            [:aFilter | aFilter isTrueFor: anEntity]
        ifNone: [generate: anEntity].
```

3.3 Core Application

You can start *UMLDesignExtractor* with the following command in the transcript window of SMALLTALK:

DesignExtractor openWith: '[*Projectname*].dec'.

(Where '[*Projectname*]' is to be replaced with the name and path of an existing project).

For a more convenient interaction there is a Graphical User Interface (GUI), explained and illustrated with screenshots in the following section.

3.4 Graphical User Interface

In this section I describe the graphical user interface of *UMLDesignExtractor*. You can read the section like a step-by-step tutorial:

The main window (Figure 3.3) shows the complete configuration (as saved in [*Projectname*].dec). From here you start all other functions (Table 3.1):

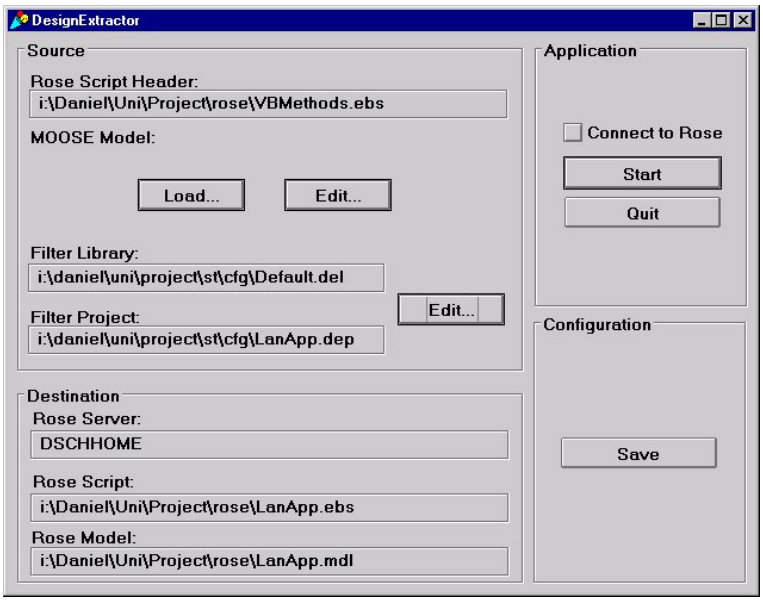

Figure 3.3: Screenshot of Main Window

The Project Editor (Figure 3.6) allows the definition of *"Filter Projects"*. (Table 3.2) shows its components.

Source:	
Button *"Load..."*	Loading a MOOSE model (reads classes or whole categories from the SMALLTALK image itself or imports a .cdif file). The Model Loader is shown in (Figure 3.4).
Button *"Edit..."*	Editing model, deselection of entities that you don't need (that are not relevant, therefore should not be generated). The Model Editor is shown in (Figure 3.5).
Inputfield *"Rose Script Header"*	Path and filename of a Rose Script Source File (.ebs). This VB script contains the definition of the methods to generate entities in a Rational Rose model. If I can not connect to Rose and communicate with its API, I still have the possibility to generate a script and run that later. The script first generated by *UMLDesignExtractor* contains a list of calls of methods of the *"Rose Script Header"*. So this header is automatically inserted at the very beginning of the final Rational Rose script.
Inputfield *"Filter Library"*	Path and filename of the project's Filter Library (.del).
Inputfield *"Filter Project"*	Path and filename of the project's specification called. The Filter Project is a file with extension (.dep).
Button *"Edit..."*	Editing the specified Filter Project and the specified Filter Library.
Destination:	
Inputfield *"Rose Server"*	This is the computername or IP address of the machine running Rational Rose.
Inputfield *"Rose Script"*	This is the target file (path and filename) to store the script, that alternatively to a direct communication to Rose, there could be imported and run.
Inputfield *"Rose Model"*	Path and filename under which the generated Rational Rose Model (.mdl) is stored.
Application:	
Checkbox *"Connect to Rose"*	Select whether you want communicate directly with Rational Rose, or just generate a script, that can be run later, importing it into Rational Rose.
Button *"Start"*	Starts the generation of the entities of the previously loaded model, filtered with the specified Filter Project.
Button *"Quit"*	Quit the application.
Configuration:	
Button *"Save"*	Stores the configuration (the content of all the above inputfields) to the Configuration File (.dec).

Table 3.1: Main Window Explained

The Filter Editor shown in Figure 3.7 provides editing a filter from the *"Filter Library"*:

The resulting model (.mdl) in Rational Rose might look like illustrated with one of the screenshots in the next chapter "Case Studies".

3.5 Summary

What I have done: *UMLDesignExtractor* is a prototype of a tool to extract the design of a system having only its source code. The implementation includes classes, metaclasses, its attributes and methods as well as the entities properties such as visibility

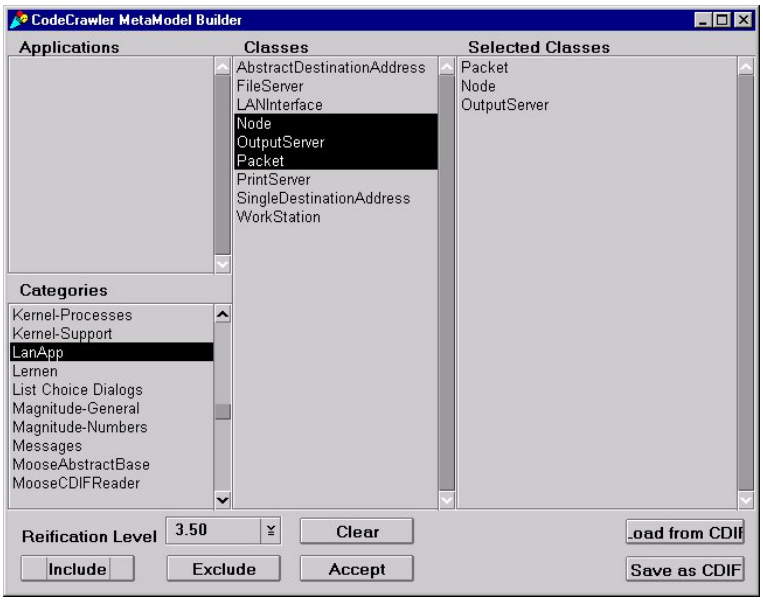

Figure 3.4: Screenshot of Model Loader (provided by MOOSE)

(private, protected, public). The only relation is the generalization (inheritance definition). The application extracts the static architecture (class diagram) only, and makes no statements about the behavior and sequence of interactions between the classes.

What I have not done: An advanced reverse engineering tool could support other nice features like: Showing selected attributes as related classes, more association (invocations), aggregation, other diagrams (activity, state), dynamic object structure and interaction (sequence and collaboration diagrams). DCOM was never tested out due to the circumstances that in the group there is one PC only.

One big disadvantage not to use Petal but CDIF as textual model representation is that we loose layout information. Especially when generating models in a iterative way it is very annoying everytime having to rearrange the classes on the class diagrams in Rational Rose.

The introduced functionality to filter a model, in fact, should in a next step be integrated into the kernel of MOOSE. The Model Editor would greatly profit from this extension. After this refactoring also other reengineering tools could easily be applied to any appropriate model subset.

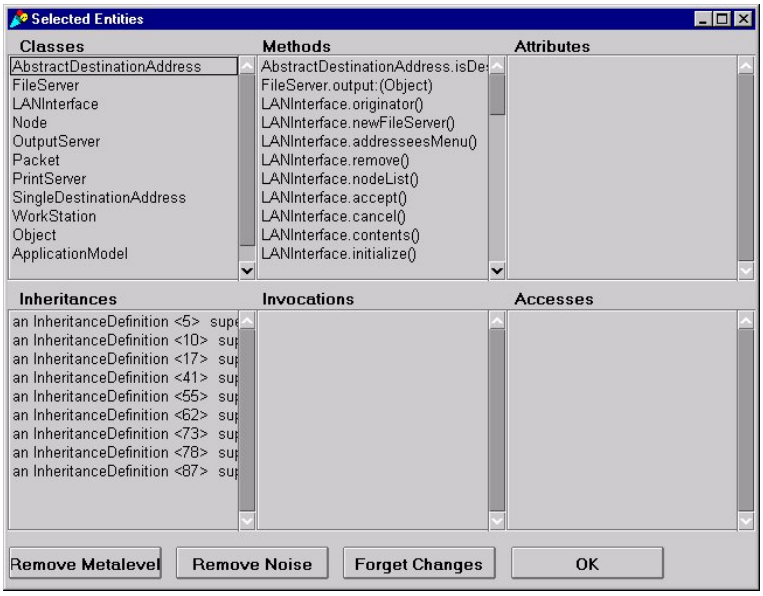

Figure 3.5: Screenshot of Model Editor (provided by MOOSE)

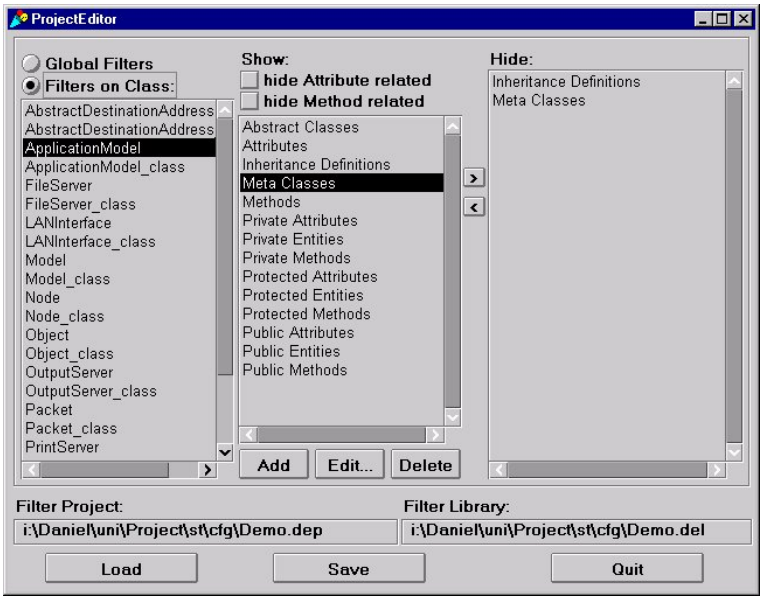

Figure 3.6: Screenshot of the Project Editor

Radiobutton *"Global Filters"*	The filters in the "Hide" list have global scope.
Radiobutton *"Filters for Class"*	The scope of the filters in the "Hide" list is limited to the selected class.
Listbox *"Classes"*	Shows all classes in the current model. Select one class to see its own filters in the "Hide" list.
Checkbox *"hideAttributeRelated"*	Hide all filters from the library that are applicable to Attributes only.
Checkbox *"hideMethodRelated"*	Hide all filters from the library that are applicable to Methods only.
Listbox *"Show"*	Filters in the library.
Button *"Add"*	Create new filter in library.
Button *"Edit Filter..."*	Open filter editor
Button *"Delete"*	Delete selected filter from library
Button *">"*	Add a filter to current project
Button *"<"*	Remove a filter from current project
Listbox *"Hide"*	Selected filters in this project.
Inputfield *"Project File Name"*	Path and filename of the Filter Project (.dep).
Inputfield *"Library File Name"*	Path and filename of the Filter Library (.del).
Button *"Load"*	Load Filter Project and Filter Library.
Button *"Save"*	Save settings Filter Project and Filter Library.
Button *"Quit"*	Quit Project Editor.

Table 3.2: Project Editor Explained

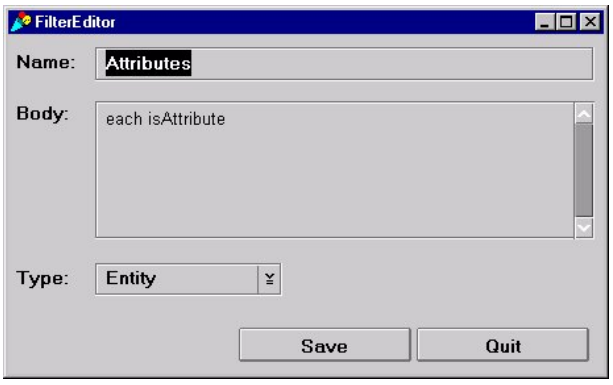

Figure 3.7: Screenshot of the Filter Editor

Inputfield *"Name"*	Name of the filter.
Inputfield *"Body"*	Specification of the filter.
Combobox *"Type"*	This filter is applicable to the selected entitytype only.
Button *"Save"*	Save this filter in the library.
Button *"Quit"*	Quit Filter Editor.

Table 3.3: Filter Editor Explained

Chapter 4

Case Studies

In the following chapter four case studies are presented. The examples include respectable up to good results but show also the limits of *UMLDesignExtractor*'s practicability. Enjoy it!

4.1 ColoredPoint

I have chosen this tiny model to provide you with a good example in the sense, that this complete model with all its indepth information can be nicely shown on one class diagram. ColoredPoint was imported from a .cdif file, and originally consisted of two Java classes.

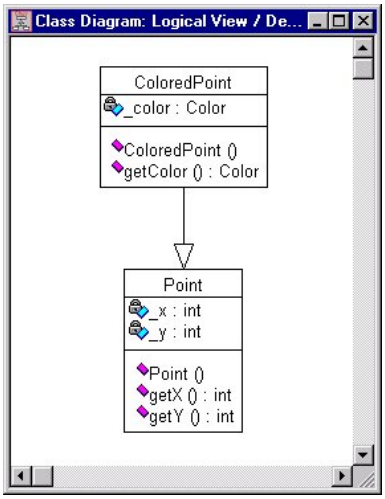

Figure 4.1: Class Diagram of ColoredPoint

4.2 LanApp

LanApp was selected to illustrate another model, that can be displayed with all its information just like it is. In this example we just do not exceed the magic limit of approximately 25 classes, where a diagram looses its clearness because of the vast quantity of information and because of the physical limits of readability, when the fonts just get too small. Maybe a developer team will produce a similar diagram with far more classes. But then it needs to be printed out in poster size, e.g. as a constuction plan for a whole system to hang on the wall. With such a plan you can get an view on a complete system within seconds. It shows the classes, its methods and relations.

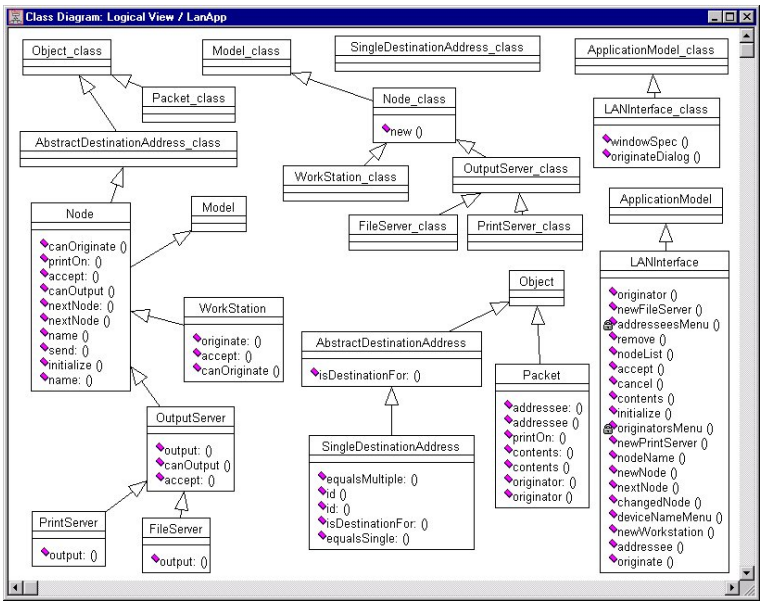

Figure 4.2: Class Diagram of LanApp

4.3 DesignExtractor

Figure 4.3 gives a view on all the classes of *UMLDesignExtractor* itself. Applied filters are 1) “Methods”, and 2) “Attributes”.

The model shown in Figure 4.4 contains the complete information of the *UMLDesignExtractor* application. Obviously there is too much information to be displayed in a window of this size. I know that there are plenty of methods and attributes that are not really relevant to the design of the system. What if we could get rid of them?

A reasonable filter allows us a representative view on a system’s design. To get Figure 4.5 and Figure 4.6’s models, first of all the complete model is loaded. After that we might remove noisy methods and attributes with the Model Editor. Candidate entities conceptually seem to come from protocols like “private”, “accessing”, “utilities” or

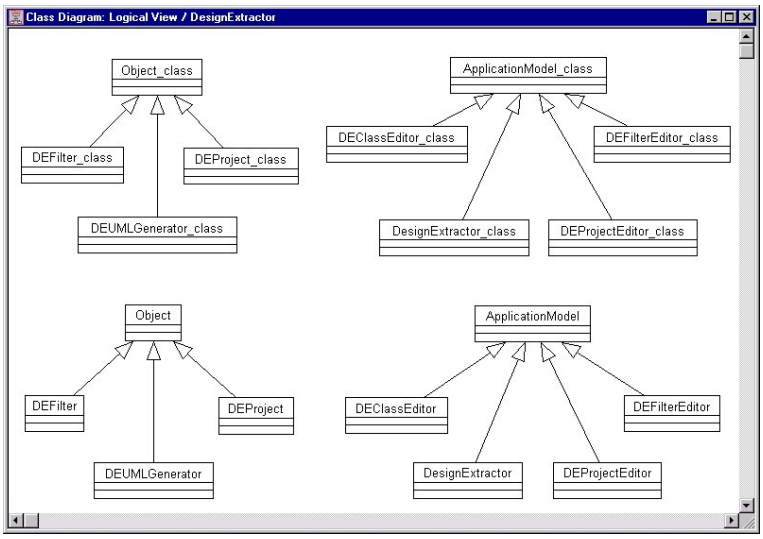

Figure 4.3: Filtered Class Diagram of DesignExtractor (Classes only)

“initialize”. To this reduced model further I apply the filter “Meta Classes”. Now I may apply “Children of ApplicationModel” and I get only the children of MSEAbstractRoot and in analogy vice versa.

4.4 MOOSE

MOOSE was selected to give a bad and ugly example. In this model I have imported all MOOSE categories with all its entities. I believe that any further comment is superfluous!

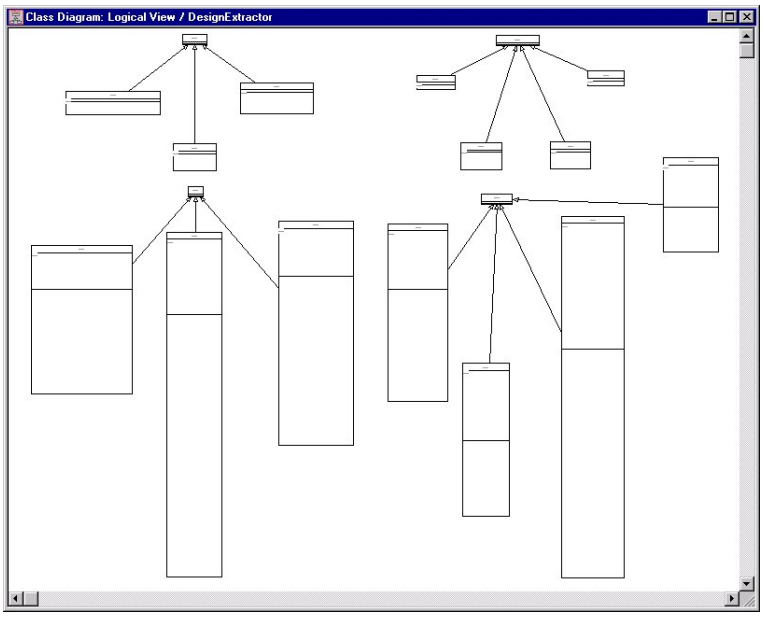

Figure 4.4: Unfiltered Class Diagram of DesignExtractor

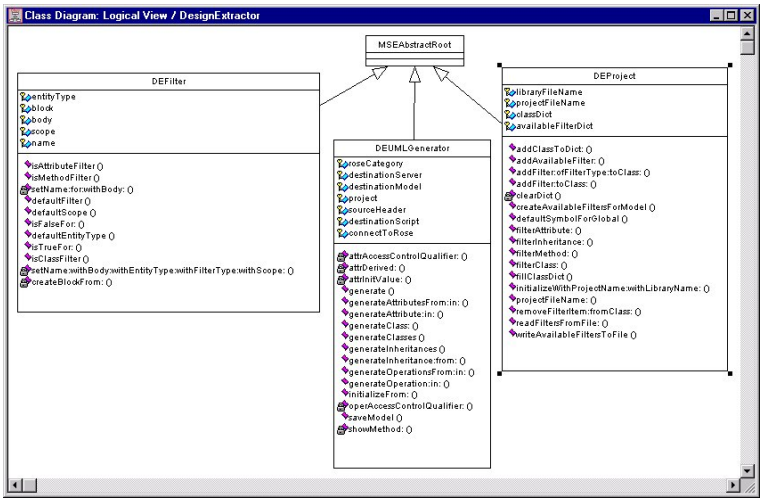

Figure 4.5: Filtered Class Diagram of DesignExtractor (MSEAbstractRoot)

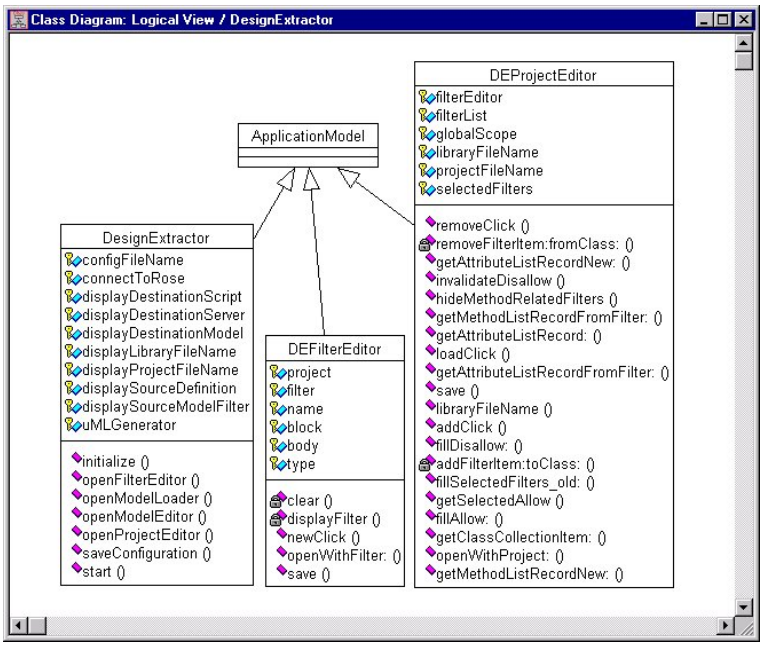

Figure 4.6: Filtered Class Diagram of DesignExtractor (ApplicationModel)

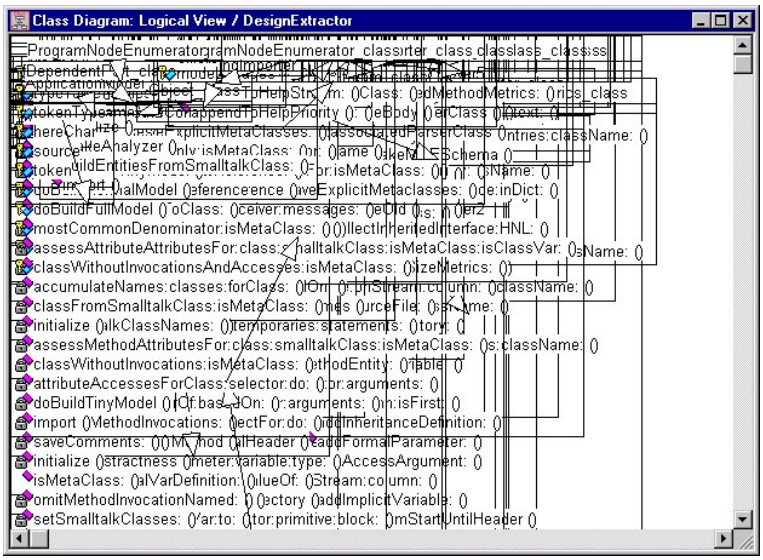

Figure 4.7: Unfiltered Class Diagram of MOOSE

Chapter 5

Project Experience

In this final chapter I tried to reflect my personal experience made during this work. I hope, hereby to be able to give some input to the planning of future computer science projects.

5.1 What Have I Learned ?

With this project I encountered many new technologies and architecture. At the very beginning I had to learn SMALLTALK. I was introduced in the development environment of VisualWorks and in the basic syntax of the language. In a next step I learnt how to build GUI's and I/O functions. With this basic understanding I was ready to be introduced to MOOSE.

By reading the Re-engineering Handbook and other articles in the periphery of the FAMOOS project, slowly I got a very broad view on software engineering. Especially my knowlegde about re-engineering, reverse engineering and the concept of object-orientation grew continuously. After that I begun to read books about UML. I had to understand the techniques, the corresponding specification and use.

In the next period I consulted the website and documentation of Microsoft's Component Object Model (COM). At the end I integrated all this knowledge in a reverse engineering tool prototype called *UMLDesignExtractor* . To finish the work I learnt how to work with LaTeX, a prerequisite to the existence of this document.

In many hours of pair programming with the group's assistants, I could greatly profit from decades of experience in the field of object-oriented development. That includes proper modeling, programming techniques and styles as well as design patterns.

Before this project I did not know that one could do anything else with a book than begin reading the first line and then go on line per line until the end. Now I hope to be able to pick the essentials of a book in a few minutes, just jumping through the contents and key sections.

Last but not least I improved my English in word (communicating with the SCG people) and writing (doing this report). Reading a lot of books as well as searching

through many websites for valuable information was an effort in better understanding the English language with a big return of investment.

5.2 What Have I Not Learned ?

I could not really discover the secrets of Rational Rose. From what I have seen, Rational Rose is not a tool that I would work with in the design phase. It is just not fast, handy and intuitive enough. So maybe I learnt to keep on working with good old-fashioned handwritten sketches or some kind of CRC's.

Definitively I did not learn how to work effectively from an economical point of view. This is the price you pay 1.) for the liberty to do what you are interested in, and 2.) for the luxury (but after my experience in practice 'unreal') circumstance that you may re-engineer an application not only until it works, but also until it fulfills all academic requirements like good design, styles and patterns.

Requirement specification, or time and cost estimation were subjects out of question in this project.

5.3 What Has Been Good ?

The subject of this work was chosen in a way that I was 'learning' far more than 'working'. That means: The final application consists of a couple of lines of code only. The main responsibility for me was to know all the involved techniques and architecture, and then integrate them.

The liberty to read a couple of books about design patterns, aspect-orientation, or also read about eXtreme programming gave me the ability to discover fields of interest, some of them without any direct coupling to the project's subject.

The group offered me pair programming whenever I needed it. This is a very effective way to learn from others. Generally the group members supported me whenever I stocked.

5.4 What Could Have Been Better ?

The project took me eight months - this is too much. I stopped working with my former company with the intention to be able to finish project and diploma during the next 9 month working 100% on it. This plan fatally failed. I consider this work to be far, far more than the expected 6E from the faculty's regimentation. (The diploma is postulated with 45E - will that take me 60 months or 5 years ?)

The coordination between the authors of MOOSE and its enhanced tools could have been better. I was programming new features only to throw them away some short time later, realizing that this was already implemented by someone else. This problem should now be solved with the just announced 'MOOSE coordination meetings'.

5.5 Considerations to Future Projects

As starter to object-oriented software engineering projects like this are great. I believe that the content of the work could have beed defined a little bit more precise from the beginning. A project plan could have been set up.

Especially if somebody will do a project in the field of MOOSE I propose him to switch immediately to ENVY. I did not do so - and therefore was never aware where exactly MOOSE was. Generally I consider ENVY to be a worthwhile experience as preparation to future teamwork, anywhere.

My experience was that even when I stopped working commercially, I was not able to go on faster with my project, than I would have, working 50% beside. One reason for that was myself: I just had too many other interests and the liberty to follow them. Another reason was the missing project plan: I only iteratively, by showing what I made, got the next task. Then, finally and in association with the previous reason, the coordination with the university blocked me many times, because I could not go on working before meeting people who were absent for a conference this week, for illness the next week, or for holidays that week. In fact, I was told so before - I just did not believe it. Knowing this I would have begun earlier with the project, not only when having finished all lectures.

Appendix A

File Formats

A.1 Configuration File

(*.dec ← **D**esign**E**xtractor **C**onfiguration File)

Entries:

Path to header file for Rose VB Scripts
Path to filter project
Path to filter library
Path to Rose VB script file
Path to Rose UML model
Computername or IP-Address

Example:

/home/dschwzr/project/rose/VBMethods.ebs
/home/dschwzr/project/st/cfg/Pixel.dep
/home/dschwzr/project/st/cfg/Pixel.del
/home/dschwzr/project/st/rose/Pixel.ebs
/home/dschwzr/project/st/rose/Pixel.mdl
euler

A.2 Filter Library

(*.del ← **D**esign**E**xtractor Filter **L**ibrary)

Entries: Filter [tab] Body [tab] ApplicableToEntityType [tab] Scope

Example:

Private Attributes	each isPrivate & each isAttribute	Attribute	$Global
Methods	each isMethod	Entity	$Global
Public Attributes	each isPublic & each isAttribute	Attribute	$Global
Public Methods	each isPublic & each isMethod	Method	$Global

The scope can either be global, or the name of a existing class. "$Global" is a constant expression representing the global scope. Because no classname ever begins with a special character like "$", we guarantee that there is no conflict between potential classnames and the expression representing the global scope. This constant is a (constant) message of the class "DEUMLGenerator" and could be changed anytime.

A.3 Filter Project

(*.dep ← **D**esign**E**xtractor Filter **P**roject)

Entries: Scope [tab] Filter

Example:

$Global	Methods
$Global	Class: ColoredPoint
Point	Attribute: Point._x
ColoredPoint	Protected Attributes
ColoredPoint	Private Entities

Bibliography

[ALPE 98] S. R. Alpert, K. Brown, and B. Woolf. Design Patterns in Smalltalk. Addison-Wesley, 1998. (p 21)

[BECK 89] K. Beck and W. Cunningham. *A Laboratory for Object-Oriented Thinking*. In Proceedings of OOPSLA '89, volume 24, pages 1–6. SIGPLAN NOTICES, 1989. (p 16)

[BECK 97] K. Beck. Smalltalk Best Practice Patterns. Prentice-Hall, 1997.

[BOOC 94] G. Booch. Object-Oriented Analysis and Design with Applications. Benjamin/Cummings, 1994. (p 16)

[BOOC 95] G. Booch. Object Solutions: Managing the Object-Oriented Project. Addison-Wesley, 1995. (p 16)

[COAD 91a] P. Coad and E. Yourdon. Object-Oriented Analysis. Yourdon, 1991. (p 16)

[COAD 91b] P. Coad and E. Yourdon. Object-Oriented Design. Yourdon, 1991. (p 16)

[COAD 93] P. Coad and J. Nicola. Object-Oriented Programming. Yourdon, 1993. (p 16)

[COAD 95] P. Coad, D. North, and M. Mayfield. Object Models: Strategies, Patterns and Applications. Prentice Hall, 1995. (p 16)

[DEME 99] S. Demeyer, S. Tichelaar, and P. Steyaert. FAMIX 2.0 - The FAMOOS Information Exchange Model. 10 1999. (p 8)

[DUCA 99] S. Ducasse. Smalltalk - a Pure Object-Oriented Language and its Environment. 1999.

[FAMO 99] FAMOOS. The FAMOOS Object-Oriented Reengineering Handbook. University of Berne, 1999. (p 8)

[FOWL 97] M. Fowler and K. Scott. UML Distilled. Addison-Wesley, 1997. (p 16)

[HOPK 95] T. Hopkins and B. Horan. Smalltalk: An Introduction to Application Development using Visualworks. Prentice-Hall, 1995.

[JACO 94] I. Jacobson. Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley, 1994. (p 16)

[JACO 95] I. Jacobson, M. Ericsson, and A. Jacobson. The Object Advantage: Business Process Engineering with Object Technology. Addison-Wesley, 1995. (p 16)

[JACO 99] I. Jacobson, G. Booch, and J. Rumbaugh. The Unified Software Development Process. Addison-Wesley, 1999. (p 20)

[MART 94] J. Martin and J. Odell. Object-Oriented Methods: a Foundation. Prentice Hall, 1994. (p 16)

[MART 96] J. Martin and J. Odell. Object-Oriented Methods: Pragmatic Considerations. Prentice Hall, 1996. (p 16)

[OBJE 98a] Object. COM Connect - User's Guide. Object Share, 1997-1998. (p 11)

[OBJE 98b] Object. VisualWorks - Application Developer's Guide. Object Share, 1993-1998.

[RATI 98a] Rational. Rose Extensibility Reference Manual. Rational Software Corporation, 1998. (p 20)

[RATI 98b] Rational. Rose Extensibility User's Guide. Rational Software Corporation, 1998.

[RUMB 91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorenzen. Object-Oriented Modeling and Design. Prentice Hall, 1991. (p 16)

[RUMB 96] J. Rumbaugh. OMT Insights. SIGS Books, 1996. (p 16)

[RUMB 99] J. Rumbaugh, I. Jacobson, and G. Booch. The Unified Modeling Language Reference. Addison-Wesley, 1999. (p 18)

[SHLA 89] S. Shlaer and S. J. Mellor. Object Lifecycles: Modeling the World in States. Yourdon, 1989. (p 16)

[SHLA 91] S. Shlaer and S. J. Mellor. Object-Oriented Analysis: Modeling the World in Data. Yourdon, 1991. (p 16)

[SHLA 97] S. Shlaer and S. J. Mellor. *Recursive Design of an Application Independent Architecture*. IEEE Software, vol. 24, no. 5, 1997. (p 16)

[WIRF 90] R. Wirf-Brock, B. Wilkerson, and L. Wiener. Designing Object-Oriented Software. Prentice Hall, 1990. (p 16)