ACKNOWLEDGEMENTS

In contributing to this thesis, we would like to thank our supervisor Ole Lehrmann Madsen for proof-reading and for being a big source of inspiration. We would also like to thank Henrik Rasmussen, Thue Kjelstrøm and Henrik Røn for their critical comments on early drafts.

Finally, acknowledgements to Adam and Lilith for never loosing faith and energy in the project.
## CONTENTS

1 INTRODUCTION ........................................................................................................... 1

2 COMPONENT MODELS .................................................................................................. 4

2.1 Component Architecture ............................................................................................... 6

2.1.1 Language Inspired Concepts .................................................................................. 6

2.1.1.1 Objects, Classes, and Interfaces ......................................................................... 6

2.1.1.2 Instantiation, Initialisation and Finalisation ............................................................. 8

2.1.1.3 Inheritance, Specialisation and Delegation ............................................................. 9

2.1.1.4 Encapsulation ......................................................................................................... 10

2.1.1.5 Polymorphism ......................................................................................................... 10

2.1.1.6 Composition and Communication ......................................................................... 11

2.1.1.7 Exceptions and Events ............................................................................................ 12

2.1.1.8 Introspection ........................................................................................................... 12

2.1.1.9 Persistence ............................................................................................................... 13

2.1.1.10 Concurrency .......................................................................................................... 14

2.1.2 Data Related Concepts ............................................................................................. 15

2.1.2.1 Revision Management and Dynamic Replacement ............................................... 15

2.1.2.2 Transaction Management ...................................................................................... 16

2.1.3 Distributed Concepts ............................................................................................... 16

2.1.3.1 Portability, Activation, and Instantiation ................................................................. 16

2.1.3.2 Naming and Location Transparency ..................................................................... 17

2.1.3.3 Load Balancing ...................................................................................................... 18

2.1.3.4 Security .................................................................................................................. 19

2.1.4 Peripheral Concepts ................................................................................................. 19

2.1.4.1 Customisability and Graphical Interfaces ............................................................... 19

2.1.5 Summary ................................................................................................................. 20

2.2 Commercial Component Models .................................................................................. 22

2.2.1 CORBA .................................................................................................................... 22

2.2.2 Java Beans ............................................................................................................... 25

3 THE MICROSOFT COMPONENT WORLD ..................................................................... 29

3.1 The Component Object Model ....................................................................................... 30

3.1.1 Classes, Objects and Interfaces ............................................................................... 30

3.1.2 Instantiation, Initialisation and Finalisation ............................................................. 34

3.1.3 Inheritance, composition, aggregation and containment/delegation ....................... 35

3.1.4 Polymorphism and Encapsulation .......................................................................... 39

3.1.5 Object descriptions and Introspection ...................................................................... 40

3.1.6 Method invocation .................................................................................................... 41

3.1.6.1 The Proxy and Stub Components .......................................................................... 42

3.1.6.2 Type Library/Automation Marshaling ................................................................. 43

3.1.6.3 Standard Marshaling ............................................................................................ 45

3.1.6.4 Custom Marshaling ............................................................................................... 48

3.1.7 Evaluation ................................................................................................................ 49

3.2 DCOM ......................................................................................................................... 50

3.2.1 Location transparency ............................................................................................. 51

3.2.2 Performance ............................................................................................................ 52

3.2.3 Connection Management ....................................................................................... 53
3.2.4 Security.......................................................................................................................... 53
3.2.5 Load Balancing .............................................................................................................. 54
3.2.6 Transaction Management ............................................................................................. 55
3.2.7 Evaluation ...................................................................................................................... 55

3.3 OLE .................................................................................................................................... 56
3.3.1 Automation .................................................................................................................... 56
3.3.1.1 V-Table, dual or dispatch? ........................................................................................... 58
3.3.1.2 Automation types ......................................................................................................... 59
3.3.2 Events ............................................................................................................................. 61
3.3.3 Exception Handling ....................................................................................................... 63
3.3.4 Persistent Storage ......................................................................................................... 64
3.3.5 Evaluation ...................................................................................................................... 66

3.4 Summary ............................................................................................................................ 66

4 DEVELOPMENT ENVIRONMENTS FOR COM ..................................................................... 70

4.1 IDE Appraisal ..................................................................................................................... 71
4.1.1 The Tasks ........................................................................................................................ 71
4.1.1.1 Task 1.1: Create a component with an arithmetic interface ........................................... 72
4.1.1.2 Task 1.2: Change the existing members of an interface .................................................. 75
4.1.1.3 Task 1.3: Add a member to an existing interface ............................................................ 75
4.1.1.4 Task 2.1: Add an interface to an existing component ...................................................... 76
4.1.1.5 Task 2.2: Remove an interface from an existing component .......................................... 76
4.1.1.6 Task 3.1: Add a new component .................................................................................... 76
4.1.1.7 Task 3.2: Remove a component ...................................................................................... 77
4.1.1.8 Task 4.1: Containment .................................................................................................. 77
4.1.1.9 Task 4.2: Aggregation .................................................................................................... 78
4.1.1.10 Task 5: In-process vs out-of-process servers .................................................................. 80
4.1.1.11 Task 6: Create a client with a graphical interface .......................................................... 80
4.1.1.12 Task 7: Change interfaces to implement the IDispatch interface ................................ 81
4.1.1.13 Task 8: Create installation disks .................................................................................... 81
4.1.1.14 Task 9: Exchange implementations .............................................................................. 82
4.1.1.15 Additional tasks ........................................................................................................... 83
4.1.2 The Taxonomy ................................................................................................................ 84
4.1.2.1 Component Server ........................................................................................................ 84
4.1.2.2 Compiling and Distribution .......................................................................................... 85
4.1.2.3 Reuse of Controls ......................................................................................................... 85
4.1.2.4 Development Environment Support .............................................................................. 86

4.2 Evaluation .......................................................................................................................... 89

5 TOOLS .................................................................................................................................... 92

5.1 The Toolbox ....................................................................................................................... 93
5.1.1 OO-CASE Tool ............................................................................................................... 93
5.1.2 Free Threadder .............................................................................................................. 93
5.1.3 Semantic Wrapper for Safe Arrays ............................................................................... 94
5.1.3.1 Considerations ............................................................................................................. 98

5.2 Idun, A CASE Tool for COM Development .................................................................... 98
5.2.1 Interface Design ............................................................................................................ 99
5.2.2 Interface Object Syntax ............................................................................................... 100
5.2.3 Object Functions .......................................................................................................... 103
5.2.4 Properties of Objects ................................................................................................................... 104
5.2.5 Interface definition input/output capabilities ............................................................................... 104
5.2.6 Editing utensils .............................................................................................................................. 105
5.2.7 Extending the capabilities ............................................................................................................. 106
5.2.8 Plugin implementation issues ...................................................................................................... 108
5.2.9 Reengineering .............................................................................................................................. 109
5.2.10 Integration with third party tools ............................................................................................... 110
5.2.11 Implementation .......................................................................................................................... 111

5.3 Summary ....................................................................................................................................... 112

6 CONCLUSION AND FUTURE WORK .......................................................................................... 114

6.1 Criteria for Evaluating Component Models .................................................................................... 114
6.1.1 Language Inspired Concepts ....................................................................................................... 114
6.1.2 Data Related Concepts ................................................................................................................. 115
6.1.3 Distributed Concepts ................................................................................................................. 115
6.1.4 Peripheral Concepts ................................................................................................................. 116

6.2 Commercial Models ...................................................................................................................... 116
6.2.1 Language Specific Concepts ....................................................................................................... 116
6.2.2 Data Related Concepts ................................................................................................................. 117
6.2.3 Distributed Concepts ................................................................................................................. 117
6.2.4 Peripheral Concepts ................................................................................................................. 118

6.3 Evaluating IDEs ............................................................................................................................ 118
6.4 Tools ............................................................................................................................................... 119
6.5 Future Work .................................................................................................................................... 119

7 KEYWORDS .................................................................................................................................. 121

8 REFERENCES ................................................................................................................................. 123

9 APPENDIX A: TYPETLIBMARSHAL ............................................................................................ 126

9.1 Server ............................................................................................................................................ 126
9.1.1 Project file: prjTypeLibMarshal.dpr ............................................................................................ 126
9.1.2 Main form: unitForm.pas .......................................................................................................... 126
9.1.3 Implementation unit: UnitImpl.pas ............................................................................................. 127
9.1.4 Type library wrapper file: prjTypeLibMarshal_TLB.pas ............................................................... 127

9.2 Client ............................................................................................................................................ 129
9.2.1 Project file: prjClient.dpr .......................................................................................................... 129
9.2.2 Main form: unitForm.pas .......................................................................................................... 129
9.2.3 Dialog box form: unitKidDlg.pas .............................................................................................. 131

10 APPENDIX B: STANDARDMARSHAL ..................................................................................... 133

10.1 Server ........................................................................................................................................... 133
10.1.1 Project file: prjMarshalStruct.dpr ......................................................................................... 133
10.1.2 Main form: unitForm.pas ....................................................................................................... 133
10.1.3 Server main code: unitImpl.pas ................................................................. 134
10.1.4 Type library wrapper file: prjMarshalStruct_TLB.pas .................................................. 134
10.1.5 Proxy/Stub def file: Proxy.def ................................................................. 136
10.1.6 IDL Code: MarshalStruct.IDL ........................................................................ 136
10.1.7 Proxy/Stub makefile: makefile ........................................................................ 137

10.2 Client ......................................................................................................................... 137
10.2.1 Project main file: Project1.dpr ........................................................................... 137
10.2.2 Client main unit: unitClient.pas ........................................................................... 137

11 APPENDIX C: AUTOMATION ..................................................................................... 139
11.1 Server ......................................................................................................................... 139
11.1.1 Project file: prjAutoTest.dpr ............................................................................ 139
11.1.2 Main unit: unitAutoTest.pas ............................................................................ 139
11.1.3 Type library wrapper file: prjAutoTest_TLB.pas .................................................. 140

11.2 Client ......................................................................................................................... 141
11.2.1 Project file: prjClient.dpr ............................................................................... 141
11.2.2 Main unit: unitClient.pas ............................................................................... 142

12 APPENDIX D: EXPLORER SPY ................................................................................ 147
12.1.1 Project file: MassaJrLib.dpr ............................................................................. 147
12.1.2 Main unit: unitForm.pas ............................................................................... 147
12.1.3 Event sink unit: WebBrowserEventsImpl.pas ................................................ 149
12.1.4 Type library wrapper file: MassaJrLib_TLB.pas ................................................ 153

13 APPENDIX E: TAXONOMY ..................................................................................... 156
13.1 Taxonomy Document ............................................................................................... 156
13.2 Taxonomy answers for Delphi ............................................................................. 162
1 INTRODUCTION

The 1990s became the decade when the Internet was commercialised in the shape of the World Wide Web (WWW). The reason that such a thing was at all possible can be sought in the gradual shift from mainframe to personal computer (PC) based systems up through the 1980s. Computer hardware went through an evolution, whose pace was previously unheard of, and today a PC is so cheap that most people can afford it.

In parallel with the hardware evolution, client/server programming is gaining still more popularity over mainframe computing, and connecting computers from different vendors in a local area network (LAN) in a manner transparent to the user has become a necessity. This factor in connection with the popularity that the Internet has gained outside the academic world, has meant that much attention is given to distributed computing by the software industry.

At the same time object-oriented (OO) programming has become the dominant paradigm, which means that developers tend to structure software in a modularised manner. Objects in different source modules may encapsulate different functionality of the entire program, display and storage management for example. The train of thought that OO provokes joined with that of distributed programming has meant that monolithic applications are no longer as popular as they used to be.

Monolithic applications have the inherent problem of being hard to maintain because the entire program must be recompiled each time a small part of it has been upgraded. The different parts of such applications can be hard to use across applications, and even so it seems wasteful to have the same code potentially reside several times on the same machine or indeed on different machines in a network.

A challenge that faces the software industry today is therefore to build frameworks for developing software that

- is **platform neutral** and can therefore run on any target machine,
- is **distributed** in nature and defines a standard for network communication,
- can be **reused** across applications without redundancy of code, and
- is **neutral** with respect to implementation language.

Component based software development (CBSD) is a recent invention, which targets these challenges. Central to CBSD is the notion of a component as a piece of software encapsulating some service, and which can be dynamically attached to different programs at runtime. Components therefore usually reside in some kind of runtime module.

CBSD extends the client/server paradigm by allowing components to interact between different machines and sometimes even move across a network. A component encapsulates behaviour and state, and as such resembles the traditional OO notion of an object, which is no coincidence since CBSD borrows most of its foundations from OO.

Building a component model in the spirit of OO is not an easy task at all. Many of the concepts from object-oriented programming are not directly transferable to CBSD, and some of them can only be partially mapped. The fact that components reside in runtime modules, the implementation of which is not visible to the user, makes inheritance and related concepts like specialisation and virtual methods very difficult to support in a component model.
The runtime modules in which the components live also raise a number of issues that must be dealt with. How is different versions of the same module managed? Can modules and thereby components be dynamically replaced even when there are clients of it running? But that is not all.

By supporting distributed computing, component models must confront and overcome difficulties pertaining to that area also. Some strategy for transparent interconnection must be defined, and schemes for dealing with network congestion, in the case of many clients using the same component, needs to be invented. In addition issues like access security and managing transactions between multiple clients of a component must be addressed.

Using components as building blocks in software development raises the problem of defining components that can be easily reused across applications, which may not be an easy task at all. By all means it is at least as difficult as building general classes in traditional OO. This carries over to the problem of using third party components as part of the application model. A developer needs to know exactly what a component does and how to use it, and even so fitting the component into the model may still be difficult.

With all the issues that component models embrace, actually using them to build software may not be trivial. A designer would very often benefit from possessing tools that support CBSD in a way that fits into the object-oriented design strategy that he has employed for the entire application. He should be able to select certain classes as suitable for being components, and should not need to specify the implementation language until such time as the entire design is finished. The tool should also automate trivial and bothersome tasks that pertain to the specific component model.

The topic of this thesis is component models and tools that support their use. The focus is on component models, not on how a design strategy using components can be carried out. Addressing the problems that have been outlined above, the structure of the thesis is divided into four areas.

First, an investigation of the issues that component models must address, seeks to build a foundation on which concrete, commercial models can be evaluated. The issues are classified with respect to the area from which they have been inspired, and argues why each particular concept should be dealt with by component models. On this basis a short review of how the commercial models CORBA and Java Beans tackle the issues, provides a comparative fundament for the next chapter.

The specific component model, which we have treated in depth is Microsoft’s Component Object Model (COM). COM has grown to become the facility for controlling applications externally on the Microsoft Windows platform, and a large majority of the operating system facilities can also be accessed through that technology. Today it is possible to control applications like Word and Internet Explorer fully through COM, which means that enriching new applications with text editing capabilities or web browsing facilities boils down to learning the COM interfaces of these programs. A developer can even hide the fact that other programs are used, by customising the applications and embed them within a frame of his own application.

COM is also viewed in the light of our requirements for component models in general, while using the examples of Java Beans and CORBA to contrast actual solutions to the issues. The chapter concludes with a detailed summary of how and how well COM relates to the general issues. In order to provide detailed knowledge of how COM components are built, the chapter will use sample code where needed.

Having focused on the model, it is time to take a look at development environments that claim to support COM based software engineering. Chapter 4 introduces a method for evaluating COM
support in development environments, which was defined as part of a research project under the Centre for Object Technology (COT). The method was used for investigating four different environments, in order to reveal strengths and weaknesses.

We were engaged in the project from the very beginning, and participated in defining the investigation method, which came to consist of two parts. The first, a test application that should be built in an environment sought to answer specific questions in the second part, the Taxonomy document. It became our responsibility to apply the method to one of the four environments, Delphi from Borland Inc., and so chapter 4 presents that investigation.

Working with the COM model and investigating environments for its support helped identifying areas that could be assisted by tools. The last chapter introduces a number of such utensils and provides a detailed discussion of a visual tool for designing and generating component source code for various languages. We have in fact implemented the better part of this last utility, and so the tool discussion is not hypothetical.

All source code referred to in the thesis including the tool and the corresponding binaries can be found on the companion CD, which is attached to the cover of this thesis. On the CD the actual evaluation from the research project as well as an electronic form of this document can also be found.

Throughout the chapters, important concepts will be highlighted with boldface. A list of these can be found on page 121.
2 COMPONENT MODELS

A concept, which really buzzes in the digital community right now is that of component based software development (CBSD). CBSD is an approach to structuring programs, which borrows its foundations from the object-oriented (OO) programming paradigm. An investigation of this concept, then, must start with a brief survey of OO, blazing the path for an understanding of what CBSD is, and how it fits into existing areas of software development.

OO hails out of Scandinavia where, rooted in Algol, the world’s first object-oriented language Simula was developed as a tool for simulating concurrent processes by Ole-Johan Dahl and Kristen Nygaard in the late sixties [OIOOP95].

The dominating programming paradigm at the time was the procedural, in which data and the functions that operate upon it is identified. With the introduction of Simula and the advent of object-oriented programming, program-structure became subject to a paradigmatic shift. With OO, code and data are organised into objects that conceptually represent the behaviour and state of a phenomenon within the problem domain. This way of thinking was inspired by the philosophy of Aristotele [BETA93].

In the classical Aristotelian philosophy the world is basically made up by things that consist of matter and form. It is not the matter of an entity, which characterises it, but rather its shape. The matter of an axe, for example, is wood and iron, but what makes it an axe and not a hammer, is its shape.

These concepts in place, Aristotele reasoned that things could be organised according to their shape. This way of thinking makes it possible to talk about axes as a concept, instead of forcing the description of every single axe-object in terms of its attributes, shape, matter, and function. Yet shape is but one category, by which entities lend themselves to grouping: a human, for example, may be characterised by its sex or by its lineage to name a few possibilities. Animals can be ordered according to their species, a task that Aristotele began some thousand years before Darwin [POLFIL95].

The OO paradigm borrows many of Aristotele’s thoughts, and defines the entities class and object as the basic abstractions. An object corresponds to an actual phenomenon, and the class to a concept, which identifies collections of objects. Just like animals can be classified into species, that have evolved from other species, so can classes be specialised to form new classes that are like their predecessor, but contain additional functions or attributes. This is the basis for the well-known concept of inheritance.

In the Scandinavian school of object-orientation, the focus is on building programs that correspond to models of the problem domain. This is achieved by identifying the phenomena and their attributes and actions, and by grouping these into collections that are conceptually alike. This makes the structure of the final program mirror the original problem domain, and so allows for a better understanding of both.

A benefit of the OO approach to programming is the fact that classes may be reused, when designed appropriately. With the popularity this programming paradigm has achieved world wide, OO has become subject to a shift of focus. In the better part of the literature outside Scandinavia, the benefits of reuse are the one major reason mentioned for developing programs this way. This means that the modelling aspect is suffering from diminishing attention, and that classes with no coherent connection to the original model are being made in the name of reuse.
So far reuse of classes has been obtained through class libraries, which are statically linked into the program at compile time. While this is an efficient strategy, the result may very well be that many programs potentially contain much of the same code. In the face of this fact, the notion of dynamic link libraries as a collection of functions has evolved to let many programs use the same code, without each program having its own copy. These libraries, which may be linked at runtime thus minimising the size of each program, have gained wide spread popularity.

The idea of externalising objects that represent some kind of general abstraction is obvious, when viewed in this perspective: just as there can be static and dynamic function libraries, there might as well be internal and external objects or class libraries. The internal objects exist inside the program and the external ones reside inside another module. In modern computer terminology, such external objects are known as components.

In the hardware industry, from whence the term originates, a component is an integrated circuit (IC). These are building blocks, which are made up of smaller blocks to form a functional whole that can perform some task. At the most primitive level, these hardware components consist of logical gates, which by being grouped together form larger logical units. When an engineer wants to add some functionality to his design, he will go and buy an IC, which does the task for him. He need not care how that component was made, as long as it performs as expected, and abides to the specifications promised by the vendor [COMINT96].

Software components are independent entities that represent some functionality, just like their hardware cousins. Components are connected to their host application at runtime, and so are not an integral part of the host binary. All that is visible of a component from the outside is its interface, which roughly corresponds to the legs of an IC: the component must be wired correctly to make it co-operate with the program. Just as a sledge hammer must be used to open an integrated circuit, it takes a disassembler to gain access to the internals of a software component. The bottom line of this is that the programmer should not – and need not – know the details of a component’s implementation in order to use it: it is by all means a black box.

While it is true that the notion of software components has grown out of a desire to reuse existing functionality in a manner more practical than that of runtime function libraries, there are several other reasons for adapting this strategy in software development. The perhaps most obvious advantage of splitting a program into smaller parts is that each of these can be regarded as separate entities, and so be developed by different people. In addition, the software can be incrementally updated by simply replacing a deficient component with a new and stronger one. This improves the maintainability of large programs, and reduces the cost of upgrading software [FCBSE96], [REUSE97]. Of course this is only true if the design of the components has been carefully crafted to support these notions.

Components need not be local to a single machine, but might very well be used by different applications in a network. This means that there must be some system-wide mechanism for locating, instantiating, and connecting components to running programs. As such, components provide an alternative to distributed models such as the client/server paradigm.

Even though the benefits of using components for software development may seem intriguing, there are a number of pitfalls and problems that face the developer. First of all, components are little black boxes, the innards of which he cannot see. This means that the developer has no way of knowing whether or not a component is in fact robust, fail-safe, and efficient, if he has not made it himself. Another problem is that it is hard to develop components that can be widely reused without compromising their size and specialisation.
2.1 COMPONENT ARCHITECTURE

Components can be regarded as the external cousins of ordinary OO-objects, and as such most of the issues pertaining to OO in general, spills over into the realm of component based software. This said, there are of course issues that do not match, and there are quite a few new ones to consider.

This chapter seeks to identify the criteria that a technology should address before it deserves to be called a component model. There are concepts that are more fundamental than others, and ones that are not strictly necessary to have a functional component model. The subsections of this chapter can be divided into four parts each with its own underlying inspiration. The first group of concepts relate to programming language constructs, the second to data management facilities. The third group is focused on distributed aspects, and the fourth on more peripheral concepts that are not as important as the first three groups.

We base the criteria on the requirements put forth by the software industry, and identified in a number of articles that discuss aspects of component models. The aspects are elaborated through literature about object-orientation in general, through issues from commercial models that we find important, and through our own experience gained by working with components.

2.1.1 Language Inspired Concepts

Component models are meant to be used in connection with traditional programming facilities, and as such most of the basic inspiration has been gathered from programming language constructs. Even though object-orientation is by far the dominating source of influence, concepts that are not specific to object-oriented languages, such as exception handling and event mechanisms, also play an important role.

2.1.1.1 Objects, Classes, and Interfaces

The basic abstraction, which separates object-oriented programming from procedural, functional or any of the other programming paradigms, is the notion of an object as a collection of actions, attributes, and data. The object models some phenomena of the world, either real or imaginary [BETA93].

Objects belong to a class\(^1\), which is a generalisation that identifies groups of objects. Classes may inherit attributes and actions from other classes; in effect making it possible to build hierarchies of abstract and concrete concepts that relate to the model. As an example, consider the hierarchy shown on Figure 1: a vehicle is a more general abstraction than a bus or a car, which in turn are more abstract than a Greyhound or a Mercedes Benz. The adapted notation is due to Coad and Yourdon\(^2\) [OOPAT92].

A vehicle, a bus and a car are abstract entities, whereas many Greyhounds may in fact be found, scattered across the USA. Greyhound is a class, of which new objects may be instantiated, whereas vehicle and bus are abstract concepts, of which this should not be possible. Abstract classes are marked with a grey border, whereas classes that should have concrete instances have a solid, black

---

\(^1\) This is not true for prototype based languages, but when we discuss OO languages in general and use the term “traditional”, we exclude the prototype based ones.

\(^2\) This syntax may seem a little outdated in a time where alternative representations such as UML dominate, but we feel that Coad/Yourdon syntax is by far the more readable.
one. Inheritance is depicted by means of an arc, where the super class is drawn above, and sub classes below it.

![Simple class hierarchy](image)

**Figure 1: Simple class hierarchy**

Classes are made up of attributes that define the visible properties of the objects that belong to them. The attributes can be inspected and modified from the outside, which may result in changes in the internal state of the object. Picking up the example from above, a vehicle may define properties such as wheel, engine, speed, windows, and so on. It is worth to note that in OO, classes should be built according to the phenomena in the model, they represent. So if the area of interest is the physical layout of cars, the wheel and the engine might become attributes of part objects of an encapsulating car object.

Apart from properties, classes have actions, usually modelled as ordinary procedures, that can be performed on object instances. Again the same modelling rules as described above apply to the actions of a class.

The notion of class and object carries on into the component world, and should be no different, since components are just external objects. Components belong to classes, of which new components can be instantiated. A component is characterised by having attributes and actions, usually known as **properties** and **methods**.

Just like some languages do, a component model might define a set of constraints for each property, thus preventing a user from setting illegal values on a property. As an example, a clock component would have properties “Hour” and “Minute”, the first of which should only accept values in the range zero to twenty-three, and the second only values between zero and fifty-nine. This is a nice feature, which will diminish the number of errors a client will provoke when using a component. To be practically useful, a programmer should be able to read from the component description, whether a property is restricted or not, and how.

Most OO languages introduce a way of separating the **interface** of a class from the actual implementation. This strategy allows other programmers than the developer of a class to gain knowledge of how to use instances, without bothering about the implementation. Languages like BETA, Java and Delphi employ this strategy, for example.
Consider an ATM\(^3\) in a bank, which will let a customer withdraw cash from his bank account, when it has not been overdrawn. To make the machine let go of his savings the customer will have to follow certain rules. First he must insert his credit card, then punch in his secret four digit code, then select the amount of money from the display and finally get the receipt and the cash from the machine. The keyboard, screen and card slot make up the interface of the machine that the customer must use appropriately in order to get some cash. He is totally oblivious to how the machine actually withdraws the amount from the bank account, and from where it gets the cash. All he needs to care about is to insert the card properly and punch the right code.

Components must also adhere to this scheme, because components cannot expose their implementation to the user, unlike ordinary classes. They are in effect black boxes with an interface, which is their only visible part.

Interfaces can be defined purely in terms of methods, since it is possible to provide “get” and “set” functions for simulating properties, a strategy known as the abstract data types approach (ADT). An ADT is a type, which is characterised completely in terms of its operations [OIOOP95].

2.1.1.2 Instantiation, Initialisation and Finalisation

All object-oriented languages have a means of creating a new instance of a class or prototype, either by a keyword or by special methods. Languages like C++ and Delphi equip their classes with such methods for constructing new instances of the class, and for destroying these again. In C++ the keyword “new” will also be used in front of the class of which a new instance is wanted like

\[
\text{MyObj} = \text{new MyClass()}.
\]

In Delphi, a special method must be executed on a class variable to get a new instance like

\[
\text{MyObj} := \text{MyClass.Create}.
\]

Termed constructors and destructors, these methods allow explicit control over object life times, but if the destructor is not invoked, when an object is no longer in use, this strategy will unfortunately also be a source of memory leaks. The constructor contains code that is executed when the object is instantiated, allowing initialisation of local variables and special processing to occur. The same is true for destructors, which execute when the object dies and thus provides a means of finalisation.

Constructors and destructors are not the only means by which a language may provide initialisation and finalisation support. Languages such as BETA and Simula equip their objects with so called “do parts” that are executed when the objects are created, and Smalltalk will allow the definition of constructor-like methods through its meta-class construct. Generally, therefore, providing a means for initialising objects seem to be part of most OO languages, whereas finalisation is more rare.

Since components can be regarded as external objects, belonging to some class, creating new instances and initialising these is a fundamental issue. A component model is not a language, and so keywords cannot be used for constructing new instances. Instead the model will have to provide some framework for creating new instances of a class and connecting to its interfaces. Similarly it must define some mechanism, by which a programmer can specify special code that is to be executed when the instance is created. A component model lacking a standard way of creating new instances will be as useless as an object-oriented language with the same problem: if it is not clear exactly how to instantiate a new object, what use is it?

---

\(^3\) ATM: Automated Teller Machine, a machine by which cash can be withdrawn by using a credit card.
Initialisation would definitely also benefit from being standardised, but since this is an implicit event taking place when a new instance is created, standardisation of this feature is not crucial.

2.1.1.3 Inheritance, Specialisation and Delegation

The feature, which most people connect with object-orientation is inheritance. In ordinary OO, new classes can be built from existing ones by inheriting both their implementation and interface in the new subclass. The new class may access the internals of the parent class, and so in effect is an extension of its ancestor.

Specialising the behaviour of subclasses is a concept, which is closely related to that of inheritance. Subclasses may need or want to redefine or extend functions that were originally defined in a superclass, and methods that allow this are usually known as abstract or virtual. Most OO languages employ a mechanism for providing this feature, which was first introduced by the Smalltalk language [SMALLTALK80]. With this scheme, a virtual method may invoke the equivalent implementation in the parent class by referencing to that class’s implementation through a special keyword. In Smalltalk and Java this keyword is “super”, in Delphi it is “inherited”, and in C++ one will refer to the name of the superclass followed by “::” and the name of the function to invoke.

The Simula and BETA languages employs another strategy for specialising functionality in subclasses via the INNER keyword. This construct turns things upside down, in that virtual methods in subclasses will have their implementation executed as part of their ancestor’s method body, namely at the position of the INNER keyword. This lets the subclassed method body become part of the superclass body, and so all local variables in the super’s method are visible in the subclass’s specialisation of that method 4.

In connection with component models, inheritance and specialisation turn out to be difficult to model in a manner similar to that of the OO languages described above. First of all, the fact that components must separate interface and implementation, the latter of which cannot be controlled from the outside, makes specialisation impossible. Even though a manner by which components can inherit the interface from one another may be defined, providing access to the local variables of an ancestor’s implementation of specialised methods is not possible since these are hidden in the black box.

All is not lost though, since by allowing components to inherit the interfaces of one another, the methods of ancestral components may still be used in a manner similar to the Smalltalk abstraction. In the OO literature, using an existing implementation as the basis of a new one by invoking it, is known as delegation. This is a simple mechanism, by which the sub-class simply acts as the client of another component, which implements an ancestral interface. This exact way of reusing old implementations is discussed further in connection with containment in section 3.1.3.

Another stratagem than the above described may also be employed in order to model inheritance. The Self language [SELF87], which is a prototype based OO language, uses the notion of a “self” pointer to let a method refer to the object that invoked it. The Self strategy is the opposite of that provided by Smalltalk via the “super” keyword: the called object gets a reference to the caller, which it can use to gain access to members and methods in it.

A similar strategy may be used in component models by the client explicitly passing a pointer to one of its own interfaces as a parameter in the construction of the component, whose methods it

4 See [BETA93] for a more thorough description.
wants to use. This will allow the old component to set the properties and invoke the methods of the owner, much in the same manner as Self does for objects. This strategy is possible with Microsoft’s COM, as will be further discussed in section 3.1.3.

Any component model should support interface inheritance, so that the structure of the problem domain may be reflected in the model. Further it should not hinder delegation as a means of reuse, and may support a more sophisticated method via a self pointer.

2.1.1.4 Encapsulation

Adding access restrictions to the members of a class is something many OO languages allow. Sometimes it is for example desirable to indicate that certain variables are tightly coupled to the implementation of certain functions in the class, and should not be tampered with in subclasses or from the outside. These would then be marked as private. On the other hand subclasses might benefit from adjusting certain properties that have no meaning from the outside. These should therefore not be visible from the outside and could in turn be marked protected. Finally some properties should be accessible from everywhere, which would classify them as public.

In general known as encapsulation, different languages provide different ways of enabling this feature. Most languages support the notion of private and public members, either implicitly or explicitly by having keywords for it. The BETA fragment system makes it possible to separate interface and implementation, thus in effect providing the most basic encapsulation mechanism [BETA93].

The Delphi language as well as C++ and Java provide the keywords private, public, and protected, a strategy known as the export approach [OIOOP95]. In addition to these keywords, C++ defines the notion of “friends”. These are functions that do not belong to the class, but are permitted to use the private and protected names from the class. The name of a friend is not in the scope of a class, and so members of the class have no way of accessing friends [CPPPL93].

With components information can easily be hidden by taking the BETA approach, and simply not expose private members in the interface. Similarly public members are simply added to the interface, and are readily accessible. It is hard to see how sophisticated access specifiers such as protected and friends could be integrated with component software. This would require some kind of mechanism, in which a component could identify itself to another component, as implementing a sub-classed interface of the other. Even so, providing support for private and public members is the most basic mechanism, and so every component model should at least provide support for these constructs.

2.1.1.5 Polymorphism

The ability to treat objects of the same base class in a uniform manner, is a feature which most object-oriented languages expose. The idea is that common functionality for a collection of conceptually related objects can be defined in a super-class. When advantage of that functionality is to be taken, one can be totally oblivious to the actual qualification of the object, as long as it inherits from the common ancestor. This feature is known as polymorphism.

As an example, an abstract super-class “DrawableObject” of which there could be several sub-classes like “Button”, “Window”, and “Bitmap”, as depicted in Figure 2, might be defined. In the super-class there would be a function “Draw”, which each of the sub-classes would specialise, and provide their own implementation of.
Now, a program might declare an array of DrawableObject, make new instances of “Button”, “Window”, and “Bitmap”, and assign these to the elements of the array. Later on, the program could then make a sweep through the array, invoking “Draw” for each element, e.g. to make them visible on the screen. The “Draw” function has many shapes, and so is polymorphic.

Polymorphism is coupled to the concept of dynamic binding of variables, since a language must check that each object has the right qualification before invoking the method in question. Many languages, which are inherently statically typed, provide this feature also. This means that most, but not all types can be checked at compile time, and the language must have some kind of exception mechanism for halting in the case of illegal use. This is true for BETA, Java, Delphi, and even C++, the latter of which provides the “dynamic_cast” function for changing the qualification of an object.

A component model should provide the flexibility of polymorphism, for two reasons: first of all it is very convenient to be able to treat collections of related objects in the same manner, as the example above should have made clear. Secondly it strengthens the conceptual bonds between super-class and sub-classes, and so enables one to build a hierarchy which closely relates to the original OO model. In the example above, a “Button” is also a “DrawableObject”, and so should be able to be treated as such.

Since components are black boxes, of which only the interfaces are visible, the polymorphic nature of components will naturally relate to the interfaces only. If the class hierarchy of Figure 2 was implemented as components, there would be a super-interface named “DrawableObject”, from which the others will inherit. By regarding “Button”, “Window”, and “Bitmap” interfaces as if they were “DrawableObject” interfaces, the Draw function can still be invoked on each and thus provide polymorphic behaviour. Restricting polymorphism to interfaces is no big deal, since it is in fact about invoking methods, without being concerned about the implementation.

2.1.1.6 Composition and Communication

As with the integrated circuits of the hardware analogy, components should have the ability to be used as building blocks for other components. In traditional OO languages, this facility allows objects that are somehow related other than in an inheritance hierarchy to be grouped. Consider the example of Figure 3: a car typically has four wheels, a couple of doors and an exhaust pipe. The wheels, doors, and pipe could be modelled as separate objects, as indicated by the arrows, which could be part of the larger car object. This ability is known as composition, and is a strong companion in building the correct model of the problem domain.
Composing new components from existing ones is a defining feature of the technology (see for example [REUSE97]). By grouping together collections of interfaces, it should be possible for a component model to facilitate both simple and complex components of various granularities as noted in e.g. [GAP97]. Yet, since the ability to use components as part objects is such a crucial feature in using them at all, building components from sub-components should be no different from building programs using this strategy.

Sometimes part components may need to communicate, in order for the entire component to behave appropriately. Similarly the outer component must have a way to interact with the ones it contains. No matter the physical location of the components, the communication should take place in a uniform manner. A component accessing methods and properties of its part components should by no means be different from a program using components.

2.1.1.7 Exceptions and Events

The concept of components that communicate carries on into that of propagating error messages back to the caller. Many languages, including some of the object-oriented ones, define the notion of exceptions for handling error situations. Exceptions can be modelled as messages or objects that are passed back out of the method, in which they were thrown, and then handled by the caller or one of its callers.

Since error situations are in no way unique to ordinary programming languages, but can be expected to occur inside components, also, a true model should provide a standard exception mechanism. It must be possible for a program to respond appropriately to any error, and take evasive action to let the entire program continue to function.

Closely related to exceptions is the notion of events. Like exceptions, events occur inside a component and must be propagated back to the caller. This abstraction of a server interrupting its client’s execution, is a highly successful one, and is the basis of most windows-based operating systems such as those of Microsoft and Apple.
Events are similar to the general Observer design pattern discussed in [DP96]: a subject may have any number of dependent observers that will be notified whenever the subject undergoes a change in state.

Like exceptions, events are not an inherent object-oriented feature, but they are definitely necessary in a component model. In a model with no event mechanism, a client requesting data from a server, with a lengthy and time consuming execution, must frequently poll the server to check if the requested data has become ready. This is highly inefficient, and certainly not desirable.

Since exceptions and events are concepts that a program must take into consideration when using a component, the model should provide some way of specifying that an object will be the source of events or might throw this or that exception. The Java language provides a good example of this strategy: in Java the exceptions that may be thrown can be specified as part of a method declaration [JAVAS96].

Exceptions may be modelled as messages or objects, whichever is the more appropriate for a given component model. Messages may be strings or pointers to structures, whereas exceptions as objects will have to be pointers to interfaces. We feel that modelling exceptions as components is superior to defining them as messages, since doing so will provide a cleaner object model: just as there should be a homogenous way of connecting to objects, it should not be necessary to worry about too many abstractions, such as messages or structures, when there is no need.

2.1.1.8 Introspection

Learning about the capabilities of a component is an important task, in order to be able to use it. Before launching a component, one might at least want to know which interfaces, methods, properties, events, and exceptions that pertain to it. Furthermore details of parameters for methods and maybe even constraints on properties may be needed. Examining a component or an object to get this kind of information is called introspection.

To this end a component model should support an efficient and flexible model for introspecting the capabilities of objects. This should be possible without instantiating the component, since such an invocation will potentially slow down the process, and because it may not even be possible to instantiate it correctly without knowing its details first.

Component information may come in different flavours: one strategy is to link component and information into a unity, another to separate the two. The information may be distributed with the component or stored in central repositories for global access. Whichever method is employed, a component should always support introspection mechanisms.

2.1.1.9 Persistence

Since a component represents some aspect of a problem domain, and thus contains data that can be manipulated as the simulation runs, there may be a need for saving its state to a persistent media. This feature, known as persistence, is present in some object-oriented languages, e.g. BETA, but far from all. In a large majority of the OO languages, writing objects to a persistent storage, is something that the programmer will have to implement himself.

As discussed in [TOP97] there are three main principles governing the acquisition of persistence in a system:
• **Persistence independence**: an object’s persistence does not depend on how programs manipulate it and programs are expressed independently of the persistence of the objects they control.

• **Persistent data type orthogonality**: all objects no matter their kind should have the ability to be persistent.

• **Orthogonal persistence management**: the choice of how to provide and identify persistence is independent of the type system, computational model, and control structures in a language.

These three principles can easily be applied to component models, with slight changes to the third rule. This could then be restated as follows:

• **Orthogonal persistence management**: the choice of how to provide and identify persistence is independent of the type system, computational model, and control structures in a language that implements component classes.

Component models should provide persistence and abide to the three rules for maximum flexibility. It should be possible only to specify parts of component attributes as being targets of persistence to avoid unnecessary overhead.

Since component models should be independent of implementation languages, rule one and rule three are inevitable. Rule two ensures that a programmer does not have to bother identifying persistence capabilities in objects before he uses them.

There are different ways that persistence could be added to a component model. The most simple method is to let the components themselves handle the details, but this scheme has the disadvantage of imposing the burden of implementing persistence on the programmer. He will thus be confronted with issues that are not directly related to the component functionality, he is currently defining. It should come as no surprise that this scheme is the one, which has been chosen in most commercial object models.

Another strategy could be to have a centralised mechanism in the component model, which kept track of all instances and their relationships to other components. Such a manager could employ different schemes to ensure that the objects that an instance depends upon were in fact also written, when that instance was made persistent.

One such scheme could be to calculate the transitive closure of all referred objects. While this is definitely not trivial, some programming languages like C++ are in fact unable to supply such a facility. In the case of C++, the untyped void pointer makes it impossible to infer the type pointed to, and so the transitive closure cannot be built.

In a component model, being able to write the transitive closure of a cluster of components is perhaps even more difficult. Writing a singular object might be achieved by having the centralised manager keep pointers to all instances that have been created, and then support a mechanism, by which a program can have its objects stored. In the case, when a component is made up of smaller parts, these must also be made persistent, and so the manager must keep track of all instances and the ones they refer to, to be able to build the transitive closure. The persistence manager must be an integrated part of the component model, and is probably not trivial to build. On the other hand, all components can be persistent from birth, and programs as well as components need not take any special action for this to be true.
2.1.1.10 Concurrency
Modern programming languages usually provide mechanisms, by which a program may consist of multiple threads of execution, instead of just one. In relation to components, there are at least a couple of issues regarding concurrency, viz. that of building multithreaded components, and that of having multiple clients connect to the same component.

Since components are black boxes, having one or more threads of execution running inside an instance, does not change anything with respect to the clients using it. It should therefore be up to the programmer how he would like to implement a given component, single-threaded or not.

In the case of multiple clients connecting to a single component, the model should disallow objects to use shared data. Firstly this will make the components non-thread-safe, thus forcing the programmer to guard the critical data with software sentinels such as semaphores or monitors, and second guarded regions might well hamper the performance of many programs, since these would have to wait for the shared data to be released, before they could continue executing.

Unfortunately a programmer will have to consider making his code thread-safe, in the case when a client program is multithreaded, and is using the same instance of the object with multiple threads. It is hard to see how multithreading problems could be dealt with in an efficient manner in other contexts than that of the programmer.

Other than this, a component model should flexibly allow any number of instances of any components, connected to any program.

2.1.2 Data Related Concepts
The fact that components are external entities makes it necessary to view them as data, also. In an environment with more than one client of a component, for example, it may be necessary to have mechanisms for controlling concurrent access. Additionally components may be substituted with new versions, and so there is a need to manage revisions. The concepts of this second group are inspired by ordinary data handling mechanisms and data bases.

2.1.2.1 Revision Management and Dynamic Replacement
When programming, it can often be convenient to be able to manage different versions of a source file. When a new revision of e.g. a class in a source file is defined, the entire source must be committed as a new revision. This is in contrast to what a component model ought to provide. Since software usually evolves over time, and components can thus be expected to emerge in different revisions, there is a strong need for a predefined versioning mechanism that all component providers will abide to.

A new version of a component should expose at least the same functionality as its predecessor, and perhaps more. This means that the properties or methods from interfaces must never be removed, just as it should not be legal to change their declarations, because this would mean that existing clients might cease to function.

In effect this means that an object model should support a way of augmenting existing components that would allow old clients to continue working with the old part of the interface, while letting new ones take use of the added members as well. This is what is meant by revision management.

When a new version of a component has been defined, a programmer should be able to commit the new piece of software to the system, in effect substituting this for the old one [GAP97]. At this
time, a number of applications that are using the old version of that component may be running. Of course the most ideal situation would be if these applications had their running instances of the component dynamically replaced with the new version, while maintaining the internal state. In reality this is probably too difficult to handle, and a more simple scheme might suffice.

Instead the system could contain a global component server, to which a new version of the component could be committed. This server would then use the new component with all successive requests, while maintaining the old one for the applications that were using it at the time the programmer committed the new version. When all applications had released their hold on the old component, the upgrade would be complete. In the case that the programmer regrets the commit of the new version, he should be able to revoke it and let the system return to its prior consistent state with the old version. In effect the server should thus be transacted. Note that there could be a potential danger here in the case that some client had started using new functionality of the new server.

Replacing components raise a number of issues that ought to be addressed. First and foremost the substitution should not give rise to any side effects in the clients. Second, the component model should carry out the tasks involved in replacing the old component or rolling back to a prior version without the users needing to know or care [USAGE97]. In practice providing a means for such dynamic replacement of components at runtime is not of the most fundamental concepts in a component model.

2.1.2.2 Transaction Management
In a distributed environment, multiple clients may access the same servers of some functionality, possibly each performing a sequence of actions on it. Consider a database, which is updated by multiple clients simultaneously. Each client wants to perform at least a couple of actions sequentially to update the data of the server. Without any co-ordination, operations from two or more clients might get mixed up, effectively leaving the data on the server in a state, which neither of the clients expected. This is a well-known problem within the area of distributed computing, and is usually solved by introducing transactions and transaction management.

By integrating a dedicated service to maintain the access to shared resources, a component model can help leverage this problem, by providing a transaction mechanism in which a sequence of actions can be grouped to form an atomic entity. Since components can be expected to need access to shared resources quite often, there will be a need for a standard way of dealing with this problem. Such a standard will allow the use of new kinds of resources, which might not even have been present at the time a component was built, and so ought to be part of the model.

2.1.3 Distributed Concepts
The advent of global as well as local networks, has brought interconnection of platforms into the limelight. Components need not be local to the machine on which they were defined, but lend themselves to distribution. This immediately implies that there are a number of concepts from the world of distributed programming that also apply to component based software.

2.1.3.1 Portability, Activation, and Instantiation
A component model should not be restricted to one particular architecture or operating system, but should be fully portable. Nor should a component model restrict itself to one specific target
language, but instead expose itself to any system and language that may want to build or use components.

In an ideal situation, components should be able to move across a network to live on another computer, an action known as migration. This would improve performance in the case, when the network connection is unstable, congested or slow. The migration should be transparent to the client of a component, and the provider should be equipped with a means for tagging components as being migrateable.

<table>
<thead>
<tr>
<th>Activation \ Execution</th>
<th>Local</th>
<th>Remote</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local</td>
<td>The component is activated at the client machine, and a new instance executed there.</td>
<td>The local component is activated, and sent to a target machine where a new instance is executed.</td>
</tr>
<tr>
<td>Remote</td>
<td>The remote component moves to the client machine, where a new instance is executed.</td>
<td>The remote component is activated and a new instance executed on the remote machine.</td>
</tr>
</tbody>
</table>

Figure 4: Activation/Execution scheme

The fact that components may reside on any machine in a network makes the combinations of Figure 4 possible: on the vertical axis is the locations, where a component might be activated or woken up, and on the horizontal is the locations, where the awakened server might make a new instance of an object and execute it.

Today most commercial models support three of these combinations. All of them allows local components to be executed on the machine they live, i.e. local activation and local execution. Likewise, support for activating and executing an instance on a remote machine is widely supported, e.g. by DCOM as described in chapter 3.2. Some of the models allow remote components to migrate to the client machine to be executed there, e.g. Java Beans and ActiveX, and finally we have not encountered any that support the last combination. Even so it could possibly be useful to activate local components and ship them off to target machines, e.g. as spies or agents of some kind.

Since there may be a need for all four possible combinations of activation and execution, ideally a component model should allow its components to live wherever they want, and migrate as needed. It should be possible for a programmer to select acceptable locations for instances of the components he uses, in order to tune performance. On the other hand some components might be tied to a specific machine because they service some special hardware for example, and so a component developer should also be able to restrict the activation/execution possibilities.

2.1.3.2 Naming and Location Transparency

When a program wants to use a specific component, it should ideally not have to bother where that object resides, a concept known as location transparency. For performance reasons, however, it might be necessary to state that only local objects are of interest or that the object should migrate after being instantiated. An object model must supply location transparency, when the client does not care, but must also be able to launch objects whose location has been explicitly given.
To be able to refer to the correct object, i.e. the one which implements a certain behaviour, which is desired, is a fundamental facility that any component model should somehow expose. There are at least a couple of ways this concept called **naming** could be supported. Each object could for example be tagged with a unique name, which no other component in the world has. This strategy calls for a strong identifier generation algorithm, but is possible to undertake, as we shall see in section 3.1.1 that discusses Microsoft’s component object model’s solution to the naming problem.

Another strategy is to provide a centralised repository, in which each component resides. These could then be stored in a hierarchical manner similar to an ordinary directory tree. At the topmost level is the repository, and directly below this the component servers that may potentially contain many components. Next comes the components themselves and their interfaces, and under these methods and properties. Such a repository is known as a namespace, and providing naming facilities through such a service is the strategy employed by CORBA as discussed in section 2.2.1.

Whichever method is employed for providing naming capabilities, the bottom line is that no component model can do without it, and so this issue must be addressed.

The basic naming facilities could be enhanced to work network-wide: in a large network with thousands of components, classifying these according to their functionality is a must in order to keep the chaos under some control. A centralised object manager could for example have a list of all objects, their interfaces, and a description of their services. The manager could then provide a query mechanism, by which programmers or programs could locate components of specific functionality.

**2.1.3.3 Load Balancing**

Location transparency brings performance into consideration. Imagine a situation, where many clients rely on the same service from one computer in a network. All clients maintain a remote connection to the object, thus congesting the net, and slowing down the overall performance of the server.

This situation can be remedied by using a technique known as **replication**, which makes a copy of a running object instance, moves this to a less burdened server, and forces a suitable amount of the clients to use the copy instead. Replication is not at all a trivial problem to solve, but the presence of this feature will allow a network to scale, either in a dynamic or static way. The act of detecting network congestion and doing something about it is usually known as **load balancing**.

The component model could support a dynamic means of allocating new servers, and redirecting new clients to these, thus over time reducing the load of the original server machine. A component model should allow components to be mobile, but since some will be tightly coupled to some intricacy of their host machine, this feature should be optional.

Balancing network load in a manner thus dynamically is quite difficult to implement, and fortunately another strategy might also be employed. Given the right tools for monitoring the overall performance of the network, an administrator could manually maintain replications of bottle-neck objects, and statically assign clients to the different versions.

While it is true that the dynamic approach is by far the most superior because of its adaptive nature, most commercial component models have restrained themselves to static load balancing because that is by far the easier to implement.
2.1.3.4 Security

Components residing in a network, and possibly even somewhere on a global one like the Internet, need some kind of security mechanism for restricting their use. There are many aspects of security, some of which pertain to components, and others that are related to the network’s own security model.

To stick to the last one first, certain servers could be set up for exclusive use by a subset of the users. This kind of security enables some people to use all objects on a given machine, and others none, and is possibly too coarse. A component model should thus be able to tag components with security information, effectively enabling or prohibiting clients to launch these. Moreover, it could be possible to restrict parts of an object’s interface, by tagging methods and properties also. For properties it might even be desirable to provide security information for both reading and writing to and from their values. Integrating component security mechanisms with those already engaged at the operating system, would improve the model further, by helping the object model keep security issues somewhat more transparent to the user.

Since of-the-shelf-components can be expected to become an industry, providers will want to tag components with license information: in order to use such an entity, the serial number which came with the package, in which the object resided must be passed along. This feature ought to be part of the model, since a standard will again help keep things uniform and simple.

2.1.4 Peripheral Concepts

The first three groups of concepts are the more important ones that component models must address, but there will of course be numerous other aspects that can be investigated. This section introduces two of the more popular, yet peripheral, concepts that could also be treated.

2.1.4.1 Customisability and Graphical Interfaces

Turning to the commercial component models, many of these define components with the ability to have a graphical user interface (GUI). While modelling a user interface through components in an object-oriented manner is definitely obvious and practical, providing a framework, which will work across platforms is known to be a hairy business. Since different operating system vendors have their own way of building the interface APIs\(^5\), it will be difficult to maintain an identical look and feel across platforms for components with a graphical interface.

Many commercial models have some way of defining visual components, but we feel that this is an issue, which is not directly related to the component model. Instead GUI standards should emerge to form a solid basis by which all applications could conform to allow cross-platform interoperability at the topmost level. This said, a component model should not be restricted to non-GUI components, but let the programmer build objects with a visual interface also.

For components with a visual interface, a neater way to customise their internals than simply adjusting properties via code, might be defined. Some commercial models, e.g. Java Beans and ActiveX, will allow the modification of properties of a visual component through a special window often known as a property page. The component itself contains the code for the window, and an application wishing to use it, will simply invoke a certain part of the component’s functionality to

\(^5\) Application Programming Interface, a set of functions that the operating system provides for a programmer.
have it shown. Again this is a plethoric mechanism, which is merely a convenience in some situations, since customisation can be done equally well by adjusting properties.

2.1.5 Summary

Components are reusable software pieces, modelled after object-oriented prescriptions, but are in contrast to objects external to their host application. This means that components may reside inside files on a local disk or somewhere in the network. Components can be thought of as the software equivalent of hardware integrated circuits, and as such the only interaction possible with these entities is through their interface, a concept known as black box use.

Their roots in the object-oriented paradigm, component models should in general address the issues brought forth by this strategy, but that is not all. The inherent distributed nature and the fact that components are external to their hosts, brings a number of extra issues into light. This chapter has identified the following concepts as being the ones, component models should address:

Language inspired concepts:

- **Objects, Classes, and Interfaces**: like traditional OO, component models should define the notion of objects, classes, methods and properties. In addition they must separate the interface part of a class, since the implementation of a component is a black box, and thus only its visible parts can be used. An object model could further specify that certain properties had constraints on their input to minimise failures due to misuse.

- **Instantiation, Initialisation and Finalisation**: a component model should have a uniform means of creating any object, since doing so is fundamental to the concept. Further, it should be possible to specify code to execute when new instances are created and destroyed.

- **Inheritance, Specialisation and Delegation**: though inheritance is a cornerstone in the object-oriented world, components can only provide this facility for interfaces. It is hard to see how specialisation as defined in OO terms can be achieved in a black box world. Instead delegation can be used, and through interface pointers, a model may choose to let components access members of an outer component.

- **Encapsulation**: since the implementation of a component is completely hidden, components expose their functionality through interfaces. Since the interface may contain only properties and methods of interest to a user, encapsulation is unavoidable.

- **Polymorphism**: the ability to treat a collection of components of the same base class in a uniform manner is a highly desirable feature and is a must for component models.

- **Composition**: at the most fundamental level of component use is the ability to group components, in order to build more specialised ones. A component model, which does not support the notion of composition is hardly useful at all.

- **Communication**: setting the properties and invoking the methods of a component should be possible in a uniform manner no matter whether the invocation is performed by another component or by an application, and regardless of the location of the component, be it local or remote.

- **Exceptions**: since errors will occur at least as frequently in components as in ordinary software, a component model should provide a standard mechanism for propagating errors back to the client.
• **Events**: having a component report back, when an interesting event occurs is a highly useable feature, which would be quite annoying to miss, and so ought to be part of the model.

• **Introspection**: learning about the intricacies of a component is a fundamental concept, without which dynamic binding would be impossible. A component should therefore either provide implementation information itself, or it should be available from a global source.

• **Concurrency**: to be robust in multithreaded environments, components should take into consideration the use of global data. Since making code thread-safe is not something a component model can handle itself, the only issue here is that it must not prevent components to work in a multithreaded environment.

• **Persistence**: writing a component instance, which has been running for some time, to a persistent media is a feature, which will allow programs to shut down without loosing data. Since components contain state that may be vital to the functionality of the overall application, persistence is something a model should provide. The three orthogonality rules should be followed to ensure language independence and full persistence.

**Data related concepts:**

• **Revision Management**: since software tends to evolve, a model must provide some way to upgrade existing components with new versions. This must be solved in a way so that old clients will continue to work, even if the interface of the component has changed. This means that a model should not allow removal of properties and methods, but only adding more of these.

• **Dynamic Replacement**: when a new version of a component is installed, running clients that use the old one should not be aware that a new component is in the system. Instead the model should incrementally shut the connections to the old component down, as the clients release their hold, and finally stuff it away. If the programmer desires, such an installation should be possible to revoke.

• **Transaction Management**: in order to synchronise the access of a global resource amongst multiple clients, a global service for grouping actions into atomic entities can be introduced.

**Distributed concepts:**

• **Portability**: in today’s networks, there will often be more than one architecture present, and so the component model should not be restricted to just one of them, just as it should not be restricted to work with one language only. Ideally components should be able to run on all architectures, and be defined and accessed from any language.

• **Activation and Execution**: components should be able to exist on any machine in a network, be accessed from any machine, and finally migrate to any machine and run there.

• **Naming and Location Transparency**: locating components in a network should be possible in an unambiguous manner, which means that they must have a unique name. A system could provide a global name server with query capabilities for locating objects with special functionality. The location of a component should not be something that the programmer should deal with, lest he has the interest.
• **Load Balancing**: when many clients use the same components in a network, replication of the one that is the bottleneck should be possible, following the redirection of some of the clients. This can be done either statically or dynamically.

• **Security**: components should have a way of restricting their usage to certain users only. This is true for the entire component as well as for its properties and methods. Furthermore, some kind of licensing mechanism will allow third party vendors to sell components, while restricting their use to licensed clients only.

**Peripheral concepts:**

• **Customisability and Graphical Interfaces**: A last aspect that a component model could deal with, but which we feel is really an extra feature, is providing a framework for building components with a graphical interface.

### 2.2 Commercial Component Models

A number of different vendors have defined component models that may actually be used in software development processes. Originating from different communities and design goals, these models range from simple ones to comprehensive specifications that will allow the definition of components of any granularity. Some are coupled with one specific language or operating system, while others have been defined with cross-platform interoperability in mind.

The most important of today’s commercial component models are CORBA and COM. In addition to these, recently Sun Microsystems have defined the Java Beans model, and according to them it classifies as a component model also. The model, which this thesis treats in-depth is Microsoft’s COM, but to clarify its strengths and weaknesses compared to the other models, this chapter will discuss CORBA and Java Beans cursorily. All three models are treated by relating them to the topics of chapter 2.1.

We have not looked at other component models than these, since a comparison is not the main issue of this thesis. Left out are among others IBM’s (D)SOM model, which serves as the foundation of a higher level specification known as OpenDoc. But since the OpenDoc project was cancelled by Apple and IBM in early 1997, this model is bound not to become the most successful one anyway [PARADIGM97].

#### 2.2.1 CORBA

The Common Object Request Broker Architecture, CORBA, is an extensive specification defined by the Object Management Group (OMG). Founded in 1989, this group has devoted itself to promote the theory and practice of object-oriented software development. It is the world’s largest software association, consisting of more than eight hundred members, including giants in the industry such as IBM, Apple, and Microsoft.

CORBA is the result of numerous ideas, brought forth by members of the organisation and compiled into an extensive specification. Available for download from OMG’s homepage, the CORBA specification comes in three volumes, embracing all of the concepts we have defined in chapter 2.1. Its more than two thousand pages address almost every issue from the core component model to high level services such as query mechanisms and trader services. The documentation for CORBA, which serves as the foundation of this chapter, can be found in [CORBA98], [CORBAS97], and [CORBAF95].
Central to CORBA is the notion of an ORB\(^6\) from which the component model lends its name. The **ORB** is the middle-man, which connects the clients of a component with its implementation. This means that a client does not have to be aware of a component’s physical location, but can simply ask the ORB for an object and get a pointer corresponding to it back, thus providing pure location transparency. This strategy is depicted in Figure 5, the arrow of which illustrates a typical client request.

The ORB can come in several flavours, depending on the need of the application programmer: some ORBs are linked directly into client and server code, others are stand-alone applications that will mediate clients and servers. A vendor may even choose to integrate an ORB with his operating system or provide its services through a runtime library.

Components are defined using a dedicated language, the **CORBA IDL**, which is short for Interface Definition Language. Originally adapted from the IDL specification of the Open Software Foundation (OSF), as specified in their X/Open specification [DCE94], the CORBA version of the language lets a programmer define the entire component structure.

The IDL language has constructs for describing modules that correspond to classes, and interfaces with attributes and methods. In CORBA a module has **one interface only**, but that interface may inherit from many ancestors, providing multiple inheritance. The properties of an interface can be readable, writeable, or both, and as usual only the members that should be programmatically available will be exposed in the interface. The components themselves are binaries, which can be implemented in **any language**, and so CORBA employs the simple strategy for **encapsulation**. In being binary pieces, CORBA components are not directly portable to other platforms, but there is no hindrance to ORBs on different systems talking to each other, and so a client running on one architecture can use components on another.

CORBA components are **instantiated** by the ORB launching the server in which they reside. When the server has been launched, it will internally create the object instance and block, telling the ORB that it is ready for incoming client requests on that particular component. If no requests are made within a specified period, the server will gracefully shut itself down again. **Initialising** and **finalising** an object can be done from the main loop of the server or within any language specific constructs of the component such as do-parts or constructors/destructors.

---

\(^6\) ORB: Object Request Broker.
The life cycle of a CORBA object is maintained in a de-coupled manner: clients can hold references to the server objects, even when the servers are no longer running. Server objects may choose to deactivate and remove themselves from memory, when no client is currently using them. CORBA objects that behaves in this manner therefore needs to support persistence.

Since the CORBA specification separates interface and implementation, and provides interface inheritance, components may be treated polymorphically, by interpreting them as instances of a particular base class. Of course a language using CORBA components will have to provide dynamic type checking itself to make this work.

To compose new components from smaller parts, a programmer can simply instantiate the sub components inside the new one and use them there. In a distributed environment, however, there will be a need to make this relationship explicit to the entire system, so that sub components are not casually removed. To this end the CORBA specification defines the Relationship Service, which will keep track of which components reference which, and the nature of the relationship: referential, contained, etc. In many ways this service resembles that of a database, which keeps the integrity of the system in place.

All components, including all information about them, are stored in a database known as the Interface Repository (IR). A system may have many of these, possibly at different locations in the network, and each should have a unique ID. All components, interfaces, structures, and so on, which are stored in an IR have a name. Names are local to the scope they are defined in, and so a module may have an interface with the same name as another module in the same IR. This scope model provides an unambiguous way of referring to CORBA entities: simply concatenate IR identifier with the container objects’ IDs, followed by the identifier of the entity in question. The IR also provides a means for clients to introspect components before using them.

A client wanting to use a CORBA component must contact the ORB, passing the correct identifier. If such an identifier was not known at compile time, a program can dynamically run through the content of the IR, thus locating components that fulfil the requirements. In addition OMG defines a service, known as the Trader Service, which is an object query mechanism, by which a programmer may ask for components that conform to certain specifications such as an interface, but also query on more human readable attributes.

The CORBA IDL specifies an identifier for tagging methods with the exceptions they may throw. Events, on the other hand, are not part of the most basic object model, but provided through the Event Service. This specifies a model of providers and consumers, where the first ones will register themselves with the service as sources of given events, and the latter register themselves as interested in notification of given events. A client can sign up for “push events”, which will interrupt its execution, and “pull events”, where the client explicitly waits for some event to occur.

The IR allows programmers to update the system with new versions of a component, by including a version ID as part of the information stored. Apart from that, the new component will have a new identifier, and in all respects be a separate entity. The old server will continue to exist alongside the new one. As a consequence of this strategy, CORBA does not define any transaction mechanism for substituting components, and no dynamic replacement.

The CORBA specification defines the Transaction Service as a means of co-ordinating multiple requests or updates on a shared resource, as atomic entities. This means that CORBA components can be designed so as not to compromise the integrity of global data.
In CORBA, components may be persistent via the **Persistent Object Service**. Through this manager, an object can have the values of its attributes, and the rest of its state written to a disk file, and subsequently reloaded to memory. The model is very open, and the developer will have to implement the most basic parts of it himself. It abides to the three principles of persistence, with the exception that components must implement a certain set of interfaces in order to gain persistence abilities.

Managing the intricacies of **concurrency** is dealt with through the Concurrency Control Service. With this part of the system, a component can mark certain data as shared, and have the manager control the locking and unlocking of the resource, thus preventing multithreading conflicts.

Another service provides a mechanism for managing **load balancing**. Known as the Life Cycle Service, this part allows a component to migrate from one machine to another, and even running instances to be moved. All this does not happen transparently, however, and the service will have to be explicitly informed of all details in the migration process.

Providing the facilities for constraining the usage of certain components is acquired through the **Security Service**. With this facility, the CORBA system provides fine grained access to objects, properties, methods, and so on, based on the identity of the person logged onto the client machine, the groups he belongs to, his location, the current time, and many more. The standard specifies that the CORBA security mechanism should be coupled with operating system security to minimise administration and number of required logins. Another service the **Licensing Service** will provide a means for COTS\(^7\) components to require the client to provide information that authenticates the use of its facilities.

In being a vast and compound specification, the CORBA manuals define how user interfaces for components should be built, describes a global network time server, additional query mechanisms and even more specific services targeted at certain business categories. All this is well and good, were it not for the fact that CORBA is merely a specification, and lack standardisation at the implementation level of inter-ORB communication. This means that there are now a number of implementations of CORBA available from vendors, but these do not inter-operate. What is more, one should not expect an implementation to supply all the services mentioned in the specification, and so a commercial CORBA ORB may not be what one had expected.

In order to remedy the problems of ORB communication, the OMG has recently defined the Internet Inter-ORB Protocol (IIOP), which provides a communication layer via TCP/IP that will enable an ORB to talk to other ORBs from different vendors. As the name suggests, this chat might well be mediated through the Internet.

### 2.2.2 Java Beans

Following the tremendous success of the Java language, Sun Microsystems decided to build a component model on top of it. This is a recent addition, and the first specification was released in December 1996. Termed Java Beans, this supposed component model is, like CORBA, also the result of suggestions from many industry leaders, again including Microsoft and IBM. In contrast to CORBA, however, Java Beans belongs to Sun only, who has compiled the suggestions into the final specification. This specification [BEANS97] along with [JNDI98], form the basis for this chapter, wherefore no further reference to either will be made.

\(^7\) COTS: Commercial Off The Shelf, pre-made software, which can be bought through an appropriate dealer.
Java Beans is tightly coupled to the Java language, and so Beans cannot be built in any other language. On the other hand, the model benefits from the “compile once, run anywhere” design principle of Java, and so is in effect platform neutral.

Inspired by constructs from a number of OO languages, including C++ and Smalltalk, the Java language embraces the classical notions of class, object, and interface. Java has single inheritance, but allows classes to implement multiple interfaces. With respect to encapsulation, there are three access specifiers for describing private, public, and protected methods, and implementations of the parent class can be accessed via the “super” keyword.

When building a Bean, these facts also apply, but since Beans will be compiled into binary files in the end, the Bean encapsulation mechanism resembles that of CORBA: only the public methods and properties can be accessed from a program using the Bean. In contrast, a Bean inheriting from another Bean, will be able to access protected members also.

In being ordinary Java classes also, Beans are constructed as any other Java object. They are activated, instantiated, and executed locally, and will run in the same address space as their host. This said, Beans can also be Java Applets, i.e. an applet may be a Bean, which means that it can run in a WWW browser, and so actually does have the potential for remote activation and local instantiation.

Java Beans are initialised in the same manner as any other Java object namely through the constructor method. Finalisation on the other hand is not part of the Java language, and thus not supported in ordinary Beans. If a Bean is an Applet, however, the “destroy” procedure may be overridden in order to specify finalisation code. This method will be executed by the browser just before the Applet is garbage collected.

Properties are implemented via “get” and “set” methods, making the Beans model follow the ADT approach. Since Beans are simple Java classes, making a class a Bean means abiding to certain naming rules, such as those of prepending property names with “get” and “set”.

The Java language is built around a runtime environment, known as the Java Virtual Machine (VM). This program is the one that takes care of managing the life times of objects, as well as the dynamic type checking, which is needed to support polymorphism. Again Beans used in Java programs benefit from this feature.

Realising that there are other languages in the world than Java, Sun provides a number of bridges, enabling Java Beans to run as ActiveX or CORBA components. When wrapped up in these contexts, Beans will expose themselves to the same restrictions as their host architecture. Wrapping a Bean as a CORBA object for example, means specifying IDL code for the interface, being restricted to interface inheritance, and all the other constraints imposed by CORBA.

The Java Beans model defines a core API for allowing applications to work with these little grey boxes. Through the Introspector class, it is possible to learn all about a component, by simply passing the component reference to a function. The result is a number of classes, which will describe properties, methods, and events. As an alternative, a Bean may itself implement the BeanInfo interface, which has methods for extracting this information also. It is worth to note that any Bean can be introspected via the Introspector class – no additional constructs are needed.

8 Java Applets: lightweight Java classes that may run in a browser, but are not stand-alone programs.
9 See chapter 3 for a very short description.
The Java language provides an exception mechanism through a “try … catch” clause, in which the code that might throw an exception is specified right after the “try” statement, and code to handle it after “catch”. Throwing exceptions can be done via the “throw” keyword. The same mechanism applies to Beans.

Java Beans are sources of events, and as such they need a mechanism to specify which events they support, and a method for clients to connect to those events. Events are specified by adding an “add” and “remove” procedure to the interface of the Bean that will be the source. The names of these two functions must be appended by the name of the event, and both take as argument an interface that derives from the “EventListener” interface. A client wishing to be notified of events will implement that interface, and invoke the “add” method of the server, passing itself as reference. When the event occurs, the server must make a sweep through all registered objects, and invoke the appropriate methods on the event interface.

The Java API defines classes for writing objects into streams on persistent storages. The Beans specification takes advantage of this mechanism to provide persistent objects. There are two ways that a Bean may be written to a storage media: either the client simply uses the Java Serialisation API, or the Bean itself supports the so called Externalisation mechanism for this purpose. In the latter case the Bean is responsible for writing all data of interest itself, as well as restoring its state later on.

Even though all Beans may be persistent, the model violates the third rule by equipping the Java language with the “transient” keyword for marking attributes of classes that should not be saved persistently. It is, however, good that it is possible to specify this kind of information, and the violation of the third rule is inevitable because of the single implementation language strategy.

The notion of a thread as the means by which concurrency is achieved in Java, makes multi-threading an issue, which the Bean developer should address. The specification bluntly states that it is the responsibility of the programmer to make sure his Beans can function correctly in such an environment, and suggests that simple Beans should mark all their methods as “synchronized”. This will make them function as atomic entities, of which there can be at most one controlling thread. This is of course deeply inefficient since such a strategy makes multiple clients wait for an entire procedure to terminate where they could have been waiting somewhere inside that procedure after having executed a part of it.

An addition to the Java Beans specification, the Java Naming and Directory Interface (JNDI), adds a global name server to a network of Beans and other resources. The JNDI has methods by which it can be asked to locate a given resource and pass it back to the caller. It is worth to note that this service can embrace any resource within a network, which is achieved by simply letting the JNDI function as a common name space interface, under which other servers perform. The JNDI, which was released as of January 1998, does not support any predefined mechanism for querying for Beans, but could be augmented to embrace this, also.

The intention of the Beans specification was to provide a model for light-weight components, which would be reusable across applications. As such Sun decided that a graphical interface would be something many Beans would have, and so has added rich support for this specific feature. The full Java AWT\(^{10}\) may be used to define the user interface, and the Beans API also provides the notion of

---

\(^{10}\) AWT: Abstract Window Toolkit, a library of useful graphical interface classes.
property-pages to let users modify properties visually. This latter mechanism allows a graphical way to change the internals of Beans at design-time.

Even though Java was made for building applications that might very well be distributed in nature, the Beans specification does not address issues such as revision management and dynamic replacement. In the same spirit it largely ignores global naming, location transparency and load balancing. Finally the Java Beans technology does not concern itself with the security issues of user access; in fact the only security issues pertaining to Beans are those that pertain to the Java Language.

Compared to CORBA, Java Beans exposes itself as a toy component model, which is best used in conjunction with Java programs, and preferably on a single machine. Classifying it as a component model is possibly even wrong, since the difference between Beans and ordinary OO objects is diminishing: they are bound to a single language, and can thus not be directly called from clients that were not written in Java. We feel that a component model should at least be language neutral.

Of course a Bean may be wrapped up in, say, ActiveX or CORBA objects, but the overhead is significant, and relying on the facilities of somebody else’s component model to provide crucial features hardly qualifies.

A positive thing to say about Java Beans is that when created as Applets, they may well play a dominant role in the future, when it comes to building applications that run inside WWW browsers, because of the inherent platform neutrality of Java.
Microsoft’s model for building component based software has – very appropriately – been coined COM for “Component Object Model” [COMSPEC95]. This term, however, has been underway for a long time, and in that period other terms dominated the stage. To understand what COM is and how it relates to the myriad of other buzzwords that inhabit the Microsoft component world, we must start with DDE [DCOM97].

In the heydays of ancient Windows 3.0 and its sixteen bit applications there was a technology named DDE for “Dynamic Data Exchange”. This was a primitive protocol for exchanging data between a client and a server application, which used the underlying message system of the Windows platform as its means of communication. When a DDE server would make known to the environment that it was online and ready to serve, it would call an API function and then sit tight and wait for incoming client requests. A client needing a service would broadcast a message to all programs, asking for a specific server and a specific service, which the server would then pick up and respond to. Windows messages are not very suitable for storing large quantities of data, and so the DDE protocol would shamelessly exploit the Windows clipboard for this purpose.

Programming DDE in these dark ages was a tough task, but the introduction of the DDE Management Library (DDEML) solved some of the complications. It was still, however, difficult to make scenarios with multiple clients and servers, and the overall stability of the product was not an issue one would discuss with the employee in charge of the corporate finances [COM97].

On this solid ground OLE was born, originally to mean “Object Linking and Embedding” [OLE95]. OLE 1 used DDE as its underlying communication mechanism to provide a means for embedding one application in another and linking to an application that would know how to handle documents of a specific type from another. This was basically it.

OLE 2 [OLE95] was originally planned as an improvement of OLE 1, but grew to be much more and involved an entire redefinition of the grounds that the technology was built upon. DDE was no longer used as a basis, and was reduced to that dubious corner of the Windows platform in which it still resides for compatibility reasons. Instead a new actor was introduced to the stage, namely COM.

At this time COM and OLE 2 were rather mixed up, because OLE 2 was in fact the only technology that built upon the foundations that COM would provide. However, as time grew by, the 2 concepts were disjoined and are now regarded as separate technologies.

OLE 2 embraces a lot of different terms, two of which are still linking and embedding that, along with In-place activation, now comprise the notion of “OLE Documents”. In addition OLE 2 defines “Structured Storage” for generic file manipulation and the basis of persistent objects, “Uniform Data Transfer” which relates to clipboard functionality and drag-and-drop capabilities. Of other quite touted OLE 2 features are “OLE Automation” that lets a program expose a set of commands for control by other applications, and “Monikers” that equips files with intelligence of how they can be displayed, edited, etc. [OLE95].

All these different features of OLE 2 are built on a simple common ground, viz. COM. The basic concepts of COM are interfaces and classes, usually known as CoClasses. Interfaces are in fact just abstract collections of function headers, while the CoClasses contain the actual implementations. All COM interfaces inherit from a common ancestor, IUnknown, which introduces basic reference
counting and interface retrieval capabilities. With this in mind, OLE 2 is nothing more than a huge collection of interfaces and their corresponding implementations.

OLE 1 was created back in 1991, and redefined and split into COM and OLE 2 in 1993. Soon after the marriage of OLE 2 and COM, it became clear that there would never be an OLE 3, since the COM foundation would allow OLE 2 to develop without the need for a redefinition. Today OLE 2 is therefore simply referred to as OLE [OLE95].

The notion of “OLE Controls” was coined at the introduction of OLE 2. An OLE Control is a collection of functionality and content that can be accessed from the outside. Such a control can be manipulated programmatically by other programs that can use the control’s functionality to perform specific tasks. This may not appear to be any different from COM objects, and the only difference is just the level of complexity. For something to be an OLE Control, it will need mechanisms for displaying itself and be manipulated visually. This capability can be utilised in a development environment to display the component, even if it has no visible interface at runtime. Recently some smart marketing executive came up with the term “ActiveX” for controls with these abilities, but that was really nothing more than old wine on new bottles. [BUZZ98].

To sum up: OLE and ActiveX build on COM. OLE is a collection of utilities and functionality, while ActiveX is a short hand for OLE Controls, components with a visual interface.

### 3.1 The Component Object Model

In being a model for creating object oriented software components, COM lends itself to an investigation through the concepts identified in chapter 2.1. This chapter then seeks to clarify how COM relates to component models in general, by describing it through the identified terms. The aim of this chapter is not to fit COM into some tight component model, but rather to describe its extents, while clarifying how it relates to the concepts that we find, a component model should address.

#### 3.1.1 Classes, Objects and Interfaces

COM defines the traditional OO notion of a class, which represents a description of objects of a given type. The objects represent the actual data that can be manipulated, and every object is said to belong to a class. In COM terms objects are known as components or simply as objects. This motivates the fact that COM classes are called CoClasses for “Component Classes”.

Since components are external to their host application, they will need some container, in which they can reside. In COM such a container is known as a library, because there can be more than one class of components within a library. Libraries can be standalone executables or binary collections, known as Dynamic Link Libraries or DLLs [OLE95]. Sometimes libraries are referred to as object servers, a term we will also employ.

At the core of the model is the notion of an interface, which serves as the contract between the component and its client. The deal is that the component implements the methods that make up the interface in question, and that the client abides to the conventions imposed by the interface declaration.

As mentioned in [OIOOP95] and section 2.1.1.1, many object oriented programming languages have the feature of separating interface and implementation. The advantage is obviously that a user of an object is freed from worrying about the code that fills out the methods of the object, and can focus on how to connect to it instead. In COM an interface can only have methods, which means that the ADT approach has been employed.
Interfaces are defined using a special language known as **IDL** for Interface Definition Language. As mentioned in the discussion of CORBA in section 2.2.1, the X/Open group in their Distributed Computing Environment (DCE) specification [DCE94] originally defined IDL, which is an open standard.

Just like OMG, Microsoft then took the IDL specification and altered it to their needs, maintaining the IDL name. This means that there is now a number of incompatible IDL languages, a fact which may easily confuse the unaware.

The Microsoft IDL was originally used for specifying how remote procedure calls (RPC) would pack and unpack their parameters in the Microsoft RPC model, but nowadays it is also used for describing COM components. This is no coincidence, since the distributed version of COM, DCOM, uses RPC to remote method invocations of interface member functions between programs.

COM interfaces support single inheritance, which means that sub-classed interfaces will inherit the methods of their ancestor. The mother of all interfaces is known as **IUnknown**, and is the one every COM interface has as its top most ancestor. IUnknown has three functions, viz. QueryInterface, AddRef, and Release. Figure 6 lists the IDL declaration for the IUnknown interface.

Using the QueryInterface method, a client can ask the component whether it supports a specific interface, and if so, get a pointer to it. As can be seen in Figure 6, QueryInterface takes two arguments, a GUID and a pointer that will point to the interface in question afterwards. Note that the ppvObj parameter that will hold this pointer is qualified as void** instead of IUnknown*, as would be expected. We have not been able to figure out why, but the implication is that one must know that it does in fact reference an IUnknown interface, since there is no way to determine the qualification of a void pointer.

A **GUID** as defined in [DCE94] is a 128 bit long identifier, which uniquely names the interface.

```
[odl,
 uuid(00000000-0000-0000-c000-000000000046)
]
interface IUnknown {
    [restricted]
    HRESULT _cdecl QueryInterface(
        [in] GUID* riid,
        [out] void** ppvObj);
    [restricted]
    unsigned long _cdecl AddRef();
    [restricted]
    unsigned long _cdecl Release();
};
```

**Figure 6: IDL declaration for IUnknown**

GUID is a short hand notation for “Globally Unique IDentifier”, a feature which such a number is guaranteed to possess. No two GUIDs in the world are alike, a bold presumption, which is guaranteed by the fact that such properties as the net card ID of the computer, which happens to be unique, the time, and the location are ingredients in the GUID generation algorithm. On the Windows system, a new GUID can be generated by invoking the API function “CoCreateGUID”. GUIDs are also known as UUIDs for Universally Unique IDentifiers [DCE94]. In COM, libraries, CoClasses, and interfaces are equipped with GUIDs.
In the realisation that GUIDs can be quite hard to remember, Microsoft has given components yet another identifier, the **ProgID** [DCOM97]. This identifier has the advantage of being readable by humans, but is unfortunately not necessarily unique. If the CoClass implements more than one interface, requesting an interface pointer through the ProgID will yield the default interface of that class. If other interfaces than the default interface will be needed, QueryInterface must be used again. A ProgID could for instance be “InternetExplorer.Application”, which names the COM interface of Microsoft’s web-browser.

**Figure 7: Structure of a COM Library**

COM interfaces are **immutable**, which means that once defined, they will never change. Therefore old components will continue to work, even if the original interface they utilise is revised, since in doing so a new GUID will have to be attached. An effect of this strategy for evolving the functionality of interfaces is that one may end up with many interfaces, which are somehow related, but are not connected in any way, since COM does not allow grouping of interfaces.

Interfaces are usually depicted by a rectangle with a lollipop sticking out of it, as illustrated on Figure 7 [OLE95].

```c
HRESULT __stdcall MyCoClass::QueryInterface(const IID& iid, void** ppv) {
    if (iid == IID_IUnknown) {
        *ppv = static_cast<IUnknown*>(static_cast<IX*>(this));
    } else if (iid == IID_IX) {
        *ppv = static_cast<IX*>(this);
    } else if (iid == IID_IY) {
        *ppv = static_cast<IY*>(this);
    } else {
        *ppv = NULL;
        return E_NOINTERFACE;
    } (*ppv)->AddRef();
    return S_OK;
}
```

**Figure 8: QueryInterface implementation**

Drawing the interface inside the class shows the fact that a CoClass implements an interface. The lollipops that stick out of the CoClasses indicate the only way a client can connect to instances of
the class. In general, entities that belong to some other entity are drawn inside the owner, as Figure 7 illustrates by having the CoClasses in the library box.

QueryInterface is quite simple to implement since the sources of two different CoClasses only differ with respect to the GUIDs that represent the interfaces they support. Figure 8, which is inspired by [COM97], lists a standard implementation of this function, written in C++.

Evidently QueryInterface is merely a big conditional, which tests the requested interface ID with the ones that this CoClass supports. When a match is found, the instance of the class is typecast into the appropriate interface and returned to the user through the pvp variable. Note that if no matching interface is found, the function will return the COM error-code E_NOINTERFACE and pass a NULL pointer in pvp.

The “static_cast” imperative will make pvp point to different addresses depending on which interface was requested. The reason for this is that interface members are laid out in a table known as the V-Table for Virtual function Table [COM97]. A client will know the layout of an interface’s V-Table through its IDL description, and can now invoke functions by using the V-Table as an indirection mechanism.

![Diagram of VTable layout for a component with two interfaces, IX and IY](image)

**Figure 9: VTable layout for a component with two interfaces, IX and IY**

The V-Table layout of the component with interfaces IX and IY as described in Figure 8 is shown on Figure 9. Notice how the client uses its interface pointer to invoke the methods of IX by utilising the V-Table as a jump table.

Turning back to Figure 8, notice how the IUnknown pointer is determined by first casting “this” to IX and then IX to IUnknown. Since all COM interfaces inherit from IUnknown, they will all have QueryInterface, AddRef, and Release as their first three member functions. Instead of providing a separate IUnknown implementation, a server may thus choose to use the implementation from one of its interfaces. In the example IX could have been substituted with IY.

V-Tables are discussed in further detail in section 3.3.1.1, where it is contrasted with an alternative.

Interface methods are modelled using C [CPL88] style function declarations; the only visible difference is the fact that interface methods can also contain the optional attributes IN, OUT, and RETVAL, which will be discussed in section 3.1.6.3. Furthermore they should always return the
special parameter HRESULT. This type is in fact an integer, which may return a fixed number of values, indicating whether or not the function call was successful, and if not, what went wrong.

### 3.1.2 Instantiation, Initialisation and Finalisation

COM does not have constructors on CoClasses, but will use global API functions for the purpose of instantiation new objects. Similarly there is no notion of a destructor method in COM; instead it uses explicit reference counting for controlling the lifetime of a component.

To create a new instance of a COM class, a client must know the GUID or the ProgID of the CoClass, which implements the interfaces that the client will be using. This GUID, usually known as **CLSID** for Class IDentifier [COM97], is passed on to the API function **CoCreateInstance** along with the GUID of the interface in question. This function will then make sure that a new object is created and return a pointer to the requested interface.

When a client plans to use multiple instances of the same class, it can ask for a CoClass’s **ClassFactory** [OLE95], which is a specific component that resides in the same object server as the CoClass implementation. The ClassFactory contains a method CreateInstance, which doesn’t need the CLSID, because it is specific to that class and can only create instances of that type. To get the ClassFactory, a client would call **CoGetClassObject** and would in return be given an IClassFactory interface pointer. **CoCreateInstance** uses **CoGetClassObject** to create a single instance of a given class, so calling this function directly is simply more efficient when the client needs more than one instance.

When an object server is created, an independent component, hosting an implementation of the IClassFactory interface must be provided for each object in the library. This is a tedious task, since such factories will almost always be identical in functionality: they will simply allocate a new instance of the object they serve, and pass back the pointer to it, cast as the interface that the client requested. Some languages like Delphi and Microsoft’s Visual Basic (VB) provide a standard implementation of IClassFactory that may be used.

It is worth to note that since the ClassFactory is so tightly coupled to the component, which it services, it will have direct access to the internals of that component, only restricted by the implementation language encapsulation mechanisms. This means that the CreateInstance method is an obvious candidate for initialising the public members of the object.

Behind the scene, **CoGetClassObject** invokes the **Service Control Manager (SCM)** [DCOM97]. This is a special daemon server, which is always running, and will lookup and load libraries upon request. When the object server has been started, the SCM invokes the member function **DllGetClassObject**, which is always present in a COM object server. This method will make an instance of the ClassFactory for the requested class, which will in turn be passed back to the client via **CoGetClassObject**.

The steps taken by the SCM to get the ClassFactory depends on the object server’s type, which can be in-process, out-of-process or remote. If the server is in-process or out-of-process, the SCM locates the file in the local file system and loads it. The remote case is somewhat more involved, and will be treated in connection with the discussion of DCOM in chapter 3.2.

In Microsoft Windows, **in-process** servers are implemented as DLLs, meaning that they will be executed in the client’s process space, which is faster than running them in their own, separate process space. In contrast, an **out-of-process** server is an ordinary executable, running in a separate process space.
To ensure that an object stays in memory, the client must call the AddRef member function of the IUnknown interface of the component it has obtained a reference to. This must be done as the first thing after the pointer has been returned from the ClassFactory, since the SCM periodically invokes the “DllCanUnloadNow” function of all running in-proc object servers. Similarly a client must call Release, when it has finished working with the interfaces of a component.

By default AddRef and Release do little more than increment and decrement a local reference counter and return its new value. Release will additionally destroy the running instance of the CoClass, when the counter drops to zero.

Sometimes, however, finalisation of the component may be needed, and Release is an obvious candidate for doing this. Consider a component, which keeps a connection to a database. When first instanced a connection must be made to the database, and this could be performed in the CreateInstance method of the ClassFactory, e.g. by invoking a method on the new instance. Similarly the component could close the connection in Release, when the counter drops to zero.

The thirty two bit versions of the Windows operating system are multi-threaded, and so components should be constructed with threading in mind. Since COM is so tightly coupled to Windows, the threading model of COM is merely that of the operating system. For some odd reason, Microsoft has chosen to rename some of the different ways threading can work, but in essence it is the same [COM97].

A component, which will be used concurrently by multiple threads, must either encapsulate all method calls in guarded regions that ensure mutual exclusiveness or make sure that all methods are thread-safe themselves. The standard functions for IUnknown are no exception to this rule. On the Windows platform there exists two API functions, which can be used instead of the ++ and – operators to make the implementation thread safe.

A standard implementation of AddRef and Release, which happens not to be thread safe, is shown in Figure 10.

```
ULONG __stdcall MyCoClass::AddRef()
{
    return ++m_cRef;
}

ULONG __stdcall MyCoClass::Release()
{
    if (--m_cRef == 0) {
        delete this;
        return 0;
    }
    return m_cRef;
}
```

**Figure 10: default implementation of AddRef and Release**

Unlike traditional Object Oriented frameworks COM does not allow implementation inheritance. This, of course, is tightly coupled to the fact that a CoClass implementation is only available as a binary, rather than as a human readable text. Interfaces may be inherited, but the actual implementations of existing CoClasses are beyond control. As mentioned in section 2.1.1.3, this problem relates to the black box situation that components face in general, and so is not specific to
COM. In order to remedy this problem, Microsoft has incorporated the notion of aggregated and contained CoClasses into the COM model.

**Containment** in COM terms is really nothing more than delegation as discussed in section 2.1.1.3, a notion which is also used in the COM literature. Traditionally the term **aggregation** has been used somewhat interchangeably with the term containment in the OO literature, to mean that an object is composed of smaller objects. COM uses this term in a somewhat different manner, namely to mean that an object gets a reference to its container, also discussed in section 2.1.1.3.

Whenever a new CoClass implementation is created, the facilities of existing components can be used as building blocks. Let’s say a programmer is writing code for a CoClass which implements an interface B that he has created by inheriting from some other interface A, different from IUnknown.

On the system there is a library which contains a CoClass implementation for interface A. Interface B has only one extra function, and the developer is therefore keen on using the implementation that already exists on his system for the inherited methods. He therefore instantiates an object that implements interface A, in the initialisation section of the component, and keeps the reference in a global variable. Interface A will then be available to receive invocations of its member functions throughout the lifetime of the new object. The new component can therefore be regarded as a client of the other object. All the inherited methods of new interface can simply delegate the calls to the interface on the other object, and in effect have the same implementation. This scenario is known as **containment** in COM terms [COM97].

```pascal
function TCoNew.Power(v1: Integer; out v2: Integer): HResult;
begin
  Result := CoOld.Power(v1, v2);
end;
```

**Figure 11: Contained method call**

An example of an implementation of a method, which delegates its call to a contained CoClass is shown on Figure 11. As can be seen, the new implementation of Power does nothing, but invoke the same function on the object CoOld, which was in turn instantiated, when CoNew was created. This scenario resembles the Smalltalk “super” construct, and can be used in the exact same way, which means that custom code can be added to the implementation, and the inherited method called when appropriate. The code on the figure was in fact auto-generated by the CASE tool Idun, which will be described in greater detail in chapter 5.2.

On occasion it might be desirable to implement the exact same interface that has previously been defined in some other object server. This could for instance be when a number of clients for a component already exist, but the implementation of a few of its methods are too slow. A new component server with a new CoClass could then be created, the CoClass implementing the exact same interface whose implementation it wants to improve on. The methods that were effectively implemented in the old server could then be delegated, and the ineffective ones reimplemented.

In some situations it may be preferable to have a new CoClass implement the exact same interface as another one, but without changing the implementation. This could be convenient in connection with modelling aspects, where a programmer wants to group related functionality in one server, for example. Containment could be used for this purpose, but the extra level of indirection may seem

---

COM. In order to remedy this problem, Microsoft has incorporated the notion of aggregated and contained CoClasses into the COM model.

**Containment** in COM terms is really nothing more than delegation as discussed in section 2.1.1.3, a notion which is also used in the COM literature. Traditionally the term **aggregation** has been used somewhat interchangeably with the term containment in the OO literature, to mean that an object is composed of smaller objects. COM uses this term in a somewhat different manner, namely to mean that an object gets a reference to its container, also discussed in section 2.1.1.3.

Whenever a new CoClass implementation is created, the facilities of existing components can be used as building blocks. Let’s say a programmer is writing code for a CoClass which implements an interface B that he has created by inheriting from some other interface A, different from IUnknown.

On the system there is a library which contains a CoClass implementation for interface A. Interface B has only one extra function, and the developer is therefore keen on using the implementation that already exists on his system for the inherited methods. He therefore instantiates an object that implements interface A, in the initialisation section of the component, and keeps the reference in a global variable. Interface A will then be available to receive invocations of its member functions throughout the lifetime of the new object. The new component can therefore be regarded as a client of the other object. All the inherited methods of new interface can simply delegate the calls to the interface on the other object, and in effect have the same implementation. This scenario is known as **containment** in COM terms [COM97].

```pascal
function TCoNew.Power(v1: Integer; out v2: Integer): HResult;
begin
  Result := CoOld.Power(v1, v2);
end;
```

**Figure 11: Contained method call**

An example of an implementation of a method, which delegates its call to a contained CoClass is shown on Figure 11. As can be seen, the new implementation of Power does nothing, but invoke the same function on the object CoOld, which was in turn instantiated, when CoNew was created. This scenario resembles the Smalltalk “super” construct, and can be used in the exact same way, which means that custom code can be added to the implementation, and the inherited method called when appropriate. The code on the figure was in fact auto-generated by the CASE tool Idun, which will be described in greater detail in chapter 5.2.

On occasion it might be desirable to implement the exact same interface that has previously been defined in some other object server. This could for instance be when a number of clients for a component already exist, but the implementation of a few of its methods are too slow. A new component server with a new CoClass could then be created, the CoClass implementing the exact same interface whose implementation it wants to improve on. The methods that were effectively implemented in the old server could then be delegated, and the ineffective ones reimplemented.

In some situations it may be preferable to have a new CoClass implement the exact same interface as another one, but without changing the implementation. This could be convenient in connection with modelling aspects, where a programmer wants to group related functionality in one server, for example. Containment could be used for this purpose, but the extra level of indirection may seem

---

11 Computer Aided Software Engineering: an acronym for graphical tools that aid a programmer in designing his program model. See for instance [BETA93].
like an unnecessary overhead, when the client of the component might as well have asked for the interface of the contained object directly.

To avoid the overhead in such situations, COM defines the notion of aggregated interfaces [COMSPEC95]. These are, in contrast to contained interfaces, exposed directly to the client, while maintaining the illusion that they are in fact implemented by another component. But this introduces a problem, namely the fact that any interfaces, which an object implements can be obtained by invoking QueryInterface. Since the aggregated object could impossibly have known how it would be used, when it was defined, it could not have foreseen the situation, and so would claim that it knew nothing of the outer component’s other interfaces, when asked. This is circumvented by having CoGetClassObject pass a pointer to the controlling object’s IUnknown interface, if any, and NULL, otherwise. This little pointer is the solution to the problem, as we shall see.

Whenever a component aggregates another one, it is referred to as the outer of the aggregate, while the aggregated component is known as the inner of the aggregate. An inner object must be implemented with support for aggregation in mind, so that it can take special action, whenever the pointer to the outer IUnknown is non-NULL. Additionally, the outer IUnknown needs a slightly different implementation of QueryInterface than that of Figure 8.

![Diagram](image)

**Figure 12: Requesting an inner interface**

The standard trick is to equip the inner with two versions of IUnknown, viz. the delegating and the nondelegating unknown\(^{12}\). The delegating unknown will simply forward all QueryInterface requests to the controlling IUnknown, while the QueryInterface of the nondelegating will actually work as that of Figure 8.

When the outer object first calls CoCreateInstance or the CreateInstance method of the inner object’s ClassFactory, passing its own IUnknown pointer to the function, it gets a pointer to the nondelegating unknown in return. This pointer will be used, when the client asks the outer IUnknown for an interface that it does not support itself. The scenario is shown in Figure 12, where the arrow at 1 indicates that the client requests the IMyInner interface. Immediately, the outer object senses that this call is for the inner component and forwards the call to the nondelegating unknown, to which it has previously obtained a pointer. Finally, at 3, a pointer to the interface is returned.

\(^{12}\) For a detailed discussion of this method, see [COM97].
The pointer to IMyInner could now be used to request the IMyOuter interface of the outer object, because all interfaces inherit from IUnknown. This is illustrated on Figure 13. The method call at 1 is the client asking for the IMyOuter interface on the inner object’s IMyInner interface, which uses the delegating implementation of IUnknown, as shown at 2. At 3 the call is delegated to the outer object, and finally at 4 the IMyOuter interface pointer is returned. Even if the interface requested by the client were implemented by the inner object, the call would still delegate to the controlling unknown and then be forwarded to the nondelegating unknown of the inner object, before the interface pointer could be returned.

If an object, which has support for aggregation, is not part of an aggregate when it is instantiated, it will simply delegate all calls from the delegating unknown to its own nondelegating unknown, thus effectively providing the same functionality as that of Figure 8.

Since an aggregate is to be considered a single object rather than two or more, the inner components may only live as long as the outer does. To facilitate this, the AddRef and Release functions of the inner objects will also delegate their method calls to the controlling unknown, and the inner object can thus not be de-referenced without the outer being it too.

An important point of aggregation is that the pointer of the outer object’s IUnknown interface can be used by the inner object to access functionality in its container. This is similar to the way an object in the SELF language uses its “Self” pointer to access properties in the object that called it.

Microsoft’s commercial Java interpreter and compiler, J++, uses aggregation and containment to simulate implementation inheritance. If the programmer specifies that he wants to inherit an interface, but does not override the old implementations, J++ will interpret this as an aggregate, and the old interface will be directly exposed. On the other hand, when a user changes one of the inherited functions, J++ will instead employ containment to simulate the implementation inheritance [JAVA95].

Containment and aggregation may also be used as means for composing new objects from smaller parts, a concept known as composition [BETA93]. In the case of containment, it is straightforward to have local instance variables that hold pointers to part objects, but when we move to aggregation, the trouble begins.

Consider a component, which uses aggregation to expose one of two interfaces of a part object. The outer component is not interested in showing the other interface, since it plans on using this only for its own private purposes. Unfortunately it is impossible to hide the second interface from an

Figure 13: Requesting an outer interface from an inner object
officious client, since this can just call QueryInterface to get a pointer to the second interface. The composition problem of aggregated components is discussed in further detail in [CPMCOM97].

### 3.1.4 Polymorphism and Encapsulation

As described in section 2.1.1.5, in standard OO terms, polymorphism is the ability of objects of the same base class to be treated in the same manner. The base class could for example define common behaviour like a draw method, and implement different versions of that function in the subclasses. In a language with support for **polymorphic** functions, one that took a reference to an object of the base class, and then used this function with all descendants of that class, could be defined.

The COM framework has inherent support for polymorphism [DCOM97]. This is achieved by allowing functions to pass pointers to arbitrary interfaces, and by letting different objects implement the same interfaces. The COM notion of allowing a class to implement multiple interfaces comes in handy here, because it lets objects with some common behaviour be treated polymorphically, even if it is only a fragment of their entire functionality, which is similar.

![Diagram of components](image)

**Figure 14: Two components implementing the same common interface**

Polymorphism can also be achieved in an environment, which only allows one interface for each class. COM’s competitor CORBA, for example, employs multiple inheritance to gain the flexibility, which COM exposes by having multiple interfaces per class. Multiple inheritance introduces the problem of name clashing, when two ancestors of a subclass contain a member with the same name. It is worth to note that COM’s single inheritance with multiple interfaces model does not have this problem.

```delphi
Procedure Foo(Obj: ICommon);
Begin
  Obj.draw;
End;
```

```delphi
... Begin
  Foo(ComponentA as ICommon);
  Foo(ComponentB as ICommon);
End;
```

**Figure 15: Polymorphic invocation of Foo**

Figure 14 shows to components, which implement a common interface with a function, “draw”. The CoClasses “ComponentA” and “ComponentB” can now be used polymorphically, as illustrated on Figure 15. In this figure, some client has instantiated the two classes and now holds a pointer to concrete instances. The client’s function Foo, which takes a reference that is qualified by the common interface can then be invoked on both. The explicit cast to ICommon in both invocations is Delphi’s implicit syntax for “QueryInterface”.
Another OO term, which deserves treatment in connection with COM is encapsulation [BETA93]. This is also often referred to as information hiding, because encapsulation means hiding the details of the implementation from the user. As described in section 2.1.1.4, a language will often have three access specifiers, for marking members public, private or protected.

COM does not have any access specifiers, but every interface a CoClass implements is considered public. A programmer can have all the private members that he wants, and since the only way a component can be accessed is through method calls on the interfaces, no external client will ever get in direct contact with its internals.

3.1.5 Object descriptions and Introspection

The IDL language, which is used for describing COM components, has quite a few keywords, of which the first and foremost is “library” that represents the entire object server. Every keyword that represents some kind of object has a set of braces that defines the scope of the object. As can be seen on Figure 16, inside a library there may be interface and CoClass declarations.

An ordinary COM interface simply has methods, but when moving to automation interfaces, as will be discussed in section 3.3.1, an interface will also be allowed to have properties. A CoClass simply has a list of the interfaces that it implements. Figure 16 shows the IDL source for most of the object hierarchy, which is depicted in Figure 7. The italicised text denotes non-syntactic elements, and the rest is actual IDL syntax.

Every object in the IDL language has an attributes section, in which information like version, helptext, and GUID belongs. This section, syntactically represented by square brackets, precedes the actual declaration of the object.

```
[Library Attributes]
library Q {

[Interface Attributes]
interface X: IUnknown {

  methods
};

[Interface Attributes]
interface Y: IUnknown {

  methods
};

[CoClass Attributes]
coclass A {

  [default] interface X;

  interface Y;
}

);}
```

**Figure 16: Basic IDL file structure**

The IDL language defines a large number of primitive types: integer, long, boolean, double, etc. All of these can also be passed by reference, rather than by value. Microsoft IDL defines a subset of the types for use in their automation concept. These types are known as automation types for this reason. Automation will be treated in depth in section 3.3.1.
Apart from the object types described above, Microsoft IDL will allow the definition of structures and discriminated unions in a C-like manner. Enumerated constants can be created via the enum keyword and custom types through the typedef keyword.

Microsoft has created an IDL compiler, **MIDL**, which will produce a binary equivalent of an IDL file. Such a binary is usually referred to as a type library or a TLB file. In contrast to the original IDL file, TLB binaries are tightly coupled to the target machine they were built for. This means that it is not possible to use a Win32 TLB file on a Macintosh or a Sun Solaris, but the original IDL file must be recompiled.

The thirty two bit Windows platforms have a special system database, known as the **Registry**, from which information about every interesting, configurable detail of the running installation can be accessed. Whenever a new type library is installed on the machine, it ends up in this database. When a user wants to access such information, he can use the API functions LoadTypeLib and LoadTypeLibEx to get a pointer to an interface, which represents the library object of the TLB file. From here he can access all types and class declarations in the library, again via subsequent interfaces. The CASE tool **Idun**, which is described in chapter 5.2, uses this technique to introspect component capabilities for drawing a graphical representation of the content within a TLB file.

COM provides a pseudo mechanism for versioning components via the Registry. As a sub-key under the entry for the type library, a programmer may specify which version of the CoClass it contains. Different versions will be registered under their own type library entry, and the only thing that connects them are the fact that they implement the same interface.

When a component is to be instantiated, there is no way of telling the system which version should be started. The programmer has to browse through the Registry, locate all libraries that contain a CoClass implementation for the interface in question, and finally get the GUID for the desired version. In effect, this kind of versioning requires that all COM programmers across the world abide to the same conventions, and since this is unlikely, and because it is so hard to find out which versions of a component exist on a given system, this scheme is next to useless.

### 3.1.6 Method invocation

When a programmer writes code in his favourite OO language, he will not have to worry about how the compiler translates method calls into assembler code. But when moving to a component-centric world, method calls will suddenly become an issue that he will have to deal with.

On the Windows platform, each new executable, or EXE, is given its own private, virtual address space, beginning at zero. When a program requests a component, which resides in a DLL object server, that DLL is launched and mapped into the program’s private address space. This means that the program and the component can directly share the same data, but also that any hostile behaviour on the part of the component may lead to the death of the entire program. Apart from this, as long as programs are restricted to using only components in DLLs, method invocation is not an issue to worry about. The problem arises, when accessing components that live in another executable.

Consider a scenario, in which some developer has created an instance of a class that resides in an EXE file. The SCM has launched the EXE, and he has obtained a pointer to the interface he asked for. The developer now wants to invoke a method on that interface, passing a pointer to a data structure as parameter. The data structure is located at address 2000 in his client program’s private address space, and this number is then given to the member function of the interface in question.
This scheme would not work, because the component would look for the structure at address 2000 in its own private address space, and not find it there. This means that some mechanism is needed to remedy the problem of passing data across process boundaries. In COM terms this process of copying data between different processes, is known as **marshaling** [COM97].

COM offers three different ways of marshaling parameters across process or network boundaries, namely automation/type library, standard, and custom, listed here in order of growing complexity from the developer’s point of view. Alas, the simplest to implement is also the least flexible, as we shall see.

This section uses a number of example applications to illustrate the differences between the possible ways of implementing marshaling. The example applications, which can be found in their entirety in the appendices, use type library and standard marshaling, respectively, to pass a data structure across a network. Since custom marshaling burdens the developer enormously [COM97], examples for this kind of parameter passing have been omitted.

3.1.6.1 **The Proxy and Stub Components**

In the case of the server object being implemented as in-process, communication with the client is fairly trivial: the client’s invocations of server functions are simply translated into ordinary procedure calls through some indirection mechanism, usually a V-Table.

Procedure calls on out-of-process servers are somewhat more involved. These use the Microsoft version of **DCE RPC** [MSDN97] to bundle the parameters, send them across the process or network boundary and de-bundle them again on the receiving side. This marshaling process is entirely transparent to the client, which only knows of the interface and the functions it can call on it.

In order to obtain this kind of transparency in the out-of-process case, special code for packing up and restoring the parameters is needed. On the server side, a COM component, usually known as a stub is therefore needed. The stub will receive RPC packets from another process or the network and translate them into actual procedure calls, which it will then invoke on the running server. Any response from the server will be sent back to the client using the same RPC mechanism.

![Diagram of out-of-process communication via Proxy and Stub](image)

**Figure 17: Out-of-process communication via Proxy and Stub**

On the client side another COM component, known as the proxy, will translate procedure calls on the interface into RPC packets and ship them across process or network boundaries to the server stub. Furthermore the proxy will translate any response from the server into the correct data type or structure. Being largely identical in functionality, the proxy and stub will often reside in the same
DLL, henceforth referred to as the proxy/stub DLL. Notice that proxy and stub components must always live in an in-proc server, since building an out-of-proc server for this purpose gives little meaning. Doing so would give rise to the old hen and egg problem: who marshals the marshaler?

The out-of-process scenario is illustrated on Figure 17, where a client connects to a server, which resides in another process space, either on the same machine or on another one in the network. The client sees the same interfaces on the proxy, as the server exposes, a fact that is illustrated by the proxy having two interfaces like the object server.

COM allows pointers to structures and simple data types as parameters to interface methods. In the in-process case this is not a problem, but when parameters have to be marshaled, a pointer structure involves keeping different instances of the same data synchronised. When a pointer is marshaled, a deep copy of the data pointed to is transferred. The scenario resembles that of passing parameters by value, in which case the value is simply copied and sent.

3.1.6.2 Type Library/Automation Marshaling

An easy way to get past the headaches of implementing marshaling is by letting the operating system handle it. This can be done by restricting the types on methods to a certain subset, known as the automation types. Automation is an issue, which will be treated in further detail in chapter 3.3.1.

With this scheme it is not necessary to supply the proxy/stub DLL, because every Windows NT 4.0 and Windows 95 contain the DLL Oleaut32.dll that will act as proxy/stub when marshaling data to and from a server that uses automation. This kind of marshaling is known as automation marshaling or type library marshaling [OLE95], the latter because the operating system will utilise a component’s type information for marshaling its method calls.
parameters without having to provide a specialised proxy/stub. By using the automation DLLs, the programmer will restrict himself to using only the OLE compatible types. He will also have to provide a type library for the client to know the interfaces.

COM defines the primitive automation type, safearray, which is an array of Variants. A Variant is a special data type, which is allowed to contain different values such as integers, strings, booleans, etc. As it so happens, the automation types are exactly the ones that fit into such a Variant, and this means that structures may be simulated by creating safearrays within safearrays.

```
theKids := VarArrayCreate([0, (Kids.Count div 3)-1], varVariant);

i := 0;
while(i < Kids.Count) do
begin
  Billy := VarArrayCreate([0, 2], varVariant);
  Billy[0] := Kids.Strings[i];
  Billy[1] := Kids.Strings[i+1];
  Billy[2] := Kids.Strings[i+2];
  theKids[i div 3] := Billy;

  i := i + 3;
end;
```

**Figure 19: Type library marshaling server**

Screen shots from an example, which uses type library marshaling to pass a safearray of safearrays, strings and integers from a client to a server are depicted in Figure 18 and Figure 19. The data sent is information about a person and his/her children, and the structure allows an arbitrary amount of children.

In the upper part of the client, are a number of edit boxes that will accept information concerning the parent. Below this is a list box containing information about the associated children. New children can be added via the “Add” button and the entire data structure can be transmitted to the server by clicking on “Send Data”. When “Add” is pushed, a dialogue box pops up, and allows the user to enter data about a child.

The server merely contains a memo box, in which it will print the received data. It is implemented as an executable, i.e. using the out-of-process convention. The source code for the example is available in appendix A.
In the first example, a safe array is used to hold a list of safearrays, which in turn contains 2 strings for the name of a child and an integer for the age. Filling such an array involves allocation and assignment as shown on Figure 20.

The Delphi version of the safearray allocation routine, VarArrayCreate, is quite similar to the API equivalent [MDELPHI97]. In Delphi a range and the type, which the allocated cells should have, must be specified, whereas in the API function SafeArrayCreate, the type, the number of elements and the destination buffer must be passed. The Delphi function, of course, is nothing more than a wrapper. The API function is shown on Figure 21 for comparison.

```
HRESULT SafeArrayCreate(VARTYPE vt, unsigned int cDims, SAFEARRAYBOUND FAR* rgsabound);
```

**Figure 21: API function to allocate a safearray**

Automation marshaling is by far the simplest type of marshaling that COM offers: simply build the type library, register it and build the component, and that’s it. Any client that is to use the component will also need the type library, but as long as the client and server reside on the same machine, this is not an issue, since the type library is registered in the Registry. However, when the client is on another machine, the type library must be installed on this, also.

### 3.1.6.3 Standard Marshaling

Sometimes the restrictions of type library marshaling are next to unacceptable. This is the case when more sophisticated structures than can be provided with safearrays and the automation types are needed.

The IDL types struct and union are not part of the automation types, and if these are used in a type library, Oleaut32.dll can therefore not be used as proxy/stub DLL. In this case a separate proxy/stub DLL must be supplied for marshaling the structures. The traditional way of defining an interface is to use IDL and compile the file with Microsoft’s IDL compiler, MIDL. Based on the interface description, MIDL can generate a type library, and the necessary C files for producing the proxy/stub DLL. Using an auto-generated proxy/stub is known as standard marshaling [OLE95].

When a method call is invoked, some parameters are inbound, and will be used only inside the method body, while others are outbound, meaning that they will point to meaningful data after the call; some parameters may even be both in- and outbound. Since marshaling is about sending data across process boundaries, this operation might as well be optimised, when possible.

The IDL language defines a number of keywords, IN, OUT, and RETVAL, for this purpose, and these are allowed in the attributes section of a parameter to a function. Figure 22 shows a typical method declaration, in which the programmer has specified that “param1” is only used as an inbound parameter, whereas “param2” is only outbound, and will in fact contain the result of the function upon completion of the call.

```
HRESULT _stdcall MyFunc([IN] int param1, [OUT, RETVAL] char* param2);
```

**Figure 22: Method declaration**

The MIDL compiler will use the keywords for building proxy- and stub components, which will only pass the parameters in the directions indicated by the IN and OUT keywords.

In order to implement standard marshaling, at least MIDL, a linker and a C compiler are needed.

The steps involved in building all the necessary files can be summarised as follows:
1. Write the interface definition in IDL and save it.
2. Write a DEF file for defining the four standard COM functions in the resulting DLL. This file will be equivalent to the one shown in appendix B.
3. Compile the IDL file with MIDL to generate the proxy/stub C files and the type library.
4. Compile the C files and link them into the proxy/stub DLL.
5. Register the DLL with the Registry using the operating system supplied tool, regsvr32.exe. The steps 3 – 5 can be put into a makefile, which can be executed with Microsoft’s nmake utility. A makefile for the standard marshaling server proxy/stub DLL is listed in appendix B.
6. Create a new source file and write the code for the CoClass. Make sure to include the creation of a ClassFactory for all components in the server.
7. Compile and run regsvr32.exe on the server to register it with the Registry.
8. Write the client using the type library to get the definition of structures and interfaces
9. Copy the proxy/stub and the type library to the client machine and register both.
10. Copy the client executable to the client machine and run it.

It is worth to note that the above steps could easily be integrated with an appropriate tool, which would itself turn IDL code or a TLB description into an appropriate proxy/stub marshaling component. As it is, for example with Delphi, there is no support for standard marshaling. This does not mean that it is not possible to build components that use standard marshaling in Delphi, but simply that the above ten steps must be followed.

In writing the IDL file, care must be taken in the specification of the interfaces and the library keyword that tell MIDL to generate a type library as well. Generally the following rules apply to MIDL:

- Omitting the “library” keyword from the IDL code causes only the C files to be generated.
- Writing the interface definitions inside the brackets for the library definition, creates only a type library
- In order to generate both the proxy/stub files and the type library, the interface definitions must be placed outside the library definition, which in turn must contain references to the interfaces wanted in the type library.

In the IDL code shown in Figure 23, which is a stripped version of the IDL code from appendix B, we generate both type library and C files from the same IDL file. Notice how the interface definition comes first outside the library definition, which in turn contains two references to IMarshalStruct: one for the CoClass and one for the library.

Working with the IDL language has taught us that it is possible to generate syntax that the MIDL compiler will accept and turn into proxy/stub code, but which the C compiler would not compile. This was the case when we tried to specify a forward reference to an interface, and defined the library before the interfaces. The C compiler would complain of an illegal syntax, something that

---

13 A source file, which is used by the C compiler to define the exposed functions of a DLL. In the case of COM servers, these functions amount to the four methods DllGetClassObject, DllCanUnloadNow, DllRegisterServer, and DllUnRegisterServer.
was not solved until we moved the definition of the type library below the definition of the interfaces and deleted the forward references. One must tread carefully when dancing with MIDL.

```c
interface IMarshalStruct : IUnknown {...};
/* Type library definition */
[...]
library prjMarshalStruct
{
    [...]
    coclass MarshalStruct {
        [default] interface IMarshalStruct;
    };

    interface IMarshalStruct;
};
```

**Figure 23: IDL code for type library and proxy/stub**

In the example, which was written in order to test standard marshaling, the structure is simpler than that of the automation example. This time we only pass one person between a client and the server, the structure of which can be seen on Figure 24.

```pascal
typedef struct _Person {
    [string] wchar_t *firstname;
    [string] wchar_t *lastname;
    int age;
} Person;

_Person = record
    firstname: PWideChar;
    lastname: PWideChar;
    age: SYSINT;
end;
```

**Figure 24: IDL and Pascal versions of the same structure**

In the top of the figure is the IDL version of the structure, and below it is the Object Pascal [DELPHI97] equivalent. Importing the type library, which contains the structure into Delphi, has generated the latter record.

**Figure 25: Standard marshaling server**

The server of this example will merely receive and print a record on a field in its main window, as can be seen on Figure 25.
The client on Figure 26 has been equipped with 3 input boxes: one for the first name, one for the last name, and one for the age of the person. When the user pushes the button labelled “Submit”, the code on Figure 27 is executed, which results in the data from the input boxes being packaged into the record “data” and sent to the client via the call to `iMarsh.GetPerson`.

```pascal
procedure TForm1.btnSubmitClick(Sender: TObject);
var
  data: _Person;
begin
  data.firstname := StringToOleStr(editFirstName.Text);
  data.lastname := StringToOleStr(editLastName.Text);
  data.age := StrToInt(editAge.Text);

  OleCheck(iMarsh.GetPerson(data));
end;
```

The code that retrieves the data in the server component is similar in complexity to that which transmits the data in the client, and as such standard marshaling is not hard to work with, once the IDL files and proxy/stub DLLs have been built. The problem is that intricate knowledge of IDL is needed, in order to get the C source for the proxy/stub. Furthermore a C compiler and a linker to build the DLL are needed. Finally the proxy/stub DLL must be distributed to all clients that will be accessing the component.

3.1.6.4 Custom Marshaling

Whenever complete control over everything that happens in the entire communication process is wanted, custom marshaling should be used. With this scheme, nothing comes for free and the marshaling process must be manually implemented in its entirety. Custom marshaling could be used for example, if some other transport mechanism for communication than the usual RPC mechanism, e.g. named pipes\(^{14}\), HTTP\(^{15}\) or raw TCP/IP\(^{16}\), or for optimising speed, will be used.

With standard marshaling, COM provides an implementation of the IMarshal interface, but with custom marshaling, IMarshal with its six member functions is – apart from the proxy/stub DLL and CoClass implementation - what a programmer will have to implement himself. The functions of IMarshal will be called by the COM API function CoMarshalInterface in order to package up the parameters and get them on their way.

---

\(^{14}\) Named Pipes: an abstraction for exchanging data between applications running on the Windows platform.

\(^{15}\) Hyper Text Transfer Protocol: the protocol for retrieving World Wide Web documents.

In reality, implementing custom marshaling is probably too much work to be practically useful. This is also the main reason why we have not dealt with it.

3.1.7 Evaluation

Just like traditional object oriented frameworks, COM defines the notion of objects and classes. Similar to some models, e.g. the Java language, the interface part of the classes is separated from the actual class declaration. However, in COM interfaces must be declared separately, while in Java this is merely one way of doing things. So far the model fits nicely into what can be expected from an object oriented framework and component model.

When it comes to naming the classes, GUIDs must be used to ascertain that the object created is the one desired. GUIDs have the disadvantage of being 128-bit long, and thus hard to remember, but are guaranteed to be universally unique, if such a thing is possible. To remedy the naming problem ProgIDs, which are humanly readable, but unfortunately cannot be guaranteed to be unique, can be used. All in all the GUID concept is acceptable, but can be somewhat confusing when discovering that not only are interfaces tagged with these numbers, but libraries, and CoClasses are equipped with these as well.

Traditionally, OO classes have only one interface that they expose: even in the Java language, in which a class is allowed to implement multiple interfaces, the interface of the class is simply the addition of the implemented ones. COM, however, keeps the interfaces that a class implements apart from each other. Getting from one interface to the other can be done via the QueryInterface method, but each of them must be asked for separately. This has the advantage of avoiding name clashes between member functions of interfaces that are implemented by the same CoClass.

The immutability property of COM interfaces is obviously an advantage, since old components will not cease to function in the presence of a new version of an interface. It does imply, however, that there may potentially be many interfaces on a system, which are largely identical, but are not explicitly related in any way. It would have been nice, had COM had the ability to group interfaces that were related in being different versions of the same base interface. In this manner a client could connect to the newest version of a given functionality.

The ability to build different versions of the same class, is not an inherent object oriented feature, but rather relates to component models in general. When instantiating a CoClass there is no way to tell the COM library which version of the class it should use. A developer will first have to locate all the CoClasses that implement the interface in question, by browsing the Windows Registry, and then selecting the GUID for the CoClass with the highest version. We feel that this is a too weak model to be practically useful, and definitely something that Microsoft should improve.

Instantiation by means of API functions is a fine solution, only it is a bit silly that the model provides two such functions, when one would suffice. Furthermore one could argue that from the client’s point of view, knowledge of the ClassFactory ought to be hidden by the system, since requesting IClassFactory is merely an optimisation, and so could be handled by the COM model. This said, the fact that initialisation code may be specified in the ClassFactory’s CreateInstance method and finalisation code in Release is fine.

By separating the interface from the class, COM provides ordinary interface inheritance. This is, as we have seen, a convenient way of specialising the potential abilities of a class, and nothing more. Implementation inheritance can be simulated via aggregation and containment, but this is not true inheritance.
**Containment** is the obvious way of reusing existing implementations: much in the same manner as e.g. Smalltalk programs can use the “super” keyword to get access to an ancestor’s implementation of a virtual procedure, containment allows the delegation of a method call to another class, which implements the same interface. COM, however, does not support the notion of virtual methods.

**Aggregation** in COM terms seems mostly like an optimisation of containment, but since the pointer to the controlling unknown is held by the aggregate, support for constructs like that of Self pointers in SELF is given. The other difference compared to containment is that it is not possible to modify the implementations of aggregated interfaces, and that it is impossible to omit ones that should not be exposed.

Even though IDL is a statically typed language, it allows constructs like the Variant type and pointers to interfaces. The latter feature allows for polymorphism, as we have seen, which is definitely a nice object oriented feature. Of course an implementation language will have to employ some kind of runtime type checking to ensure that interface pointers passed to a function are indeed qualified by the parameter type that the method expects.

Apart from the methods that an interface exposes, the internals of an object are completely hidden from the client. Speaking in OO terms, COM objects have public and private members, only, a strategy known as the ADT approach. As discussed, this kind of encapsulation is the only one possible, when dealing with black boxes like components.

One of the design goals of COM was to provide a language independent model, a goal which was achieved by creating a language for the description of components. This would all have been very nice, were it not for the fact that the IDL language has a dark past. Its origin in the RPC world, following its transition to an object centric environment, has equipped the language with an awful lot of keywords for specifying attributes. IDL is apparently not completely consistent, in that Microsoft’s own IDL compiler, MIDL, chooses to interpret code that ought to define equivalent libraries in a different way. We feel that many of these issues could have been solved in a more elegant manner, e.g. by cutting down on the keywords and the allowable types.

The binary equivalent of an IDL file, the **type library**, is nice to work with, in that the Windows system provides a number of API functions for parsing their contents. Unfortunately this representation is not portable, so perhaps Microsoft should have provided API functions for dealing with IDL files instead? We feel that Windows should ship with a tool for building component descriptions, so that a developer would never have to learn all the tedious details of IDL.

The troubles of IDL naturally carry on to the issue of marshaling method calls, obviously because this is also what the IDL language is made for specifying. We find it tiresome that one has to deal with this issue at all: Microsoft might as well have specified the language in such a way that the system could always take care of things itself. This said, the fact that the underlying RPC layer is completely hidden from the view of the user is nice. The only issue an object user has to worry about is performance, but fortunately it is possible to specify which kind of servers the component can be launched within.

**3.2 DCOM**

To be a complete component model, COM needs to address the concept of distributing components across a network. It has already been explained that out-of-proc servers may be launched on a remote machine, but not how and so far we have ignored the implications. This chapter extends the COM model to include distribution, by introducing DCOM [DCOM97].
The short introduction to DCOM is to say that basically all legacy components are ready for distribution; in fact the only difference between instantiating an object the COM way and launching it in the DCOM fashion is the initial API functions involved. This, however, is not the whole story, since DCOM introduces a lot of other aspects that a developer will have to deal with. These include security, load balancing, and location transparency, to name a few.

Even though legacy components are ready for distribution, there is one important difference between starting an out-of-process server on a local machine from starting it remote: if the server has a user interface, this interface will not be shown on the client machine in the remote case. This fact has the important impact that advanced OLE features like drag-and-drop does not work across machine boundaries, and that it simply is not possible e.g. to start programs like Word or Excel remotely through DCOM. Compare this limitation with the flexibility of the Unix windows system, X-Windows, where the display of an application can be redirected as the user sees fit.

To start a component remotely CoCreateInstanceEx, the cousin of CoCreateInstance, can be used. The most profound difference between these two functions is the presence of a parameter which names the target machine and the NT security level to launch the object server under. CoGetClassObject can also be used to get the ClassFactory of the remote object, since this function is already equipped with a parameter for naming the server.

When a client requests an interface pointer to a remote object, the local SCM contacts the remote SCM and forwards the request. This in turn launches the appropriate server and returns an RPC connection to the local SCM, which corresponds to the class factory of that server. The local SCM then returns this pointer to the COM library which creates a proxy that will internally forward requests to the remote server via RPC.

Out-of-process servers that reside on a different machine than the client will simply be launched and invoked via the RPC mechanism. In-process servers will need a little more taking care of, since they can obviously not be loaded into the client’s process space, which may physically be miles away. This is handled by starting a surrogate EXE server that encapsulates the in-process DLL and simply invokes the methods of its contained object. This feature, however, was not implemented in the early versions of Windows NT 4.0, and Service Pack 2 or greater must be obtained from Microsoft in order to use it.

### 3.2.1 Location transparency

As mentioned in section 2.1.3.2, in an ideal situation, a client should not know or care where a given component physically resides. It should just be able to call CoCreateInstance and get a reference to the desired interface, if this exists somewhere on the network, a feature known as location transparency. But this is unfortunately not how things work.

When invoking CoCreateInstance a specific context handle, known as CLSCTX for “CLasS ConTeXt”, is always passed [DCOM97]. The handle can have four values, depending on which kind of server is of interest, if more than one is registered under the same ClassID. To get a handle to an in-proc object, the CLSCTX_INPROC_SERVER constant must be used, whereas CLSCTX_LOCAL_SERVER will be used for out-of-process servers. Whenever the object server, which hosts the component of interest is on another machine, CLSCTX_REMOTE_SERVER will

---

17 A piece of software that is downloadable from Microsoft’s web site, and which will fix bugs in the operating system, when applied (and be likely to introduce new ones).
be passed. The fourth constant CLSCTX_INPROC_HANDLER is used with proxy/stub components, when implementing custom marshaling, and so is not relevant here.

By adding up the constants above, it is possible to specify one or more kinds of acceptable servers. In the case, when more servers implement the same CoClass, the SCM will first try to start an in-proc server, then an out-of-proc server, and finally go for the remote case.

To be able to auto-launch a remote server without specifying its location, a key known as an AppID is needed in the Registry. Under this key there must be a value called RemoteServerName, which names the remote server on which the component should be launched. There are in addition a number of other keys that may be set here to indicate who have access to the component, who may launch it, and under which user account the component should be run. Compare this strategy with that of CORBA’s global naming service, in which a client need not know anything about the location of a component.

DCOM’s kind of location transparency, it may be argued, is rather weak: there can be only one remote machine in the list in the Registry, and the system is not updated if, e.g. that machine looses the object server. The developer will just have to try his luck and hope that the server is present and not congested by too many users.

### 3.2.2 Performance

An important issue of distributed applications is the performance reduction, which is bound to occur. In COM/DCOM, the fastest results are obtained when a component can be loaded and run in-process, since all method invocations are direct and no marshaling is needed.

When moving to the out-of-process case, parameters have to be marshaled and un-marshaled every time a method is invoked. Marshaling is a time consuming process as can be seen on Figure 28, which shows the number of calls per second and the number of milliseconds each call takes to perform when the parameter size is four and fifty bytes. The table lists measures from three tests, the last of which is the remote case between an Alpha based Windows NT machine and a Pentium based NT. Obviously in-process out-speeds both the out-of-process and the remote case.

<table>
<thead>
<tr>
<th>Parameter Size</th>
<th>4 bytes</th>
<th>50 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Calls / Sec</td>
<td>Msec / Call</td>
</tr>
<tr>
<td>Pentium in-process</td>
<td>3,224,816</td>
<td>0.00031</td>
</tr>
<tr>
<td>Pentium out-of-process</td>
<td>2,377</td>
<td>0.42</td>
</tr>
<tr>
<td>Alpha to Pentium remote</td>
<td>376</td>
<td>2.7</td>
</tr>
</tbody>
</table>

**Figure 28: DCOM Performance measures**

In the test, DCOM was using UDP\(^{18}\) over 10mbps Intel EtherExpress PRO network cards on Microsoft’s corporate network under a normal load. The test has been borrowed from [DCOM96].

It is worth to note that in-process servers will by far out-speed CORBA components, since the latter will always reside in another process space, and so be subject to marshaling. Compared to the out-

---

\(^{18}\) User Datagram Protocol: an unreliable network transport layer, which is the basis for TCP/IP.
of-process components that relate to DCOM, however, the picture may very well be more or less the same. This of course depends on the actual CORBA implementation.

3.2.3 Connection Management
Network connections are inherently more fragile than connections inside a machine. Components in a distributed application need to be notified if a client is not active anymore, especially in the case of a network or hardware failure.

DCOM manages connections to components that are dedicated to a single client, as well as components that are shared by multiple clients, by maintaining a reference count on each. When a client establishes a connection to a component, DCOM increments its reference count, and as soon as the client releases its connection, DCOM decrements the reference count again. If the count reaches zero, the component can free itself.

DCOM uses a pinging protocol where clients periodically send a message to the server to inform it that they are still active. DCOM considers a connection broken if more than three ping periods pass without the component receiving a ping message. If the connection is broken, DCOM decrements the reference count and releases the component if the count has reached zero. From the point of view of the component, both the benign case of a client disconnecting and the fatal case of a network or client machine crash are handled by the same reference counting mechanism.

DCOM uses a per machine keep-alive message. Even if the client machine uses a hundred components on a server machine, a single ping message keeps all the clients connections alive. In addition to consolidating all the ping messages, DCOM minimises the size of these ping messages by using delta pinging. Instead of sending a hundred client identifiers, it creates meta-identifiers that represent all hundred references. If the set of references changes, only the delta between the two reference sets is transmitted. Finally, DCOM turns the big ping message back into regular messages [DCOMBIN96].

A referenced counted life cycle model such as COMs has problems when applied to heavy loaded networks, and when used in systems that are not 100% reliable. If a network connection breaks even just for a short while, the server might think that its clients had died, and would unload. When the clients a few minutes later tried to invoke methods on the server, they would hold invalid pointers, and need to be restarted or restart the server. The de-coupled life cycle maintenance model of CORBA does not have this problem, since the connection is only open when the clients perform requests.

3.2.4 Security
The DCOM security mechanism is tightly coupled to Windows NT’s built in security. Using a special tool, DCOMCNFG.EXE, an object provider can set security restrictions on his local COM servers. This security is based on a per-user and per-process level, which means that it is possible to prevent certain users from accessing certain COM servers, but not possible to prevent a user from invoking certain methods on an object, while granting him access to others.

The security by configuration mechanism allows administrators to specify two types of restrictions, namely who has the right to launch a server and who can access methods and members of server

---

19 For a thorough treatment of DCOM’s security aspects, see [DCOM97].
objects. The user names must be selected from a list of all users that are defined in the current Windows NT domain. The access list is written into the local Windows NT Registry, which means that seasoned programmers don’t have to use the tool but can edit the entries directly using API calls.

When a client attempts to retrieve an interface pointer from a remote server, the remote SCM will ask the security provider whether or not the supplied user name has the appropriate rights to perform that action. If not, the call will fail and the API function that attempted to establish the connection will return an error code.

Recently the IClassFactory2 interface was introduced to provide a mechanism for signing components. To make sure that the user of an object has obtained a legal license, this interface must be implemented instead of IClassFactory. Every time a new object is instantiated, the user will then pass a license key to show authentication. If the license is illegal, the factory can reject and no instance will be created.

It is our impression that the DCOM security mechanism is too complicated to work with in practise. Spending hours trying to set up the right access permissions for one of our legacy COM object servers, we finally capitulated and granted full access to everyone on every security mechanism. This made the technology function, but left our local NT domain completely open to hostile users or components. In general there seems to be an agreement in the DCOM community that to get DCOM to work, this is the strategy to employ, since configuring the security mechanisms is much too difficult.

3.2.5 Load Balancing

In networks with high load, or when multiple users frequently access specific components, it can be crucial to performance to be able to manage which servers run which applications. If only one server contains a piece of software that is used very often by many clients, that machine is bound to have a high load and thus a long response time. If the network consists of many servers, some of which are experiencing a low load, these could help the overburdened server by servicing some of the clients. As discussed earlier, the notion of load balancing is an issue that component models should address, and COM is no exception.

Static load balancing is the only one supported by DCOM. It is obtained by simply setting the appropriate Remote Server variable in the client’s Registry to the desired server. This has to be done by hand, or by the DCOMcnfg tool, which can be used to tune the remote settings for a single machine. Managing an entire network of distributed COM components is thus bound to be a nightmare, but will ensure full time employment for any system administrator.

Microsoft’s reason for not supporting dynamic load balancing is that intimate knowledge of the communication between client and server is necessary in order to determine whether or not the client could be routed to another machine [DCOMTO96]. Some servers keep state information between successive calls from a client, and if this client were suddenly moved to another machine, this information would be lost.

As mentioned in section 2.1.3.3, this is an acceptable strategy, but also the least convenient for the network administrator.
3.2.6 Transaction Management

A product, which is not part of COM, OLE or DCOM, but which improves on the distributed capabilities of the entire model, is the Microsoft Transaction Server (MTS). Originally introduced as the transaction manager for Microsoft’s SQL server 6.5 database, MTS has recently been separated from it, and now forms a separate product.

MTS provides a strong addition to the SCM, in that it enhances the DCOM protocol with object transaction capabilities much in the same way as CORBA’s Transaction Service. When installed, the MTS will set up a service, which will manage calls on objects that have been marked as transactional.

Just like components must be registered with the SCM in ordinary COM, MTS requires that components are installed through its visual tool the MTS Explorer. Via this tool, the transaction level for each component can be set up, and running transactions and object instances can be monitored.

The important thing about MTS is that it is not necessary to implement anything specific for components to participate in transactions: MTS will effectively configure the Registry, so that when a CoCreateInstance is invoked, its transaction manager is loaded, with the component GUID as parameter. Following this, MTS will manage the life-time of the object instance, and any requests on it.

The fact that MTS replaces the SCM in this respect means that in-proc servers will not be real in-proc anymore – they will run in the context of the MTS, wrapped up in a surrogate server. This performance overhead is what the user will have to suffer to gain the benefits of transaction management.

The transaction server imposes a number of restrictions on which components can be managed: only in-proc servers are allowed, they must implement the standard IClassFactory, and custom marshaling is not supported. As a consequence, signing objects with the IClassFactory2 interface is not possible, while working with MTS.

In a distributed system, MTS servers on different machines may talk to one another, thus providing a mechanism for launching remote objects in a transacted manner. One of the servers will be dedicated to function as a global transaction manager, which handles the connections to remote objects, and from which success or failure of the entire transaction will finally be returned.

While MTS is still in its infancy, the advent of Windows NT 5.0 will see it built into the operating system, and probably be further enhanced. Microsoft states that the MTS will then be responsible for managing different resources such as files and databases as well. How this will work, time will tell [DCOM97].

3.2.7 Evaluation

DCOM extends the COM model quite neatly in that existing components are ready for distribution from birth, and so rebuilding them is not necessary. Unfortunately this is only true for components that do not have a graphical user interface. This is a real drawback, which is probably coupled to the fact that the Windows operating system in contrast to X-Windows does not have any mechanism for remoting application displays.

The way DCOM handles location transparency works, but is not very flexible, and we feel that there must be a more convenient way of handling the location independence problem. Instead of
having components listed in the local Registry on each client machine in a network, it could be nice to have some centralised name server like CORBA’s Naming Service.

Such a server might also handle **load balancing** by keeping a list of the machines on the network that contained a given object and forward requests from clients to an appropriate, least burdened server. Furthermore, versioning of components would be easier to maintain, since the Registry server would know exactly which machines ran which versions of which objects.

Remote activation is a slower model than the local case. It is hard to blame DCOM for the bad **performance**, since potentially every distributed object model will be slower than non-distributed ones. On the other hand DCOM does introduce some optimisations for managing the connections between objects and clients, and it is quite nice that it will terminate sessions automatically, when the server no longer responds.

The DCOM **security** model is too difficult to work with, and it would definitely benefit from being made simpler. In addition some kind of tool for managing the security of the entire network would definitely be nice. This feature could well be integrated with the global object server, proposed above.

**Transaction management** can be handled with the MTS server, but alas this service is not part of the current version of Windows, and so has to be bought separately. It is slightly comforting to know, however, that future versions of the operating system will see this feature integrated.

### 3.3 OLE

The term OLE embraces many different technologies, all of which share one common characteristic, namely that they build on top of COM. It is a framework dealing with as different concepts such as file handling, clipboard functionality, drag and drop, object linking and embedding, data transfer between applications, and automating programs, to name a few. A complete list of the OLE technologies as of 1995 can be found in [OLE95], but needless to say that list is no longer exhaustive.

Even though the focus of this thesis is not the OLE technologies, some of them, events and exceptions, are required to make Microsoft’s component model complete. In addition, the automation technology [OLE95] is so tightly coupled to COM, and simplifies its use so much that it deserves attention. The OLE model also extends COM as a component model by introducing persistent objects through its persistent storage technology, wherefore this technology is additionally treated.

#### 3.3.1 Automation

Built on top of COM, OLE **Automation** is a mechanism for cross-application interaction. The protocol extends COM by allowing stand-alone executables to expose their internal functionality to other programs in a simple manner. One example is Microsoft Word, which has a spell checker that can be invoked through OLE Automation, thus allowing other programs to provide spell-checking functionality. Today Microsoft Word can in fact be completely controlled from another application, and in theory this means that one will never have to write a word processor again. Each time a new application should be enriched with word processing functionality, just use Word.

Automation was originally designed to enrich applications with the ability to be used by macro programmers, but as we shall see, it grew to become much more. One may even argue that
automation is a nicer version of COM, because it is simpler, while allowing much of the same functionality as its foundation.

Just like COM interfaces must inherit from IUnknown, automation interfaces derive from IDispatch, which in turn derives directly from IUnknown. IDispatch adds four new methods to interfaces: Invoke, GetIDsOfNames, GetTypeInfoCount, and GetTypeInfo.

Every automation interface has the property of exposing its type information, provided via a type library as described in section 3.1.5. The type information is available through the GetTypeInfo function of the IDispatch interface, and the number of individual CoClasses and interfaces in the type library can be obtained via GetTypeInfoCount. GetTypeInfo will return a pointer to an interface called ITypeInfo, through which programmatic access to all details of the type library in which the interface description lives can be gained. This means that an application can display the methods and properties, as well as parameters and member names to a user. The CASE tool Idun uses this technique to draw a visual representation of type libraries, as discussed in chapter 5.2.

Instead of using the usual V-Table methodology for executing member functions, automation interfaces allow the use of the Invoke function for this purpose. Invoke takes as arguments an integer that identifies the member function, which should be invoked, and a pointer to a structure that holds the parameters, along with some other data. The IDs of interface members can be obtained by first browsing the type library of the interface to get the names of the functions, and then call GetIDsOfNames, passing in the member names in question. GetIDsOfNames will then return an array of integers, corresponding to the IDs needed for the Invoke function.

The IDL language has an “ID” attribute, henceforth referred to as a dispID, which may be attached to the members of an automation interface in order to specify the identifier needed by Invoke. An interface that has the capability of being accessed in such a manner is known as a dispinterface, and must derive from IDispatch.

It is possible to define “clean” dispinterfaces that can only be accessed via the Invoke method and ones that support both V-Table binding and dispatch access. The latter are the most common and are known as dual interfaces. A dual interface must of course also derive from IDispatch.

 Dispatch interfaces have one additional interesting feature, namely that they allow the declaration of properties. A property is declared via two methods, viz. one for getting the value and one for setting it. The two methods will have the same dispID, and the one will have the attribute “propget”, while the other has “propput” to indicate what role it plays. Properties are just syntactic sugar, but any decent language that supports COM ought to support them and hide the underlying methods.

In order to illustrate some of the differences of invoking methods through automation and executing them via a V-Table, a test client and server have been created. The server has functions Peek and Poke for reading from and writing to a private string variable. The server has no graphical interface, and so is not shown here. Figure 29 shows the client with its large output window, its small input box and its four buttons.

The first button “GetNames” retrieves a list of all methods in the server object, the corresponding dispatch ID and the number of arguments expected via GetTypeInfo. This is done by opening the server’s type library and parsing its content via the ITypeInfo interfaces. Output from the “GetNames” function is shown in the first nine lines of the large window on the figure.
The button “Peek” calls the “Peek” method of the server in order to retrieve its locally stored string, and “Poke” sends the string in the input box to the server by passing it as an argument to the “Poke” method.

The last button uses a technique known as late binding, which will be treated further, to write the value “Late binding works!” to the object. It then reads the value again and prints it on the output window. The full source for both client and server can be found in appendix C.

### 3.3.1.1 V-Table, dual or dispatch?

Using a V-Table to access the members of an interface involves only a little overhead. V-Tables are modelled after C++ class function tables, and since a primary design goal of C++ is efficiency, this method has been defined so that it is as fast as making an indirect jump in pure machine code [CPPPL93]. In COM this, of course, is true only for in-process servers that need not perform any marshaling. The V-Table method has the drawback that the client needs to know the position of the functions that the object exposes through a type library or an IDL file. Client invocations of server members can then be translated into V-Table offsets at compile time, a method called very early binding.

Using IDispatch, the client has the advantage of not needing to know the actual positions of members within some V-Table. Instead it can ask the server for the dispIDs corresponding to known function names through GetIDsOfNames and then call Invoke on these. This methodology resembles the strategy taken by Smalltalk, where function calls are statically bound at compile time. Rather the actual method to execute is determined at run-time, by first searching for a match in the target and then in super class chain.

Even though dispatching can be convenient, it is also slow because of the two function calls needed before the actual method can be invoked. Connecting to functions at runtime is known as late binding, and is shown on Figure 30, using a notation that will implicitly be translated into GetIDsOfNames and Invoke calls. Note that the ProgID is used here rather than the GUID. There is no deep consequence of this choice, but it is merely chosen for illustrative purposes.

Compilers can support a third binding model in which they take advantage of the fact that names of methods and properties can be translated into dispIDs at compile time, thus saving a method call at
runtime. This kind of binding is known as **early binding**, but even so it is in fact merely an optimisation of late binding.

```pascal
var
  v: Variant;
begin
  // Create object using late binding
  v := CreateOleObject('AutoGetSet.AutoTest');

  // Set value of automation server
  v.Poke('Late binding works!');

  ...
```

**Figure 30: Late binding the nice way**

An example of this type of binding can be seen on Figure 31, which is taken from the first automation client. Here we utilise our knowledge of the fact that the dispID of the function Poke is two.

```pascal
// Call poke method on disp. intf.
dispInt.Invoke(2, GUID_NULL, m_lcid, DISPATCH_METHOD, Params, nil, nil, nil);
```

**Figure 31: Early binding**

Some compilers e.g. Delphi and Microsoft’s Visual Basic allow a user to invoke methods on automation server objects using a nice notation, as shown on Figure 30. In order to test the overhead involved in packaging arguments into arrays and invoking the necessary functions, we have tried performing the steps that the nice statements perform behind the scene. The invocation of the member method “Peek” expands from one line in the nice notation to some nine lines of code, as can be seen in appendix C.

It is recommended always to implement dual interfaces, obviously because following this scheme allows programs to connect using both V-Table binding and dispIDs, thus enabling a wider range of clients to attach to the server [OLE95]. Another advantage is that IDispatch automatically takes care of marshaling parameters across process boundaries even when connecting through the V-Table portion. It is a drawback that OLE parameters, as will be discussed later on, are restricted to a mere twenty basic types, and that automatic marshaling slows out-of-process V-Table calls so much that they execute at the same speed as IDispatch invocations of the same functions would.

### 3.3.1.2 Automation types

The team at Microsoft, who built VB, originally designed OLE Automation, which is why the legal parameter types of OLE Automation have a nasty smell of Basic about them [COM97]. Automation restricts the developer to using the twenty simple types that can be contained within a Visual Basic Variant, eighteen of which are listed in Figure 32. The remaining two types are Variant*, which is to say that a Variant can hold a pointer to a Variant, and to PVoid, a generic void pointer. These twenty are less useful, and so omitted from the list. All of the types in the list can also be passed as pointers.

Quite often, data types as simple as the ones presented in Figure 32 will not suffice, and the programmer needs to define a more complex structure. In C this would be achieved by defining a struct and passing a pointer to an instance to the function in question. This strategy is possible with COM, as we have discussed in section 3.1.6.3, but not with OLE Automation. However, since one of the basic types is in fact IDispatch, it is possible to define a component that reflects the desired
data structure and pass an interface pointer of it to an automation server. The server can then use
Invoke to get and set the values of the structure’s properties and perhaps eventually return it to the
caller.

<table>
<thead>
<tr>
<th>Type name</th>
<th>Variant Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LONG</td>
<td>VT_I4</td>
<td>4 byte signed integer</td>
</tr>
<tr>
<td>BYTE</td>
<td>VT_UI1</td>
<td>1 bytes unsigned integer</td>
</tr>
<tr>
<td>SHORT</td>
<td>VT_I2</td>
<td>2 byte signed integer</td>
</tr>
<tr>
<td>FLOAT</td>
<td>VT_R4</td>
<td>4 byte real</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>VT_R8</td>
<td>8 byte real</td>
</tr>
<tr>
<td>VARIANT_BOOL</td>
<td>VT_BOOL</td>
<td>True = -1, False = 0</td>
</tr>
<tr>
<td>SCODE</td>
<td>VT_ERROR</td>
<td>OLE error code (HRESULT)</td>
</tr>
<tr>
<td>CY</td>
<td>VT_CY</td>
<td>Currency</td>
</tr>
<tr>
<td>DATE</td>
<td>VT_DATE</td>
<td>Date</td>
</tr>
<tr>
<td>BSTR</td>
<td>VT_BSTR</td>
<td>32 bit pointer to character string</td>
</tr>
<tr>
<td>IUnknown *</td>
<td>VT_UNKNOWN</td>
<td>Pointer to IUnknown interface</td>
</tr>
<tr>
<td>IDispatch *</td>
<td>VT_DISPATCH</td>
<td>Pointer to IDispatch interface</td>
</tr>
<tr>
<td>SAFEARRAY *</td>
<td>VT_ARRAY</td>
<td>Safe array of variants</td>
</tr>
<tr>
<td>CHAR</td>
<td>VT_I1</td>
<td>1 byte signed integer</td>
</tr>
<tr>
<td>USHORT</td>
<td>VT_UI2</td>
<td>2 byte unsigned integer</td>
</tr>
<tr>
<td>ULONG</td>
<td>VT_UI4</td>
<td>4 byte unsigned integer</td>
</tr>
<tr>
<td>INT</td>
<td>VT_INT</td>
<td>Architecture dependent integer</td>
</tr>
<tr>
<td>UINT</td>
<td>VT_UINT</td>
<td>Unsigned architecture dependent integer</td>
</tr>
</tbody>
</table>

Figure 32: Legal OLE types

The disadvantage of this scheme is that every time an item of a data structure that is implemented in
an out-of-proc server is accessed, this translates into a procedure call, complete with marshaling.
This is of course a problem that will apply to any distributed technology that supports remote
procedure calls, and is not specific to COM.

Using an automation server as a data structure is acceptable when the represented information is
small and the server will always be used in-process. For cross-process or cross-machine boundary
calls and large structures, this method is slow, since a large amount of marshaling will be involved.
Furthermore, this strategy will always impose the additional overhead of requiring a call to Invoke
each time data is accessed.

To remedy some of these problems, the original automation server can return data structures as
IUnknown pointers that also implement an additional interface, IDataObject. This has the ability to
transfer an entire data structure at once instead of providing slow access to members. The
IDataObject is part of the OLE technology Uniform Data Transfer [OLE95], which we have not worked with, wherefore it will not be treated any further.

A third scheme would be to create a mirror of the desired data structure using safe arrays in safe arrays. In this approach the semantic information that data structures normally possess, e.g. “name” for a string that represents a person’s name, “age” for his age and so on, is lost. Safe arrays as structures is a method, which was discussed in section 3.1.6.2, and will not be treated any further here. A tool for working with safearrays while retaining the semantic information that a structure provides, is discussed in section 5.1.3.

3.3.2 Events

One of the requirements to a component model is the ability to notify clients of changes within the object. This facility allows clients to take specific action, when changes in the internal state of the component imply that it should do so, as discussed in section 2.1.1.7. In OLE terms, an object which exposes events to interested clients is known as a connectable object, and this technology hence termed Connectable Objects [OLE95].

Central to the OLE version of events is the notion of an event sink, which is an interface that was defined by the programmer of the object server. The event sink, as seen from the server’s point of view, is in fact an external COM object that the server will invoke methods on, when appropriate. This means that event sinks are not implemented by the server object, but rather by the client of the connectable object. When such a client wants to start receiving event notifications, it must pass its own private event sink to the server, and tell the server to start the notification process. When an event occurs in the method of one of the server’s interfaces, the server will invoke the appropriate function on all event sinks it currently holds for that interface, thus letting the clients take specific actions. This scheme is in essence similar to that of Java Beans.

Interfaces that are associated with a corresponding event sink are known as out-going interfaces, because they will be invoking the methods of an event sink within a client. In effect, the client-object-sink relationship is little different from what can be achieved in traditional programming languages by using call-back functions\(^\text{20}\).

It is possible to find out if a given object contains any interfaces, which may be sources of events. First, IDL defines a keyword “source”, which can be attached to the properties section of an interface, and second the object’s type library can be browsed for this kind of information. Through the information contained within the type library, the client can even determine the layout for the event sink that it should implement in order to get notifications.

For a server to expose events to the outside, it needs to implement two additional interfaces, IConnectionPointContainer, and IConnectionPoint. The connection point is an abstraction, which encapsulates the functionality needed for a client to install its event sink on a given outgoing interface. The connection point container, on the other hand, lets the client gain access to a specific connection point via its member functions. This means that an object will implement just one IConnectionPointContainer, but potentially many IConnectionPoints, one for each outgoing interface.

\(^{20}\) Call-back functions: pointers to methods that are passed to another function or object in order to be invoked when a certain event takes place. The pointers allow the other object to “call back” to the source object.
The scenario of installing an event sink in an object that provides event notifications is illustrated on Figure 33: at 1 the client asks for the IConnectionPointContainer interface of the object, uses its member functions to get the ID of the right connection point, and obtain a pointer to it. At 2 the client invokes the member function “Advise”, passing a pointer to its event sink to the function. The client is now ready for notifications, the first of which it receives at 3 by the outgoing interface. Notice that outgoing interfaces by convention are drawn with an arrow sticking out of them, rather than a lollipop.

```pascal
// Instance Internet Explorer, using ProgID
IE := CreateOleObject('InternetExplorer.Application');

// If object was successfully allocated
if(IE <> nil) then
begin
  // Get the IWebBrowserApp interface and store in IWeb
  IE.QueryInterface(IWebBrowserApp, IWeb);

  // Get connection point container, store in CPC
  IWeb.QueryInterface(IConnectionPointContainer, CPC);

  // Get connectionpoint enumerator, store in Enum
  CPC.EnumConnectionPoints(Enum);

  // Get connectionpoint (there is just 1)
  Enum.Next(1, CP, @Fetched);

  // Create and install the event sink
  Sink := TWebBrowserEvents.Create;
  CP.Advise(Sink, Cookie);

  ...
end;
```

Figure 34: Installing the event sink TWebBrowserEvents in Internet Explorer
To illustrate how event sinks work, we have created a small application, which connects to Microsoft’s Internet Explorer, installs an event sink called DWebBrowserEvents, and responds to changes in the browser such as window resize, URL changes and so on.

The source listed in Figure 34 shows how to install the event sink: first instantiate the Internet Explorer by requesting a pointer to the IExplorer interface via the ProgID “Internet.Explorer”. The next step is to get the IWebBrowserApp interface, and from here get to the desired connection point. Finally a new instance of a local implementation of the DWebBrowserEvents interface is instantiated and passed to the server via the advise function. The cookie parameter of advise will contain an integer value upon return, and this cookie must be passed back to the server in the unadvise function, when terminating the event notification process.

The complete source for the example is present in appendix D.

3.3.3 Exception Handling

The notion of handling error situations within components and propagating meaningful information about the problem back to the client, is one of the issues that component models should deal with. Termed exception handling, this feature is often also provided in object oriented languages, e.g. Delphi, Java, and Beta. For a general discussion of exceptions, please refer to section 2.1.1.7.

In the core Component Object Model, where automation does not exist, HRESULTs are the only means by which a component may inform a client that something went wrong during execution of a member function in an interface. HRESULTs can be one of eleven predefined values, or custom ones. When a method invocation is successful, it will return S_OK, and if something unexpected happened, it can return e.g. E_FAIL or custom values. Custom error codes can be very nice, but they have the inherent disadvantage that both the client and the server must agree on their semantic meaning. COM provides no way in which a client can obtain information about the semantics of these error codes.

This problem is solved when using automation. Invoke, which is used for executing methods on a dispatch interface, does in fact take quite a few parameters. One of these is the pExcepInfo parameter, which may point to a structure that describes the exception after an error has occurred within the server [OLE95].

The IDL description for this structure is shown in Figure 35, in which textual properties for the source of the exception, and the cause of the exception live along with the name of a Windows help-file and a context identifier for this help-file. Additionally, the structure contains parameters for returning an error-code in either wCode or scode.

One may argue that it is a bit odd that Microsoft has chosen to use a structure for modelling exceptions rather than an interface with the appropriate properties. There seems to be no good reason for this choice, and it adds to the feeling that the model is not entirely self consistent.

COM provides a set of API functions for dealing with exception handling on a per-thread basis. The system will automatically maintain a list of error objects, one for each thread, and it is these that the

21 Uniform Resource Locator: a way of describing references to documents, pictures and other resources on the World Wide Web. URLs are prefixed by the protocol name, followed by the address and finally the file itself as in http://www.daimi.aau.dk/index.html.
API functions can deal with. An error object is basically an EXCEPINFO structure and an interface ID of the interface that raised the exception.

typedef struct FARSTRUCT tagEXCEPINFO {
    unsigned short wCode; // A code describing the error.
    unsigned short wReserved;
    BSTR bstrSource; // Source of the exception.
    BSTR bstrDescription; // Textual description of the error.
    BSTR bstrHelpFile; // Help file path.
    unsigned long dwHelpContext; // Help context ID.
    void FAR* pvReserved; // Pointer to function that fills in Help and description info.
    HRESULT (STDAPICALLTYPE FAR* pfnDeferredFillIn)(struct tagEXCEPINFO FAR*);
    SCODE scode; // A value describing the error.
} EXCEPINFO, FAR* LPEXCEPINFO;

**Figure 35: The EXCEPINFO exception structure**

A client wanting to retrieve exception information will allocate an instance of EXCEPINFO and pass the pointer to the Invoke method. The server will then fill out that structure when an exception occurs, using the appropriate API functions, and when the call returns, the client will be able to retrieve the information from the structure again.

An object can make known to the world that it supports error information, by implementing the ISupportErrorInfo interface. This has only one method, InterfaceSupportsErrorInfo, which takes one parameter, namely an interface ID. The function will check the ID against the list of implemented interfaces in the component, which it knows have support for exceptions, and return S_OK or S_FALSE if the ID matches or not. This way a client can know how to deal with error situations in the component before starting to use it.

When a language that supports COM building constructs wants to provide a nice means of dealing with COM exceptions, it will simply wrap these in its own ordinary exceptions. Delphi and Visual Basic, for example, use this strategy to provide a homogeneous way of dealing with error situations.

Contrast COM’s exception mechanism with that of CORBA’s. We feel that the latter appears superior in many ways: first and foremost exceptions are part of the core component model in CORBA and not an add-on. Secondly the kinds of exceptions that certain methods may throw can be read directly from the specification of a component, and it is not necessary to invoke the component or parse a type library to get this information. The ISupportErrorInfo mechanism is annoying to work with, and we feel that Microsoft ought to have learned from OMG in this respect.

### 3.3.4 Persistent Storage

Microsoft’s model for supporting persistent objects relies like CORBA’s on a set of interfaces that the developer must implement to equip components with persistent capabilities. Contrary to CORBA, however, there is no system global service that takes care of managing persistent objects. Writing objects persistently is handled entirely by the object itself, while restoring state is done through a set of API functions.

The central abstraction in the persistence model is the Structured Storage technology, which embodies the notion of a file system within a file. This technology enables programs to write hierarchical data structures into single files, and in all respects treat that file as if it were a file
system. The model has storages and streams that represent directories and files, and these can be fully manipulated through the IStorage and IStream interfaces.

The functionality of these interfaces amount to the same set of operations that ordinary file systems expose. This means that it is possible to enumerate the content of storages, open and read from streams, change to sub storages etc.

When a developer wants to create an object that can be persistent, he must implement the IPersistStorage interface, which means specifying seven methods. These include load and save methods as well as functions for checking if the object has changed since its last save, and an operation to inform clients that they should release their storage pointers.

The load and save operations will rely heavily on the Structured Storage technology. In order to associate persistent data with the object type they pertain to, this technology provides a method for writing a GUID to a storage. In a similar manner, actually writing the data is done through IStream interfaces. By creating sub storages, an object can even indicate that it relies on other components and have their persistent data stored here. By invoking the IPersistStorage functions of contained components, it is thus possible to save the transitive closure of all referred objects in the same physical file.

When a client wants to restore an object that was previously made persistent, it will call the OleLoad API function, passing a pointer to an open storage that contains the persistent data. What this function basically does is to retrieve the GUID from that storage, use it to create a new instance via CoCreateInstance, and successively to call the load method of the new object’s IPersistStorage interface. The load method is passed the same IStorage pointer that was used to retrieve the GUID, and can now use its own strategy to restore internal state from the storage. Again OleLoad can be used recursively by the component to restore the transitive closure of part components.

In addition to the IPersistStorage interface, a component could also choose to implement IPersistStream or IPersistFile for simplicity, but thereby it looses the ability to be restored through OleLoad. Instead it is the responsibility of the programmer to perform more or less the same tasks as those of OleLoad. There are a few other IPersistxxx interfaces for making arbitrary memory blocks persistent and the like, but these add nothing new to the model.

This model for persistence conforms to the first design principle of persistent systems by letting the implementation language and persistence model be mutually independent of each other. On the other hand components must implement the IPersist interfaces, and so the second principle that all objects should be allowed the full range of persistence is partially violated. All components can potentially be made persistent, but it must be explicitly implemented. The third principle is fully complied with since the persistent storage model is completely independent of component implementation languages.

Microsoft’s solution to making persistent is very decentralised, and that may potentially pose problems in a network. For one thing it would probably be wise to devise some strategy for storing persistent data in globally available repositories that can be reached from any workstation. If multiple versions of an object’s persistent data lie scattered across the network, it may be quite an experience to change console.

It would have been nice to have a system wide repository for the purpose of storing objects in a persistent manner. A database could for example support the same functionality as the Structured Storage model, and definitely be much more efficient. From the documentation one can learn that a structured storage is not at all fast to work with.
3.3.5 Evaluation

By extending the COM model with automation, in effect Microsoft provides a much simpler component model. It is nice not to need to worry about marshaling, and it is quite flexible that clients can obtain all necessary information to bind to a component’s methods at runtime. This said, automation has a profound problem when it comes to passing complex data structures: by using safe arrays the semantic information that the names of an ordinary structure’s member variables provide will be lost, and if IDispatch pointers are used instead, performance can be seriously compromised. The perhaps most attractive alternative, the data object, is cumbersome to implement, and so there is no really simple and nice solution to the data structure problem.

The fact that automation components provide type information is really good; in fact this was one of the features that we missed, when first implementing a COM object. Still it appears rather awkward that type information can be separated from the component and that it is thus not an integral part of the component itself, but rather an add-on. This means that components that do not supply any type information can be built. These will therefore be practically useless to any outsiders that may want to use them.

When it comes to invoking members via late binding, the developer better pray that his favourite compiler supports wrapping up the Invoke method. Otherwise he will have to write a lot of code each time he wants to execute a simple method. On the other hand, the notion of dual interfaces as a performance optimisation is neat, since the benefits of late binding and the speed of very early binding are achieved.

The event mechanism that the OLE layer provides is in many ways very simple and relatively easy to use for a client. The fact that one interface can expose many different events, and have many clients subscribing to these at once, makes the model quite flexible. On the server side, it is a little tedious that implementations for both IConnectionPoint and IConnectionPointContainer must be provided. The model would definitely benefit from having a simpler way of letting a server expose events, e.g. via a standard implementation for the connection point container.

It is our impression that the simplicity of the event model does not carry on into the realm of OLE exceptions. On the contrary this mechanism seems quite cumbersome to work with. Of course it is nice that there is an exception mechanism at all, only it should have been part of the basic COM model. This would have ensured an unequivocal way of dealing with errors in components and, had the CORBA approach been taken, a more effective one.

The persistence feature of OLE works, but as always it is annoying that it is something that the programmer will have to implement himself. It is simple and easy to use, but relying on the Structured Storage model is perhaps not as flexible and efficient as using databases.

3.4 Summary

When viewed as a complete component model, COM, DCOM, and OLE address many of the issues we have identified that component models should treat. COM targets the basics of the model, i.e. how to define and instantiate objects, invoke methods, and deal with object-oriented issues such as aggregation, composition, inheritance, and encapsulation.

DCOM extends COM with the ability to let objects reside on a remote server, and introduces aspects like load balancing, security, location transparency, and connection management. Many of these issues, however, appear to be either weakly supported or difficult to work with. In addition,
the graphical interfaces of remote objects will not be visible on the client machine, which is a severe limitation.

Completing the component model, OLE introduces persistence, exceptions, and events. Additionally the notion of automation is introduced to create a much simpler component model, which builds directly on the foundations of COM. This model is in many ways nicer to work with, but introduces performance problems as well as severe restrictions on the allowable types. OLE is much more than this, but the additional concepts are centred on functionality needed in the Windows OS, wherefore they have not been dealt with.

The following list sums up how the Microsoft component model in its entirety conforms to the issues discussed in chapter 2.1.

**Language specific concepts:**

- **Objects, Classes, and Interfaces:** COM defines the notion of a class as a collection of interfaces, which in turn contain methods. Objects are black boxes, whose interfaces are the only visible part. It is not possible to access the internals in other ways than through these interfaces. The most primitive interface, which any class must implement, is IUnknown. It contains basic functionality for reference counting and interface casting.

- **Instantiation:** there are two API functions for creating new instances of a component class. The one returns a ClassFactory pointer which can then be used to get the object reference. The other returns the reference directly by implicitly requesting the factory and then creating the instance.

- **Initialisation and Finalisation:** new objects can be initialised in the CreateInstance method of the ClassFactory, and finalised in the Release method, when the reference counter drops to zero.

- **Inheritance:** only interface inheritance is supported in COM: every interface must inherit directly from IUnknown, and multiple inheritance is not allowed. Implementation inheritance is not supported, but aggregation and containment can be used to provide a pseudo inheritance mechanism.

- **Encapsulation:** in being black boxes, accessible only from the interface, encapsulation is an inherent feature of COM.

- **Polymorphism:** languages may choose to provide constructs that treat components implementing the same interface, or descendants thereof polymorphically.

- **Composition:** objects can be composed into new objects, simply by instantiating them in the implementation of the containing component. It is worth to note that the aggregation mechanism can be used to directly expose all interfaces of one component as if they were in fact implemented in the other. Furthermore the outer object of an aggregate may be referenced, thus enabling a SELF-like mechanism.

- **Communication:** contained components may communicate with one another through their methods and properties.

- **Introspection:** objects can be described through IDL or type libraries. When a component has a type library, this can either be linked with it or provided separately. In either case the capabilities of an object is exposed through API calls that operate on the type library. Interfaces that generate events are marked as outgoing, while exception information can be obtained through ISupportsErrorInfo.
• **Exceptions:** COM only has the HRESULTs for propagating error information back to the client. OLE extends this mechanism with exceptions through the invoke method, the catch being that interfaces must inherit from IDispatch and that the mechanisms must be hand-coded for this to work.

• **Events:** connection points provide a simple mechanism for connecting clients to the events of a component. This method scales well to allow multiple connections to a server, but again one must move into the realm of OLE to utilise this feature.

• **Persistence:** built on top of the structured storage model, persistent objects in OLE can be written to disk and resurrected again. A set of interfaces must be implemented in the component for this to be true: there is no simple way to write an arbitrary component to a stream. The model conforms to the three principles with the exception that certain interfaces must be implemented to gain persistence facilities.

• **Concurrency:** tightly coupled to the concurrency model of Windows NT, multithreading in COM is something the programmer will have to consider himself. All of the facilities within the operating system for guarding shared data can be used, but concurrency in COM is no different from concurrency in ordinary Windows programs.

Data related concepts:

• **Revision Management:** COM states that interfaces must be immutable, and so a new revision means a new interface with no inherent connection to the old one. This is annoying, since it leaves no way for a client to connect to a newer version, if it so desires. On the other hand, the immutability rule is simple and easy to follow.

• **Dynamic Replacement:** when replacing a component server in a system, all running clients must first be killed. There is no way to update components incrementally, and no way of undoing changes, other than replacing the new version with the old one again.

• **Transaction Management:** by installing the MTS server, one will get the benefits of transaction management, but unfortunately this program is not yet part of the operating system, and it must be paid for separately.

Distributed concepts:

• **Portability:** components can be built in any language, and the COM model is not coupled to Windows more than it can easily be ported to another platform, something which has already happened. Having said this, the components themselves cannot migrate to other platforms, but are constrained to the architecture on which they were built, because of their binary nature.

• **Activation and Execution:** components can be executed and activated both locally and remotely. For the mixed case, in which a component is activated remotely, but migrates and is executed locally, the ActiveX extension will have to be used.

• **Naming and Location Transparency:** the Windows Registry can be configured so that the SCM can launch any kind of server transparently, in-proc, out-of-proc, or remote. There is no global name server in the model, so if a client should use a remote component, it must either know its location or have the details in its Registry. Every component is identified by a unique name, the GUID. This a little unhandy to work with, and so ProgIDs can be added for human convenience.
Software Components and Development Tools

- **Load Balancing**: managing multiple connections to a remote server is performed in a totally static manner: each client must have its registry-entry for the component changed, which cannot even be done in a centralised manner. In a huge network, this may very well be unacceptable.

- **Security**: the security model of DCOM is totally integrated with Windows NT, and very hard to work with. On the other hand, it provides both programmatic control, and security by configuration. It is a very strong mechanism due to the fact that Microsoft has conformed to the standard set by the U.S. Department of Defence. Furthermore the fact that licensing is provided with the IClassFactory2 interface is probably good for the industry.

**Peripheral concepts:**

- **Customisability and Graphical Interfaces**: the ActiveX model which has not been and will not be discussed in this thesis actually enhances the model with these capabilities. Ordinary out-of-process servers can also have a graphical interface, but to use it the component must run on the same machine as the client.

The overall impression of Microsoft’s component model is that it is strong and scalable. Components of any granularity can be built, and they can reside on any machine in a network. The model deals with the better part of the issues we have defined as relevant for component models, and solves many of the problems in a flexible manner. Unfortunately “flexible” as pertaining to software engineering tends to mean “involved and difficult”. In being flexible, the model imposes quite a burden on the programmer: there is a tremendous amount of interfaces that he must acquaint himself with, each time he wants to explore a new corner of the model.

DCOM is by far the weakest part of the component model, and needs a lot of work before being convenient to work with. Its worst problem is probably the security model, which is simply frustrating to configure. Furthermore DCOM lacks tools for managing an entire network of interconnected components, forcing the administrator to set up information about every single component on every single machine.

As a last remark, it would probably have been wise of Microsoft to have defined the entire model the way they specified automation, since this certainly makes everything stronger and simpler. For example it is not necessary to supply a separate proxy/stub DLL for components, and run-time linking is suddenly possible. Furthermore exceptions and properties become part of the model, and so the overall impression is more complete. It is our impression that a redefinition of the COM model would be beneficial even though it is probably unrealistic.
4 Development Environments for COM

The notion of a V-Table for dispatching class member functions is not at all new, and many OO languages including C++ and Delphi have adapted this approach to organise the layout of classes. This means that it is easy to make a COM mapping for many OO languages, but that does not mean that actually defining the components is necessarily easy. Components can be built with a raw compiler and a text editor, but it is a long ride on the bug plagued road to frustration.

Nowadays more and more Windows applications are being developed in integrated environments that combine the facilities of a text editor, interface builder, and compiler in a single unit, where each part uses information from the others. In this manner, the editor may for example jump to erroneous code that the compiler caught, or the interface builder may insert textual representations of visual controls in the source.

Usually known as Integrated Development Environments (IDE), many of these are well suited for being extended with the capability of building COM components. At the time of writing, Microsoft sells IDEs with COM support for C++, Java, and Visual Basic; IBM offers Smalltalk, C++, and also Java, and Borland sells their Object Pascal environment Delphi, and an environment for C++. Possibly this list is not exhaustive, but the mentioned environments are among the most used today.

Chapter 3 used examples written in C++, but concentrated on Delphi as the sample language. The samples were developed as we acquainted ourselves with COM, and so served to strengthen our theoretical foundation with practical experiences. Delphi was chosen because it had COM support and was a non-Microsoft product, which would allow us to see if the COM technology would work with an environment that does not have the fingerprints of Bill Gates on the back of the cover.

A few months after we had begun working on this thesis, we were given the opportunity to join a research project centred around COM, and our knowledge of Delphi turned out to be useful in that respect. The project is concerned with investigating how different IDEs support the development of COM components, and since we had experience with Delphi, it became our responsibility to deal with that environment.

The research project is funded by the Danish National Centre of IT-Research (CIT), which is an institution under the government’s Department of Research. Its three departments situated close to the universities of Aalborg, Lyngby, and Aarhus, CIT’s main purpose is to promote research in the area of information technology in close conjunction with industrial partners. This co-operation seeks on the one hand to equip companies with useful knowledge gained from basic research and on the other to influence research directions by the needs of the industry.

CIT embraces a wide array of different projects, one of which focuses on the area of object-orientation. Located in Aarhus, this project usually known as COT, which is an acronym for Centre for Object Technology, seeks to carry out research, development and technology transfer within the area of object-orientation. COT is divided into four research areas that are again split into six industrial cases. One of these, case 3 of research area 1, is concerned with component based software development. It is a joint project composed by participants from the University of Aarhus, the Danish Institute of Technology (DTI), and Systematic A/S.

The main goal of the first phase in case 3 is to build a taxonomy of the abilities regarding the development of COM components, which a number of IDEs provide. The environments in question are Borland Delphi, Microsoft Visual Basic, Microsoft Visual C++, and the Microsoft J++ Java
environment. Groups of people from the participating organisations have been assigned one of the four IDEs each.

The Taxonomy is composed of a number of questions that must be answered for each environment. These questions regard aspects that pertain to the mapping from COM to the target language, tools in the IDE that help build object servers, and to the maintenance of the source.

Learning the IDEs in detail is a must in order to construct such a comparison, and to facilitate this learning process, a number of tasks were defined. These are centred around a sample program termed the “Pilot Application”, which is developed and modified as progressing through each task. The main goal of the tasks is to make sure that all relevant features of the IDE that could in any way ease the development process, are tested.

At the time of writing the only IDE for which there exists a completed Taxonomy document is the Delphi IDE, which we evaluated. Preliminary drafts of the Pilot Application implementation process are available for all environments, but the final Taxonomy is yet to be made. We have therefore dedicated a section to relating Delphi to the other IDEs, but that must not be considered an attempt to complete the Taxonomy at all.

This chapter will introduce the tasks involved in working with the Pilot Application, and discusses how Delphi passed the test. Having viewed the Taxonomy’s issues in the light of the Delphi IDE, the chapter is concluded by relating Delphi to the other IDEs.

4.1 IDE APPRAISAL

Investigating the capabilities of development environments was something neither of the involved parties had done before. This meant that we had to invent a scenario ourselves, which we felt confident would test every relevant aspects of the environment with respect to COM development. But what were the relevant aspects? To answer this question, the first meetings were dedicated to brainstorming, the results of which were digested into a number of questions to ask for each IDE. These questions became the Taxonomy document 22.

With the questions defined, we still needed some homogeneous way to progress through the investigation of each environment. This progression should then lead to a state in which we would be able to answer the questions confidently. To be fair, it was necessary to perform the exact same actions in each IDE, but still these actions should be broad enough to give justice to those that exposed more functionality than others. This goal, it may be argued, is hard to achieve with no prior knowledge of the environment, but an attempt was made anyhow.

4.1.1 The Tasks

The basic idea behind the tasks was that they should model a real development history, in which requirements changed over time as bad design was discovered and new knowledge gained. This was of course an artificial situation, since all tasks were presented at once rather than over time.

The program around which the tasks were centred, the Pilot Application, is a calculator component. In the initial tasks, simple arithmetic functionality was implemented, and over time the application was equipped with a graphical user interface and more subtle behaviour. The user interface was

---

22 The draft version of the Taxonomy document is available from http://www.cit.dk/COT. At the time of writing the final report has not yet been made, but in time it should be accessible from this URL.
implemented as a stand-alone program, which uses the calculator component to perform its mathematical operations.

The following sections will introduce all the tasks of the Pilot Application, and discuss the Delphi specific solutions, which we had the responsibility of creating.

4.1.1.1 Task 1.1: Create a component with an arithmetic interface

The initial step was to create the first version of the calculator component with a simple interface. This had methods for addition, subtraction, multiplication, and division, but was defined erroneously. The structure of the first component is shown in Figure 36.

![Figure 36: Initial component definition](image)

Figure 36 shows an excerpt from the corresponding IDL file, and as can be seen, the methods take character arrays as arguments, rather than integers as would be expected. Furthermore the name of the addition function is misspelled as “Dad”. The errors were introduced in order to examine the environment’s reactions to changes, the topic of the next task.

In this first task, the server was to be in-process for simplicity, and in order to test it we would also define a simple client. Needless to say this client could not work like a calculator.

```plaintext
interface ICalc : IUnknown {
    HRESULT _stdcall Dad([in] wchar_t n1, [out, retval] wchar_t *n3)
    HRESULT _stdcall Subtract([in] wchar_t n1, [in] wchar_t n2, [out, retval] wchar_t *n3)
    HRESULT _stdcall Multiply([in] wchar_t n1, [in] wchar_t n2, [out, retval] wchar_t *n3)
    HRESULT _stdcall Divide([in] wchar_t n1, [in] wchar_t n2, [out, retval] wchar_t *n3)
}
```

Figure 37: Interface description of ICalc

When we executed this task, we learned that Delphi allows the developer to use any interface he desires as a base for his own interfaces. We therefore employed the IUnknown interface described in Figure 37. Instead of using the IDL code, we took advantage of the fact that Delphi ships with a graphical tool for building COM servers, the Type Library Editor. Since this is the way one will usually define components in Delphi, this was the method we chose.

Delphi has a number of application wizards that will guide the developer through the initial steps of defining programs of specific types. To build in-process servers, the ActiveX library must be selected, as shown on Figure 38.
This library automatically provides an implementation of the four DLL functions DllGetClassObject, DllCanUnloadNow, DllRegisterServer and DllUnregisterServer, which are used by the Windows NT Service Control Manager to handle the instantiation of objects as well as life time and registration of COM servers. The first step in building the calculator component was to create a new ActiveX library application. The name of this wizard is rather misguiding in that the Delphi programmers mean “ActiveX” as a broad concept unifying everything from simple IUnknown based servers to full-fledged visual ActiveX controls.

Figure 38: Application wizards

This is a typical example of the confusion some marketing executive brought the market, when he coined the term ActiveX as discussed in chapter 3.

We now needed to add a component to the library, a task that was achieved by choosing the OleAutomation wizard from the ActiveX tab, also shown on Figure 38. This makes the dialogue box of Figure 39 appear, and lets the developer set the name and how the component should be allowed to be instanced.

Figure 39: Initial Component Settings

Now Delphi automatically opens the Type Library Editor, from which the desired method declarations can be added to interfaces. Enumerations, interfaces, CoClasses, methods, properties, and dispatch interfaces may be inserted, but structures, unions, typedefs, and modules are read-only for no apparent reason.
This latter fact comes into play, when opening an existing type library with the editor. It will then show these entities, but not allow them to be edited. The Type Library Editor is shown on Figure 40.

Pushing the refresh button causes Delphi to add a new source file with a skeleton for the CoClass and generate a wrapper file, which contains Pascal declarations of the interfaces. It is an interesting point that Borland actually changed the Object Pascal language from version two to version three of the Delphi IDE. In version three the notion of interfaces suddenly appeared along with a method for asking a class whether or not it supports a given interface. A class may implement any number of interfaces, and so the new features were probably added mainly to ease COM development.

When building a COM server, it is also necessary to implement an object factory for each class. Delphi defines several such factories of increasing complexity, which can be used as the programmer sees fit. When using the wizards, Delphi automatically adds the declaration

\begin{verbatim}
  initialization
  TAutoObjectFactory.Create(ComServer, TCalc, Class_Calc, ciMultiInstance);
\end{verbatim}

**Figure 41: Default object factory creation**

to the end of the CoClass unit. An instance of the object factory will then be created by the SCM when the DLL is loaded.

When we had defined the object server in the editor, all that was left to do was to fill out the implementations of the methods in the CoClass skeleton file, and use an entry on the menus to register the server. To test the component, we built a graphical client that was then to be used in all subsequent tasks. This is described in further detail in section 4.1.1.11.

The obvious way of building new components in Delphi is to use the Type Library Editor, but it is also possible to use an IDL file as basis. To do this one will have to compile the IDL file with MIDL to get the TLB file, and then manually tell Delphi that it should use the new type library instead.

This is done by first creating a new project, adding a type library to it, saving all files and then closing the project. The next step is to copy the MIDL generated .TLB file onto the Delphi generated one, and voila: when the project is reopened it has been equipped with the new type library. Using this strategy no new CoClass skeleton units will be created, since these are made by Delphi, when a new automation object is added using that wizard.
4.1.1.2 Task 1.2: Change the existing members of an interface

The previous task introduced errors in the ICalc interface, something that was to be remedied in this task. As such “Dad” was to become “Add”, and the arguments of all methods should change from character arrays to integers. The new method declarations were to become like those shown on Figure 42.

```plaintext
HRESULT _stdcall Add([in] long v1, [in] long v2, [out, retval] long *res)
HRESULT _stdcall Subtract([in] long v1, [in] long v2, [out, retval] long *res)
HRESULT _stdcall Multiply([in] long v1, [in] long v2, [out, retval] long *res)
HRESULT _stdcall Divide([in] long v1, [in] long v2, [out, retval] long *res)
```

Figure 42: Correct ICalc interface

Again the task demanded a client to check the implementations, but this time it should actually perform like a real calculator, i.e. the implementations of the functions should be semantically correct.

In order to correct the errors in the specification from task 1.1, we opened the type library in the editor by selecting the tool from the menus. We then corrected the errors, pushed the refresh button and returned to the source code. All the function headers in both the wrapper file and the CoClass unit had been altered according to our changes, but the old implementations i.e. the bodies of the functions were kept intact.

The Type Library Editor has a hard time deciding whether to use one syntax or the other when auto-defining the method declarations. Sometimes an interface method will be defined as the first declaration of Figure 43. This declaration lets the programmer return HRESULTs himself to indicate success or failure.

At other times the editor decides to use a more “Delphi-like” syntax, where the details of the HRESULTs are hidden. We have not been able to pinpoint the situations, when the editor will generate the one syntax or the other, but may simply point out that now and then it will switch declaration style.

// “IDL-like” syntax using HRESULT as return value
function Add(v1, v2: Integer; out Value: Integer): HRESULT; stdcall;

// “Delphi-like” syntax with implicit exception handling
function Add(v1, v2: Integer): Integer; stdcall;

Figure 43: Different declarations of same method

In most cases, the editor creates the second syntax initially, but when editing the type library one may actually end up in a situation where a previous declaration gets changed from one syntax to the other, and this can be quite tedious.

4.1.1.3 Task 1.3: Add a member to an existing interface

Expanding upon the concept from the previous task, 1.3 introduced a new member function for ICalc. Defined as a modulus operation, adding this method had the purpose of finding out whether or not such a task was at all possible, and checking if the environment would correctly augment any auto-generated code. The new operation is shown on Figure 44.

```plaintext
HRESULT _stdcall Modulus([in] long v1, [in] long v2, [out, retval] long *res)
```

Figure 44: Modulus method for ICalc
Adding a new member to an interface can be done at any point of time with the Type Library Editor, and the tool will update the code to incorporate the changes.

4.1.1.4 Task 2.1: Add an interface to an existing component

Following the completion of the ICalc interface with the basic operations, it was time to add more delicate functionality to the component. To this end a new interface IConv for converting between decimal and hexadecimal radixes was defined. Again the underlying goal was to test the environment’s ability to respond to changes in the library definition. The new library is shown in Figure 45, where it can be seen that the new interface is simply added to the existing CoClass.

![Figure 45: Calc component with 2 interfaces](image)

When adding a new interface in the Type Library Editor, simply push the appropriate button and add methods and properties to it. This done, the editor must be told that the interface belongs to a CoClass, which is done by selecting the “members” tab of Figure 40. This tab displays a list of interfaces that the selected CoClass implements, and through a pop-up menu these can now be adjusted.

Again Delphi had its source updated, and new skeleton implementations for Hex2Dec and Dec2Hex were added.

4.1.1.5 Task 2.2: Remove an interface from an existing component

The reverse operation of adding an interface would of course also have to be tested, and so task 2.2 asked the groups to remove the IConv interface again.

Doing this with the Type Library Editor, we right clicked the interface and selected "delete" from the pop-up menu. This resulted in the wrapper file being updated, but the CoClass still contained both declarations and implementations of the member methods that the deleted interface defined. These had then to be removed by hand.

4.1.1.6 Task 3.1: Add a new component

Moving to the highest abstraction level, the next logical thing to test was the environment’s capability of handling the addition of a new component. This would then reside in the same library as Calc. To keep things simple, we defined the new class as merely implementing the IConv interface from before and named it Conv.
This task was not very difficult or interesting, since we had to do the same actions as those of adding an interface or a method.

4.1.1.7 Task 3.2: Remove a component
As with interfaces, the reverse operation had to be tested, and in our case the Type Library Editor nicely took care of things when we removed the Calc component again. It even cleaned up the wrapper file, but left the CoClass unit as part of the project, and we had to remove that by hand.

4.1.1.8 Task 4.1: Containment
The first tasks concentrated on how the environment responded to augmentations, additions, and removals of entities within an object server. Having exhausted this topic more or less, the next issue that could easily be assisted by a smart IDE is reuse of components.

Since there are two ways in which existing functionality in COM can be reused, viz. containment and aggregation as described in section 3.1.3, it was logical to try out both.

The first task was to try and contain the old Calc component within a new one called CalcDel. This new component would implement another interface ICalc2, which in addition to the old five functions had a new one, Power. The implementations of the five functions would simply delegate to the contained Calc component, and only an implementation of Power had to be supplied. The structure of the new library can be seen on Figure 46.

![Diagram](image.png)

**Figure 46: Containing the Calc component**

To implement a server that exposes the ICalc interface through containment, we started out creating a new ActiveX library and component as described in task 1.1. The GUID defined in the IDL file for this task was pasted into the editor to make sure that we used the same GUIDs as the other groups. The interface for ICalc2 was then defined by adding it from the editor. In a manner similar to that described in task 2.1, the interface was connected to the CoClass. Again the GUID for ICalc2 from the IDL file was pasted into the editor to replace the auto generated one.

Every Delphi defined COM object defines a virtual method Initialize, which the programmer is free to override and where he can specify code that should be executed when the object is launched. In order to delegate the method calls on add, subtract, multiply and divide to the old Calc server, we needed a pointer to its interface, and this was obtained in the initialize procedure, shown on Figure 47.

```delphi
procedure TCalcDel.Initialize;
begin
  Calc := CreateComObject(Class_Calc) as ICalc;
end;
```
Declarations for the global variable Calc and the initialize procedure were added by hand. The five delegating methods were implemented by simply calling the corresponding functions in the old Calc object in a manner similar to the implementation of Add on Figure 48.

```pascal
function TCalcDel.Add(v1, v2: Integer; out Value: Integer): HResult;
begin
  Result := Calc.Add(v1, v2, Value);
end;
```

Finally the code for the new function "Power" was added to the auto generated skeleton.

There is no help to get from the Delphi IDE, when wanting to implement containment. All will have to be done by hand, but since the overhead of doing this is small, and it is quite simple to do it, we find this fact acceptable. Automatic generation of code that handles containment is an issue which is treated in greater detail in chapter 5.2.

4.1.1.9 Task 4.2: Aggregation

Trying out the aggregation facilities of the environment was the next logical step. To this end we defined a new component, CalcAggr, which would also implement the ICalc2 interface. This time ICalc2 would only have one method, though, as the rest would be exposed through an aggregated instance of Calc. The structure of the resulting component is shown on Figure 49.

![Diagram of CalcAggr](image)

Every COM object defined using Delphi and the supplied default implementation of IUnknown is ready to be the inner object of an aggregate. Delphi handles this by keeping a pointer to the controlling IUnknown and by supplying two versions of QueryInterface, AddRef and Release. The first version is for the case when no aggregation takes place, and the second handles the aggregation case. This is similar to the strategy we described in detail in section 3.1.3, which names these two versions as the delegating and the non-delegating unknown, respectively.

The outer object needs a little more work, since it is not possible to define a general QueryInterface in this case. Furthermore the outer object needs to initialise its inner object, when the server is first launched. The outer object was created in a manner similar to that described in task 4.1, the main difference being that the ICalc interface was not connected to the CoClass, since its implementation is to be found in another DLL.

As with containment, we had to add the Initialize procedure shown on Figure 50 below. Notice that CoCreateInstance is passed the variable “Master”, which references an IUnknown interface. If the “Controller” variable of the current object is “nil”, the object is not being aggregated itself, and so is...
the outermost one. A reference to its own IUnknown interface will therefore be passed to CoCreateInstance in order to tell the inner component, by whom it is being aggregated. If the outer component is itself being aggregated, the pointer to the controller must be passed to the inner in order to ensure that QueryInterface will yield the correct results for all interface pointers.

```pascal
procedure TCalc2.Initialize;
var
  Master: IUnknown;
begin
  {Ensure that the outermost controller in aggregation is used}
  if (Controller = nil) then
      Master := Self // Self is outermost object
  else
      Master := Controller; // Self is being aggregated too

  // Get IUnknown of inner object in pUnkInner
  CoCreateInstance(Class_Calc, Master, CLSCTX_INPROC_SERVER, IUnknown, pUnkInner);
end;
```

**Figure 50: Initialising an aggregated component**

The standard COM class in Delphi defines a default implementation for the delegating unknown. This implementation always forwards QueryInterface, AddRef and Release calls if the class is being aggregated. It will never be necessary to change this implementation since if the class is not aggregated, it will simply delegate calls to the non-delegating unknown.

The non-delegating version of QueryInterface on the other hand needs to change to take the aggregated component into consideration. In the COM class this method, called ObjQueryInterface, has therefore been declared virtual so that the implementation can be overridden.

Our new implementation of ObjQueryInterface first checks if the interface requested is implemented by the outer object by calling the Delphi function “GetInterface”. This will return a pointer to the interface requested, if the owner object does implement it. If it does not, ObjQueryInterface will pass the parameters to the inner object’s QueryInterface. This is shown on Figure 51 below.

```pascal
function TCalc2.ObjQueryInterface(const IID: TGUID; out Obj): Integer;
begin
  if GetInterface(IID, Obj) then
      Result := S_OK
  else {interface not in this class: check aggregated class}
      Result := pUnkInner.QueryInterface(IID, Obj);
end;
```

**Figure 51: QueryInterface for outer component**

As noted above, every Delphi component is born with the ability to be aggregated, but a new version of ObjQueryInterface must be made, when wanting to aggregate another component. There is no help to be retrieved from Delphi at all in this point.
4.1.1.10 Task 5: In-process vs out-of-process servers

The next topic to investigate, following that of reuse, became checking that the environment could create both in-process and out-of-process servers. Had the environment support for both kinds, the next logical test would be to change an in-process server to an out-of-process, and vice versa.

Delphi allows the creation of both in-process and out-of-process servers. The latter are made by simply creating a new ordinary application instead of an ActiveX library, and add the automation objects to this, using the wizard.

Changing a server of the one kind to a server of the other is a little bit more difficult. Since programs are defined using a wizard, there is no attribute of these that tell the Delphi compiler what kind of application it should build. That information is provided entirely through the files that are part of the project. This means that a new project will have to be created, and old source files added to it in order to switch from in-process to out-of-process.

When the component library has been defined in the Type Library Editor, the changes are actually saved to a .TLB file, which becomes part of the target application. Every COM library defined through Delphi will thus have its type library linked into it.

When changing the type of the object server Delphi can be fooled into using the old .TLB file by simply copying it onto the new one. The implementation units for the CoClasses from the old project can then manually be added, and the server is finished.

4.1.1.11 Task 6: Create a client with a graphical interface

Testing the components has been part of every task so far, but we have not discussed the actual clients that were used. This is mainly due to the fact that when the tasks were originally devised, it was decided that it would be acceptable to create simple text-based clients.

In Delphi, however, building pure graphical programs is easy, since the environment contains a neat interface builder. Our strategy was therefore to create the client of task six immediately, and use it in all other tasks, while simply patching it as the components changed.

Figure 52: Graphical client for the final components
Task six concerns itself with the development of such a client, primarily to test how the language supports launching of COM servers, but also to get a more convenient user interface for the components.

Delphi has a large suite of graphical components that can be used, and it also supports using ActiveX components as visual controls. Native Delphi controls are linked into the target executable, whereas ActiveX controls are not. We used the native controls for defining the interface of the client shown in Figure 52.

Launching a COM server in Delphi is done either through CoCreateInstance, through a Pascal wrapper for that function, CreateComObject or through another Delphi function CreateOleObject. The latter function takes a ProgID rather than a GUID, uses the API function ProgIdToCLSID to get the corresponding GUID, and calls CoCreateInstance.

Figure 53 shows the three methods that can be used to instantiate components in Delphi. It is worth to note that not all classes have an associated ProgID, and so the last method cannot always be used.

```delphi
Procedure TMyClass.LaunchComp;
Begin
  {use the API function for full control of component instantiation}
  CoCreateInstance(Class_Calc, nil, CLSCTX_INPROC_SERVER, ICalc, Calc);

  {use Delphi wrapper to get IUnknown based interface from GUID}
  Calc := CreateComObject(ICalc);

  {use Delphi wrapper to get IDispatch based interface from ProgID}
  Calc := CreateOleObject('MyCalc.Calc');
End;
```

**Figure 53: Instantiation strategies in Delphi**

4.1.1.12 Task 7: Change interfaces to implement the IDispatch interface

So far the tasks have been concerned with IUnknown based interfaces, but it was appropriate also to check the environment’s abilities to deal with IDispatch interfaces. Specifically it would be interesting to find out if an IDE could change the implementation of existing servers to inherit from IDispatch instead.

To be thorough, we decided to change both the in-process and the out-of-process servers from task five.

Fortunately it is easy to modify Delphi type libraries, as discussed earlier, and to go from IUnknown to IDispatch, we simply opened the Type Library Editor and selected IDispatch as the parent. Furthermore we clicked the dual checkbox in order to get a dual interface for the server. After this, all we had to do was to re-register the server. This strategy was true for both kinds of servers.

4.1.1.13 Task 8: Create installation disks

When components have been coded and are ready for use, they need some kind of installation mechanism, which will allow naive users to add them to their system. Ideally the environment should detect which resources are used by the component, and create an installation package with all relevant files, which will also set the needed Registry entries when run.

Delphi in-process servers support the register and un-register functions, and it is thus possible to install a new component by simply writing "regsvr32.exe Server.dll". Out-of-process servers will
automatically register themselves if started as ordinary executables, and can be unregistered again by supplying the command line switch “/unregserver”.

Delphi can be augmented with a customised version of the InstallShield installation builder program, and this actually ships with the Client/Server version of the IDE. Unfortunately at the time of writing, this tool will not calculate the transitive closure of components used in the project, nor will it automatically set up the required entries in the Registry. What can be done is to tell the tool manually to create registry entries, and add all relevant components to an installation project, and the result will be a self-installing package.

This task asked us to create installation disks for the IUnknown based servers of task five, and the IDispatch based components of task seven, and since these were simple ones that were self-contained, we simply made a couple of batch files that would execute the out-of-process servers and use regsvr32.exe to register the ones that were in-process.

4.1.1.14 Task 9: Exchange implementations

The final test that the IDEs should pass was that of their components being used with other IDEs. Since COM is a language independent technology, this task could be expected to run without any problems. Unfortunately, one thing is theory, and another practice.

To prepare the foundations for exchanging the implementations of task five and task seven, we altered the client of task six, so that it would supposedly work with both IDispatch and IUnknown based servers, no matter if they were launched through GUIDs or through ProgIDs.

To ensure which server was being launched, it was decided to add another interface IHello with one method World to the Calc component of all languages. When invoked, this function would reply that it was implemented in this or that language.

<table>
<thead>
<tr>
<th>Task7 DLL</th>
<th>Servers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Languages</td>
</tr>
<tr>
<td>Clients</td>
<td></td>
</tr>
<tr>
<td>MS Java</td>
<td>OK</td>
</tr>
<tr>
<td>ATL C++</td>
<td>OK (2)</td>
</tr>
<tr>
<td>MS VB</td>
<td>?</td>
</tr>
<tr>
<td>Delphi</td>
<td>OK</td>
</tr>
</tbody>
</table>

1. IHello.World() does not print anything. The rest is OK.

2. Preliminary test on the machine where the Java server was developed. Not trustworthy.

3. iconv.Hex2Dec("FEDABE")=0, whereas Dec2Hex works fine. Upon exit of the client, the server issues a "Runtime error 216 at 00002D7C".

**Figure 54: The test results of the exchange**

The actual exchange was performed during a long day at DTI, the results of which can be seen on Figure 54 above. Originally we had planned to try both the servers from task five and those of task seven, and both the in-process and out-of-process cases. Yet we had many problems, and all we had tested when day ended was the simple IDispatch in-process version, and as can be seen that test is even far from complete.
The Delphi client ran every component without problems, but the server was not tested thoroughly by the other clients, because they never got that far. Only the Java client got to the point where it could perform the test.

An important problem was discovered during the exchange process. Our servers had been defined entirely through the Type Library Editor, and we had not used the IDL files that defined the components of each task directly. Instead we had copied the declarations and made sure that the V-Table order was correct, and that the dispatch IDs matched. Even so, neither of the other clients would work with the server, and the system would angrily report a general protection error. The problem was solved by using MIDL to generate a type library from the IDL files that the others had used, and link that library into the component instead.

The conclusion of these tribulations is that it is not possible to be certain that a type library is identical to one described in an IDL file, if the Type Library Editor was used to build the exact same structure.

4.1.1.15 Additional tasks

The tasks presented in this chapter are far from exhausting the topic of testing component support in development environments. They represent some of the most basic issues that must be addressed, and so there are many other things that could be compared. When this has not been done, it is not because we did not want to, but it stems from the fact that the COT case 3 project involves a number of people from different organisations, many of whom have other things to do apart from assisting in the COM investigation. This means that the process has been lengthy, and even if we wanted to test more issues, there was no time for it.

The Taxonomy document, the results of which will be presented in section 4.1.2, addresses a number of issues that the Pilot Application did not directly force the groups to consider. In order to complete that task, we thus had to make those investigations on our own. The most important and obvious of these issues are listed and briefly discussed below.

- **Properties**: these were not at all tested, and the tasks 1.1-1.3 should be repeated for these entities. Delphi will yield the same results for properties as for methods.

- **Events**: some IDEs might support a language mapping for these, which will allow a programmer to respond to COM events as if they were native language events. Delphi does not do this, however.

- **Error-handling**: the ISupportError interface should be part of the standard implementation of components, if desired. Furthermore, a language mapping should be provided, which would convert HRESULTs and COM exceptions to native language exceptions. Delphi uses this scheme.

- **Early/late-binding**: the language mapping should hide the details of the Invoke method of IDispatch. It should allow both V-Table, dual, and dispatch interfaces, and provide a simple instantiation mechanism for all. Furthermore the programmer should have the freedom to bypass the language mapping and explicitly invoke such methods as QueryInterface or AddRef if he so desires. As the examples of section 3.1.6 imply, Delphi supports all this.

- **Complex structures**: it should be possible to create arbitrarily complex structs and unions, and the IDE should not constrain the programmer in any way here. As mentioned, Delphi does not support structures and unions.
- **Marshaling**: the tool should ideally allow the developer to build all kinds of servers from local in-process to remote standard marshaled ones. Delphi supports the creation of all these kinds, but does not supply a separate proxy/stub DLL.

- **Threading**: even though the COM threading model is the same as that of Windows NT, an IDE may or may not support it fully. It would be interesting therefore to learn how well multithreading issues are dealt with in an IDE. Delphi only supports building single threaded servers.

- **ActiveX and higher level OLE functionality**: this is a huge topic, but even so wizards for generating ActiveX controls exist in many of the IDEs, and so perhaps this ought to be addressed also.

### 4.1.2 The Taxonomy

Based on the knowledge gained from working with the Pilot Application, each group was to fill out a document, which would then be used in the creation of the final Taxonomy. Shaped as a list of questions pertaining to different aspects of working with the tool, the taxonomy document and the answers for Delphi are available in appendix E.

This section will not discuss all the details of the Taxonomy document, but will stick to the interesting points only, summarising and elaborating upon the relevant topics of section 4.1.1.

#### 4.1.2.1 Component Server

When the development of a COM component is to begin, the type of server it should reside in must first be decided upon. Any IDE should allow both in-process and out-of-process servers to be built, and as we have seen, Delphi conforms to that. The transition from one kind of server to another should also be easy, but as Delphi forces the developer to make the choice of server at the beginning of the project’s lifetime, a “copy/paste” solution must be applied for this IDE.

Adding new components to the object server is also easy, conveniently supported by the different wizards. Any kind of COM object may be created, and Delphi even has support for ActiveX and property pages as well.

The content of the COM server is created and modified through the Type Library Editor, which eventually will spit out skeleton code. The generated code only lacks the specification of procedure and property bodies, and Delphi creates a type library wrapper file, which maps IDL entities to Pascal.

If the type library definition of the object server should be changed after some of the method or property bodies have been implemented, the Type Library Editor will nicely leave the implementations alone, while changing the definitions. This means that reengineering code is not a problem.

A problem, on the other hand, is using other people’s type libraries or implementing predefined interfaces. The Type Library Editor will let the developer open existing type libraries, but not connect them to the project. Connections, i.e. references, can be added to libraries in the Windows Registry, but doing so will not allow the use of the types in those libraries in the definition of interfaces and methods. According to the rules of type libraries, it should be possible to use all types in the transitive closure of referenced type libraries. By cleverly renaming an existing type library
that was not made with Delphi to be called the same as the object server project file, but with the .TLB extension instead, any type library can be used, though.

Since any kind of servers can be built in Delphi, including ones that use predefined type libraries, standard marshaling is an issue, the tool should deal with. Unfortunately it does not do that, and the developer is forced to use MIDL and a C compiler to get the proxy/stub DLL. That DLL will, however, work fully with Delphi components. Type library marshaled components, on the other hand, are not a problem.

The language mapping from IDL types to Pascal types is completely specified, and that means that Delphi will handle any types including SafeArrays and Variants as well. User defined types such as structures and unions will also be converted to Pascal equivalents, and thus be usable from the code. This is true even if these types cannot be defined in the Type Library Editor.

The Delphi COM framework provides all the API functions needed for working with components, but also wraps up the details that a developer is basically not interested in: QueryInterface can be called directly on all interfaces, but it is also possible to type cast CoClass variables to the desired interface, which will have the same effect as creating the CoClass and then query for the desired interface.

When using Delphi’s wrapper to create the CoClass it is not necessary to call AddRef and Release explicitly, since the runtime engine will keep track of all interface variables and simply do an implicit AddRef, when assigning an interface to a new variable. Similarly Release will be invoked when such a variable goes out of scope or is assigned a new value, e.g. “nil”.

The same philosophy applies to interfaces that inherit from IDispatch: Delphi hides the details of Invoke as the code of Figure 30 and Figure 31 on page 59 shows, but lets the developer do all the hard work too, if he has masochistic tendencies. The HRESULT and exception error handling strategies are likewise wrapped up in Delphi equivalents, but again the details can be manually handled if desired.

4.1.2.2 Compiling and Distribution

Once a component server has been defined, the programmer will be interested in packing up the files that belong to it. The Install Shield program can be used for this purpose, but it will only handle projects that contain an executable file, and therefore it cannot be used with in-process servers. For that reason, an installation package might as well build by hand.

Fortunately all object servers will contain either an implementation of DllRegisterServer and DllUnregisterServer or will be self registering, when run as ordinary programs. It is worth to note that in-process servers may be registered and unregistered from Delphi’s menus as well.

The target types of the final build may be either thin servers that do not link the Delphi runtime engine or thick ones that do. When planning to install many components that were made with Delphi, the first choice is the best, otherwise the second will take up the smallest amount of bytes.

4.1.2.3 Reuse of Controls

Using and reusing Delphi COM components can be done in any of the three possible ways, depending on the kinds that are legal with that particular server. Components that only expose a V-Table, as well as ones that provide dual interfaces, may thus be created. The clients may bind very
early through the V-Table, early through the dispIDs obtained at compile time, or late at runtime through dispIDs.

When it comes to building components from components, Delphi objects are readily aggregateable from birth, which means that they can be the inner of an aggregate. To create the outer, though, a little work is needed, since ObjQueryInterface must be redefined in order to take the aggregates into consideration.

In a manner quite analogous to aggregation, Delphi exposes no direct support for containment, but there is no stopping the developer from creating the contained component himself and manually adding the statements that delegate calls. This process is in fact so simple that wrapping it up in the language seems superfluous.

4.1.2.4 Development Environment Support

Components are integrated into the Delphi IDE in such a manner that they can be manipulated as any other Delphi program. This means that the debugger will work; for in-process servers only the specification of a client executable’s name is needed, and out-of-process servers will just have to be run from within the Delphi environment to get this functionality. It is worth to note that by starting multiple instances of Delphi it is in fact possible to debug from client into server and back again. This is also true if the client or the server is being developed in another IDE, e.g. J++ or Visual C++.

All in all the Delphi IDE appears strong and flexible. It has a number of quirks that need to be taken care of in the Type Library Editor, and there is no good support for standard interfaces. These problems are not crucial though, since a work-around can help overcome the limitations.

4.1.3 Delphi and the other IDEs

Even though the full Taxonomy document is not yet available, draft versions of the Pilot Application implementation process have been created, and so it is possible to relate the IDEs anyhow. This section will not attempt a full comparison, but will present the main differences between Delphi and the other environments, and will highlight important points and aspects of each. A subsection is dedicated to each of the other three environments.

4.1.3.1 Microsoft Visual C++

In contrast to Delphi, the people behind the COM support in Visual C++ (VC), did not choose to extend the language with keywords for defining interfaces and functions for checking whether or not a given class implements some interface. Instead VC defines a macro language known as the ActiveX Template Library (ATL), which lets the developer specify the interfaces that the QueryInterface mechanism of a class should respond positively to. In addition, the C++ multiple inheritance mechanism is used to specify which interfaces a given CoClass implements. The interfaces themselves are simply defined as abstract classes.

There is a very strong wizard support in the VC environment. Similarly to the Delphi strategy, the developer must initially choose between in-process and out-of-process flavours, when starting a component project. Additionally VC also supports building COM objects as Windows NT services and through a checkbox the developer can choose to let the wizard generate proxy/stub code that will be linked into the final target. As such the project wizard in VC is superior to that of Delphi.
This is also true when it comes to adding new classes to a component server. In VC the developer has complete control over all filenames in the project and can even have the wizard generate source files that will support any of the threading models supported by Windows NT. As a bonus, checkboxes will allow the developer to select support for connection points also.

Reference counting is explicit, but by overloading the assignment operator and associated operations, it is possible to use so-called smart pointers for automatic reference counting. Similarly QueryInterfacing is explicit, and the COM API functions will be put to use for object instantiation.

VC has no notion of a Type Library Editor. All server definition, including the addition of interfaces and methods, is done through wizards. When all steps have been completed, the wizard spits out all necessary C++ files and an IDL file. At this point, there is no easy turning back.

Reengineering code automatically is not possible, and all changes must be done manually. If for example the type of a method is to be changed, the developer will have to alter the IDL file, a header file, and the C++ file that contains the CoClass implementation. This strategy is very inflexible compared to Delphi’s.

The aggregation mechanism in VC is similar to that of Delphi’s in that components can be made aggregateable automatically, but in order to aggregate other components, code must be added manually. The same is true for containment, for which there is no wizard support, but can easily be used.

All in all, VC is probably the IDE that allows the widest range of component types to be developed through wizards, but as soon as the wizard rests his case, the developer is totally alone.

4.1.3.2  Microsoft Visual J++

Java programs are compiled into a platform independent byte code, which is executed by a runtime engine known as the Java Virtual Machine (JVM). The Microsoft version of this interpreter has been extended in a manner that lets any Java object be a COM object.

The Java language already defines interfaces as separate to the classes that implement them, and like COM, a class can implement multiple interfaces. This way the Java language gets for free, what Delphi needed cosmetic surgery to obtain, namely a nice language mapping for COM.

All COM details are hidden totally from the developer, which means that reference counting is implicitly performed when variables are assigned COM instances, and when those variables go out of scope or no longer refer to the instance. In a similar manner, QueryInterface is implicitly performed by typecasting instance variables to the appropriate interfaces. As such the VJ strategy is identical to that provided by Delphi. Unlike Delphi, however, it is not possible to undertake reference counting and interface querying explicitly.

A neat feature of the language mapping in VJ is the way aggregation and containment is handled. Another CoClass is exposed in either ways by simply inheriting from it as one would with any ordinary Java class. If the methods of the old class are redeclared, containment is used, and if not, aggregation. Through the “super” keyword it is possible to access the redeclared methods of a contained CoClass. Unfortunately the Java language does not support multiple inheritance for classes, so only one CoClass can be either contained or aggregated, which is a severe limitation.

Together with VC, Visual J++ (VJ) is part of Microsoft’s multi-language IDE Developer Studio. Unlike VC, however, there is no wizard support for developing COM objects in VJ. The strategy for building components from scratch can be outlined as follows: first write the IDL file by hand.
and use MIDL to get the TLB equivalent. Next compile the TLB file with a command line tool, JavaTLB, to get binary Java class files that represent the entities in the library. Finally write the CoClass code by hand and use the Javareg command line tool to register the server.

There is no notion of an object server in VJ, and instead the JVM must be shipped with the binaries that define the COM classes. In the Registry, the JVM is specified as the local server with the particular COM classes as parameters. These classes must then be in the system search path in order to function. A consequence of this strategy is that all VJ COM components are in-process, because the virtual machine is contained within a DLL.

The overall impression of the VJ COM support is that the language mapping is nice and simple, but the wizard support too poor. One might argue that Java is not particularly suited for developing COM components, since it relies on a virtual machine that must ship with any component. Such an engine is bound to be executing slower than ordinary components, and since it must first be loaded to memory before the Java COM classes can be run, it is not suited for mission critical applications.

4.1.3.3 Microsoft Visual Basic

The COM support in Visual Basic (VB), was apparently defined with automation in mind, and as mentioned earlier that specific part of the component model was in fact largely inspired by VB. There is no way to defined other servers than those supporting dual interfaces in VB, which means that only the automation types are available.

Similar to VJ and Delphi, the COM details in VB are hidden from the developer. Objects are instanced with the “new” keyword and interfaces queried for by simple assignment to interface variables. Similarly reference counting is implicitly performed as in VJ. Unlike Delphi there is no way to handle the details manually.

New servers are created through the project wizard in a manner quite similar to Delphi and VC. Both in-process and out-of-process servers are supported, but the wizard will not generate skeleton code. On the other hand a server can at any time be changed to another flavour, in-process or out-of-process, by simply clicking a switch.

The process of defining the server structure is reversed in that the developer simply adds a so-called ClassModule to the project and subsequently adds method declarations in VB syntax by hand. The compiler will then use the information in the modules of the project to infer the server structure, automatically build a TLB file and link that file into the target.

VB is probably the most restricted of the four IDEs in that it is not even possible to name interfaces. These will automatically be given the name of the class module, but prepended with an underscore. Multiple interfaces can be added, but the result is ugly to look at if one is curious enough to view the type library in another tool. VB names the interfaces itself and attaches GUIDs automatically, which the developer has no influence upon.

Creation of instances is only possible through the “new” keyword and the CreateObject function, which takes a ProgID as argument. Direct manipulation of GUIDs is not possible. Restrictions are also imposed on the reuse mechanisms: containment facilities can be added by hand, and aggregation is not supported.

It is easy to define automation servers in VB, but if full control of all details is needed, VB is not the choice. The inability to specify GUIDs means that it is impossible to provide custom
implementations of predefined interfaces and thereby for example event mechanisms. Also the tight coupling to automation disables the possibility of using sophisticated data structures.

4.1.3.4 Summary
There is no doubt that VC is the strongest of the environments with respect to what kind of servers can readily be implemented. Right at its heals, Delphi outruns both VJ and VB, both of which are quite constrained. On the other hand programs are not developed linearly, but rather in circular processes where the project is revised and tested. Delphi scores high on supporting this process by easing reengineering. It is followed by VB, which has no graphical tool but in which it is easy to edit the components. VJ and VC share the third position by forcing the developer to manually change a number of files, including having to bother with IDL.

The conclusion of the comparison is that Delphi is a strong IDE that will suffice for most needs, and that VC is for developers that like to get their hands really dirty. With their inherent encapsulation strategies, both VJ and VB are for developers that do not know or do not want to know the details of the COM framework. Our choice would be Delphi for general component definition and VC for that rare case which Delphi cannot handle.

4.2 Evaluation
Prior to our participation in the COT Case 3 project, we had some knowledge of developing ordinary Delphi applications. We used the IDE during the work on our thesis in order to learn the aspects of COM development. When we joined the project we had worked for just a few months, and so were really novices. Being forced to consider the issues of the Pilot Application and later the Taxonomy document made us learn details of Delphi, which we would probably not have touched upon otherwise. This include the notion of aggregation and the reengineering process in detail.

We find that building a pilot application that seeks to give answers to the questions in a taxonomy document is a good strategy for evaluation of IDEs. For one thing it ensures that all the environments are exposed to the same tests, and if these are thorough enough, the final taxonomy will not have been made on false pretences. This said, we feel that the Pilot Application used did not answer all the questions, but the Taxonomy document seems to cover the interesting areas quite well.

Another lesson learned, which does not relate to the IDE, is the fact that it can be a lengthy process to work with many people across companies. Very often we would have to wait for some resource or definition before we could get on with our work, and needless to say this fact was annoying. The reason for the delays can be found in the fact that we worked full time on the case in some periods, whereas most of the other participants had other tasks to do in their jobs. Given these circumstances the delays are probably inevitable.

Returning to the Delphi IDE, our overall impression of this product is that it is highly suited for developing COM components. It has a number of features that are hard to live without once getting to know them: first and foremost the fact that a number of base classes hide the implementation details of COM specifics like reference counting and interface querying is superior, since the resulting code is small and less error prone. If the Delphi wrapping is not suitable, on the other hand, the framework is not a straight-jacket and a programmer can just build his own classes using the COM API.

Type libraries are easily defined using the editor, and the resulting automatically generated skeleton code simply needs a little work before the components are complete. Once defined, the
skeleton files may even be changed at the developer’s leisure, as long as he does not change the
method declarations of implemented interfaces. A fair restriction indeed, when viewed in the light
that type libraries may be reengineered from the editor without loosing the implementations.

When the server has been through its final build, it is self registering, taking away the burden of
building an installation program, which does this, from the programmer. Still he may need to build
such a program anyway, since Delphi does not help in collecting the files for the final project.

Unfortunately that is not the only problem Delphi has. The fact that the tool must be fooled to use
an existing type library that was not defined in Delphi is a nuisance, since this means that the
CoClass skeletons must be written by hand when doing so. These are namely generated, when the
wizards are used to add a new automation object.

Furthermore it seems strange that the Type Library Editor does not spit out skeleton code for the
outer of an aggregate. Naturally such code could not be hidden in base class like the code for the
inner of an aggregate can, since it depends on which components are aggregated. Still it is not all
that much work to complete the aggregation issue. Similarly mechanisms by which containment
could be modelled in the Type Library Editor, and corresponding code generated, could easily be
provided.

The reason that the Type Library Editor does not allow automatic code for aggregation and
containment can probably be sought in the focus of its designers. It appears that the editor was
modelled after the entities that a type library can contain and not according to more general design
concepts.

All in all the weakest link in the chain does in fact appear to be the Type Library Editor: sometimes
it will generate illegal CoClass code and it does not allow the construction of complex types or the
use of these types in referenced libraries. Furthermore help topics cannot be viewed from an
associated help file. In addition to all this, proxy/stub components will not be automatically added
to the project when creating out-of-process standard marshaled COM servers. This latter fact means
that tools from other vendors must be put to use in order to build these kinds of servers in Delphi,
and Borland can hardly find that satisfying.

Even though the final Taxonomy has not been created at the time of writing, a comparison of key
points in the four IDEs was possible by investigating the draft versions of the Pilot Application.
There is a clear tendency to wrap up and hide the bothersome issues of reference counting and
interface querying from the developer.

VB and VJ only support COM through their wrapping mechanisms, and it is not possible to do
explicit reference counting or interface querying. Delphi also exposes COM through a language
mapping, but does not enforce the use of that mapping. It is thus possible to control all aspects
explicitly. Finally VC has no language mapping, but through the use of smart-pointers, reference
counting is made easier.

The IDE that has best support for reengineering is Delphi, which will automatically update the
appropriate skeleton files, when the structure of the type library changes. VB is also simple to work
with, since it is only necessary to change the method declarations by hand. VC and VJ have no
support for reengineering, and a change in the definition of the type library forces a manual changes
of a number of files.
VC is the IDE that allows the widest range of components to be built. Delphi is second in line because it only allows single threaded servers and no merging of proxy/stub code. VB and VJ restrain the programmer to developing only simple servers and VB even just automation servers.

The overall conclusion is that anything can be built in VC, but for most purposes Delphi is the better choice.
5 Tools

The Component Object Model is wide-ranging and consists of many small areas where a programmer would definitely benefit from having a tool that could assist him. Some of these areas cover subtle aspects that will target a narrow public. Examples include utilities to prevent multithreading conflicts in servers and tools for using safearrays as if they were in fact structs, the members of which could be accessed via ordinary dot notation.

The most typical tool that a programmer would need is one for composing component servers, and all of the IDEs that were investigated in case 3 also expose some kind of mechanism for building COM components. Each has its own strategy, which may be more or less flexible. Visual C++ uses wizards entirely to let the programmer define his component servers. Having answered a number of questions pertaining to the structure of the server, the wizard will create a new project with default implementations of the four DLL functions, IUnknown, IDispatch, etc.

The language support for COM is very strong in Visual C++, because it allows the construction of any kind of COM components and modify every single tiny detail of the source. However, the wizard approach raises one major problem, namely that pertaining to reengineering. Since the code structure was defined via a bunch of questions, there is no intermediate representation that may be returned to and altered in order to modify the underlying code.

The J++ environment is even worse off than Visual C++ in that it doesn’t even create CoClass skeleton code for the classes, and in that it has no visual tool for defining the object server’s structure. In J++ the structure is hand-coded into an IDL file, compiled with MIDL and a Java type library compiler used to get Java classes that represent the entities of the library. The CoClass skeleton code must then be written and the autogenerated classes included into the project. No reengineering support is available and the developer must go back to the IDL code and perform the tasks from scratch.

Visual Basic and Delphi ship with a CASE tool in which the structure of COM servers can be defined, and through which the corresponding code can be obtained. The tool may be reopened at any time, the structure of the library altered, and the source thereby effectively reengineered. The strength of these tools are that they have been carefully tailored to fit the IDE that they service. Yet it is also a weakness that the tools are tied to their host language and not really flexible when it comes to using predefined type libraries.

Some of the IDEs force the developer to select the target language prior to defining the structure of the library, which seems odd when considering that COM is supposed to be a language independent technology. Typically one is not interested in being restrained to one target language only, but will want to make that choice after having defined the structure of the library. When crafting software, the analysis process is typically separated from the implementation and so it ought to be with components, also.

IDL is not well suited for getting an impression of the content within a library, and so what is needed is a tool that will allow the structure of component servers to be defined visually. The tool must be highly flexible and allow representations of the library in any target language to be generated. It must support reengineering of these representations also to ease development iterations and be easy to integrate with existing IDEs or design tools.
5.1 The Toolbox

This chapter identifies and discusses a number of tools that would ease the daily consort with COM. The more specialised tools in the toolbox are treated first and only briefly. An exception is the Semantic Wrapper for Saferarrays, where we plunge into greater detail because this is perhaps the most useful of the specialised tools.

In the top of the toolbox lives the utensil that the COM programmer uses everyday. Being a CASE tool for building COM components, complete with reengineering facilities and the ability to generate code in any target language, this is discussed in full detail. The CASE tool is subject to an actual implementation, a description of which concludes the chapter.

5.1.1 OO-CASE Tool

When designing the structure of a program using object oriented analysis and design methods, it can often be beneficial to build the entire model in a CASE tool. This way the design can be documented in a professional way and some tools will even allow code to be generated for a specific target language. This is true for the newest version of Freja by Mjølner Informatics [FREJA96], which will create Beta source for OO models, and for Rational Rose [ROSE98] by Rational Software Corporation, which handles a number of languages including C++ and Java. Both editors use UML for their object descriptions.

The newest version of Rational Rose will allow parts of the model to be defined as COM components, but the tool has no inherent connection to an underlying type library or IDL file. It will not import these descriptions in a project, and will as such basically serve as a documentation utility with respect to COM.

Instead of extending the existing CASE tools to embrace components also, these could be augmented through a third party tool that would do the job. A utility that works at a component level, and which can be fully integrated with third party applications is described in chapter 5.2. Such a tool could be used for creating and editing COM objects from within existing programs like Rose and Freja, with little effort.

5.1.2 Free Threader

By default COM uses the single threaded apartment model. With this scheme, the COM server contains just one thread, and each time a client accesses the server, its request is stored in a message queue. This way multiple clients will wait for the object that currently has the access to the server, and the requests are effectively serialised. The Windows system handles all this synchronisation, and the programmer is freed from the burden of identifying critical sections in his code.

For reasons of efficiency the single threaded model is not always desirable, and a method in which many clients gain concurrent access to the server should be employed. To this end COM defines the notion of a multi-threaded apartment. With this scheme, no synchronisation takes place, and it is up to the programmer to protect critical sections by such mechanisms as semaphores or monitors.

---

23 The referred version is not the newest and does not support UML. It can, however, be checked from the homepage of Mjølner Informatics at http://www.mjolner.dk that the newest version does.

24 Unified Modelling Language: a graphical notation for defining OO-models.
A tool, which could identify non-thread safe code in servers that were not written with multi-threading in mind, and maybe even spit out the thread safe equivalent, would definitely be desirable. However, identifying critical sections may not be a simple task, and would often require that the tool had access to all source files involved in building the server. In addition, the use of pointers can probably lead to situations in which the tool is not able to decide whether or not a given variable is thread safe. The tool would then be forced to walk along the safe way and protect the dubious code.

This kind of tool targets a very narrow public, and the question is whether it would be smarter to redesign the legacy component to be multi-threaded instead of relying on some tool doing an adequate job.

5.1.3 Semantic Wrapper for Safe Arrays

When defining a new COM server, a decision has to be made on the types that will be allowed for methods. If custom structures are included, separate proxy/stub code must be supplied. This is tedious, and over time quite inflexible in that a new such proxy/stub file must be created each time the server structure is altered. That file must then be delivered to both the client and the server machine again. The advantage of using custom structures is that the logic of the data can be defined by providing meaningful names for variables, which will make the programming task easier.

Another scheme is to restrict the parameters to those that can be contained within an OLE variant and let the operating system take care of the marshaling process. Most structs can be translated into an equivalent safearray, and arrays of structs can be converted to safearrays within safearrays. From the programmer’s point of view the conversion results in the loss of the logic which was implicitly provided by the meaningful names inside the struct, since the elements of a safearray variable will be unnamed. Furthermore, using safearrays implies exercising some twenty API functions for allocating, locking, unlocking, destroying and accessing elements within the safearray. With deeply nested structures, this code can be quite large and easily a source of errors.

If a programmer could keep the semantic information from the original structs while at the same time harvesting the benefits of automatic marshaling and not having to know anything about the safearray API functions, working with variant types would not be so bad after all. Providing this kind of functionality is the purpose of the tool described in this section.

The basic idea behind the semantic wrapper is to create a file that contains code for a class which given a safearray will allow the programmer to access its items in a manner identical to that, which he would have employed, were he to work with an ordinary structure instead. As input, the tool takes a file with a description of the possibly nested structure, e.g. written in a subset of IDL, and as output it produces the file that was just described, possibly as Beta or Pascal code.

```
typedef struct {
    BSTR name;
    Int age;
} person;

typedef struct {
    person father;
    person mother;
    int num_children;
    [size_is(num_children)] person *children;
} family;
```
Consider the IDL code of Figure 55. This describes the structure of a family, complete with father, mother and an array of children. The family structure uses the person structure to define its member variables, and the names of these in turn have special meaning to the programmer. He would like to be able to refer to the mother’s age as `family.mother.age`, rather than having to index his way through arrays of arrays of variants. The family structure converted to a safearray is depicted in Figure 56.

When the IDL code is compiled with the tool, a class is generated for each struct, and each simple type is mapped into a property of the class. Furthermore each class is equipped with a constructor, which takes a safearray of the appropriate structure and instantiates a tree of objects that represent the structure of the original IDL code. If we turn to the family example, shown on Figure 57, an invocation of `person.create(mySafearray)` would result in an object with part objects of the class person, each of which would have a pointer to the corresponding data item(s) in `mySafearray`. By reading from and writing to the members of the new object, the programmer can now manipulate the content of the safearray.

The person structure would, when compiled with the wrapper tool, generate a class similar to that described in Figure 57. The data member of TPerson holds a pointer to the safearray which is to be manipulated, and each of the original members from the IDL code give rise to a property of the same name and corresponding functions for reading and writing the safearray element. A special property parray makes a way of getting the encapsulated safearray, e.g. for returning it as a result of a function call. Notice how the implementation of get_name uses one of the safearray API functions for actually retrieving the data corresponding to the name of the person, which this object represents.
Once the wrapper classes have been generated, they can be used both from a COM server and a client in a straightforward manner. The client can use the member function create_new to instance a new object with an empty safearray in it.
type
type
TPerson = class
private
data: PSafeArray;
function get_array: PSafeArray;
function get_name: string;
procedure set_name(v: string);
function get_age: integer;
procedure set_age(v: integer);
constructor create(var data: PSafeArray);
constructor create_new;
destructor destroy; override;
published
property name: string read get_name write set_name;
property age: integer read get_age write set_age;
property parray: PSafeArray read get_array;
end;

implementation

function TPerson.get_name: string;
var
  o: OleVariant;
  i: LongInt;
begin
  i := 0;
  SafeArrayGetElement(data, i, o);
  Result := o;
end;

Figure 57: Auto generated wrapper class for person structure

The safearray is allocated in the create function, but no data is poured into it. Using the members of
the object, the programmer can now write a client, which fills the data structure with meaningful
content, gets the pointer to the safearray, and sends this to the server via a simple function call. This
scenario is illustrated in Figure 58.

Procedure myClient.SendFamily;
var
  myFamily: TFamily;
begin
  myFamily := TFamily.create_new;
  myFamily.father.age := 52;
  myFamily.father.name := 'Palle';
  myServer.getFamily(myFamily.parray);
end;

Figure 58: Use of wrapper in client

Similarly the programmer can use the wrapper class in the server to encapsulate a safearray, which
was, for example, received in a function call. The server could create a new instance of the wrapper
using the create function and the safearray as parameter. It could then retrieve whatever content it
desired from the safearray using the members of the object. An example of this kind of behaviour is
depicted in Figure 59.
procedure myServer.getFamily(v: PSafeArray);
var
  myFamily: TFamily;
begin
  myFamily := TFamily.create(v);
  write(myNameFile, myFamily.father.name);
  myFamily.destroy;
end;

Figure 59: Use of wrapper in server

5.1.3.1 Considerations

Working with variant types is known to be a slow affair, and wrapping them up inside an object, which has to be instanced and destroyed, doesn’t make them faster. However, we feel that the benefits gained by keeping the semantic structure may be worth the loss of speed, especially when the structure is deeply nested and complicated.

Every time the structure of the data changes, both client and server will have to be recompiled. We feel that this is more of a benefit than a disadvantage, since erroneous usage of the structure will be caught at compile time, rather than at runtime. Consider the case where the member “name” of the person structure is changed to “married” and the type is altered to boolean. Both the client and the server code illustrated on Figure 58 and Figure 59 would now be syntactically incorrect, and the compiler would catch this. Had we used a safearray without the wrapper class instead, no compile time errors would result from this change, but a runtime error would most certainly occur, when a string was attempted forced into a boolean variable.

This kind of tool would also be useful for converting legacy code that relies heavily on nested structures into COM components. The steps involved would be first to restrict the old structures to using only OLE automation types, then rewrite the functions to pass safearrays instead of pointers to structures, and finally to use the wrapper classes inside the legacy functions.

5.2 IDUN, A CASE TOOL FOR COM DEVELOPMENT

When first embarking on the task of defining a component model, there is quite a learning curve involved, since IDL must be learned. The IDL language introduces a wide array of syntactic elements, many of which mutually exclude each other, and the bindings are not always intuitive. What is probably worse, there is no way of implementing existing interfaces other than rewriting the entire definition, and reengineering of existing type library binaries is not possible. Add to this the fact that it can be difficult to maintain and get an impression of the contents within large IDL files. Each time the definition of a new CoClass is added to an IDL file, a source file with the corresponding method declarations will also have to be written in the implementation language. All these facts make the development process slow and error prone.

What is needed is a tool, which will allow the definition of a component server without having to learn IDL. Such a tool should be able to display the content of component libraries in an intuitive manner, should allow the inspection of the properties of all objects and interfaces within the library, and let them be modified. The tool should have the capability of saving the changes to a disk file in any of the formats that define a type library, IDL or TLB, or as source code.

To be really useful, the utility should then be able to reread existing IDL or TLB files, as well as source, and allow modifications to be performed. It should have the ability to generate skeleton
code in the developer’s preferred target language, without forcing him to bother with getting method declarations right.

Since classes in general and components in particular can be neatly depicted in a graphical manner, the obvious utility to design is a CASE tool. This section describes the features, which our experience with building components has made us think that such a program should contain. The tool will be referred to as if it were in fact fully implemented.

This is not the case, however, but the description has been used as the basis of an actual program, which implements some of the features. Discussed in section 5.2.11, the selected functionality of this program reflects two primary design considerations. First we had to recognise the parts that was impossible to omit if the program should function at all, and second we identified the more interesting features as viewed from a technical perspective.

Since the notion of CASE tools is not at all new, a number of problems pertaining to these particular programs have been identified. In [BETA93] three such problems are discussed: the situation, when the graphical language is richer than the programming language, the CASE gap, and the reverse engineering problem.

The first problem relates to the situation, in which the CASE language was not specifically designed for the underlying language, and is richer than the target language. We do not have this problem, and so it is not discussed any further. The CASE gap problem relates to the distance between the design of the object model and the code that the tool can generate. In an ideal situation, the CASE tool would generate complete implementations, but in practise it will only define so-called skeleton files. The programmer will then have to fill in the blanks himself.

Finally the reverse engineering problem discusses the situation, where a programmer has filled out the skeleton code and wants to change the object model again. Ideally he should be able to reload the code in the CASE tool, redefine the model and save a new skeleton file without loosing the old implementation.

Originally we had planned to develop a tool which would generate CoClass code for the Beta language only, but ended up with a more flexible one that has the potential for creating code in any language. To follow the convention for tools related to the Mjølner Beta System, we picked a name from Nordic mythology. The tool was christened “Idun” after the goddess of youth, who carries the apples that keep the gods young.

5.2.1 Interface Design

The main screen of Idun is where all component design is performed. The screen is divided into three sections, viz. the object pane, the object inspector and the function bar as can be seen on Figure 60.
The object pane represents the entire library, and its structure is mirrored here. All objects in the object pane that represent syntactic elements of the corresponding library are clickable. When selected, an object’s properties are listed in the object inspector to the left of the object pane.

This paratactic relationship is also present between the object pane and the function bar at the top of the main window. Most objects have a button with a + sign on it. When clicked, the sign will change to a – and the object will expand to display its content. Clicked again, the sign changes back to a + and the object retracts and hides its content. From the view menu all entities may also be expanded or retracted at once. This kind of functionality comes in handy when editing very large type libraries with many CoClasses and interfaces in them.

The properties of the object inspector thus vary according to the selected object in the object pane: when the background has been clicked, the inspector shows the properties that apply to the entire library, whereas when e.g. a CoClass has been selected, the inspector shows the CoClass’s properties. When appropriate, the properties can be selected and edited, and the changes will be applied to the selected object’s properties. Thus if e.g. the name of an interface is changed in the inspector, this change will be stored in the interface’s internal properties.

Whenever an object in the object pane is selected, the function bar displays a new set of buttons for adding sub-objects to the selected one. If for instance a CoClass is selected, the user will be able to add a new interface, add an aggregate CoClass, add a contained CoClass or delete the CoClass from the library. In this manner, both the object inspector and function bar are context sensitive with respect to the object pane.

5.2.2 Interface Object Syntax
The object pane is capable of displaying elements for representing a library, a CoClass, an interface, methods, properties, structs, unions, and enumerations. Furthermore syntactic elements for representing aggregated and contained CoClasses are present.

Figure 60: Main screen of Idun

Paratactic: a dependency relationship between two entities, in which the one depends on the other, but not the other way around.
In designing the syntax for the object pane, we have looked at one of the modern standards for describing objects, UML, the Unified Modelling Language defined by Grady Booch, Ivar Jacobson, and Jim Rumbaugh [ROSE98].

Already a number of CASE tools use UML to let a user describe his objects and the relations between them – the OO CASE tool Freja by Mjølner Informatics uses this syntax for example. Components can easily be described in UML – Rational Rose by Rational Software Corporation does this – but the syntax in our opinion is subject to two major flaws: it does not follow the notation for components that is used in the better part of the literature and it is hard to read when there are many entities present in the library.

The UML syntax uses arrows and lines to illustrate the relationships between e.g. CoClasses and interfaces, whereas the other notation displays relations by encapsulation. This latter syntax has the advantage of being much denser and we therefore find it easier to read.

![Figure 61: A simple CoClass](image1)

For these reasons, we have therefore chosen to employ the notation, which reflects the syntax used for drawing components in much of the literature, e.g. [COM97] and [OLE95], as our means of describing objects in Idun.

![Figure 62: A simple interface](image2)

This means that CoClasses are drawn as boxes that literally contain interfaces and aggregated CoClasses while interfaces are drawn using the usual “lollipop” notation as can be seen on Figure 61 and Figure 62.
Aggregate CoClasses expose their interfaces directly without the outer object’s interference. To reflect this behaviour, we have chosen to use syntax depicted in [COM97] and shown on Figure 63.

Figure 63: An aggregated CoClass

Contained interfaces may be used in two different ways: either the outer object reimplements an interface with the same GUID as the contained one, or it exposes a new one. When interfaces are reimplemented in the outer class, they will be depicted as on Figure 64, since they are not exposed to the outside directly, but are subject to the interference of the outer object. The inner lollipop is therefore connected to the outer lollipop. When a contained interface is merely used as a collection of sub-functions, it will be shown using the ordinary interface notation, as illustrated on Figure 62. The notation used in [OLE95] inspired the notation shown on Figure 64.

Enumerations as well as structures and unions are drawn as shown on Figure 65, but since the literature has not defined any standard for these entities, a little more work should probably be put into designing a layout that clearly distinguishes these from CoClasses and each other.
Each syntactic element has its own corresponding actions, which the user can execute on it. The library may have CoClasses, interfaces, enumerations, structures, and unions, and can be cleared.

A CoClass implements a number of interfaces either in a simple manner or by using aggregation or containment to take advantage of existing implementations. This immediately implies that the possible actions that can be performed on a CoClass are adding a new interface and adding a contained or an aggregated interface. Furthermore one may grow tired of a CoClass and so have the option of deleting it.

Interfaces define the functionality of the CoClasses, and it is therefore possible to add new methods or properties to them. Interfaces can also be deleted.

When choosing to base a new component on existing interfaces and implementations there are two ways this can be done: either by aggregation or by containment. Either way the developer will have to decide which interfaces he is interested in and what implementation he will put to use. The natural way to obtain a connection with an existing object is to open a type library and select the interesting parts from here. A type library can reside on a disk file or be found as a GUID in the windows system Registry under the key “HKEY_CLASSES_ROOT\Typelib”.

When an aggregated CoClass has been added to the outer CoClass, the only possible action is to delete it again. A new interface cannot be added to such a class since it only exists inside a compiled binary, wherefore the developer has no control over the implementation. In a similar manner aggregated interfaces, methods and properties cannot be deleted or modified, since the outer object will return the IUnknown of the inner object when asked, and since this will faithfully return pointers to all implemented interfaces of its CoClass, as described in section 3.1.3.

Containment of existing CoClasses may be carried out in two ways: either by reimplementing the existing interface in the outer class, or by creating a new interface, using the inner object’s functions as sub-routines. In the first case, the interface is the same as one that already exists, and so it is not possible to remove methods and properties from it. On the other hand, only some of the interfaces that a contained class implements may be chosen to be exposed, and so the unused ones can be deleted.

When a contained object is merely used as a collection of sub-functionality, and the new interface is a copy of the old one, complete with a new GUID, it can of course be modified as the developer sees fit, and so every part from interfaces to properties can be erased. Additionally methods or properties may be added to such an interface, something that is illegal in the first case.

Returning to the content of ordinary interfaces, namely methods and properties, we find that since these cannot contain any sub-objects, the only applicable function is to delete them.
 Enums, structs, and unions all contain simple members in that these members are not nested any further. The only operations on these structures, therefore, are delete and add member. As for the members, these can only be deleted.

### 5.2.4 Properties of Objects

Every object has a number of attributes that the user can alter. These attributes are the ones that are used when the program will finally have to export IDL or TLB files, and are thus tightly coupled with these kinds of representations. Common to them all is the name property, and many of them contain version IDs, and help information. A full list of properties can be found in [MSDN97] and will thus be omitted here.

The ancestor of an interface, however, is of special interest and deserves further treatment. In the interface inheritance model a subclass does not inherit the implementation of its ancestor, but has to provide implementations for all methods defined in the parent interface. Thus, if the IUnknown interface is subclassed, an implementation of QueryInterface, AddRef, and Release as well as custom methods have to be supplied.

The tool will aid the programmer in selecting an ancestral interface either from a dropdown containing IUnknown and IDispatch or from an IDL or TLB file. Through the selection process, the program will gain enough information about the ancestor interface to be able to create the skeleton CoClass code. As specified by the COM standard, ancestral interfaces can be selected from the transitive closure of all referenced type libraries.

Another interesting property that can be edited in the object inspector is the declaration of a method. In order to be able to describe methods in a manner that is independent of the implementation language, which the programmer will finally employ, these can be edited via a special design window that restricts the allowable types according to the library definitions. As the method is defined, the corresponding IDL syntax will be shown below.

### 5.2.5 Interface definition input/output capabilities

In order to integrate Idun with the existing lot of component implementations and definitions, it can read the most basic formats that describe libraries, viz. IDL and TLB. Inside files of these types are all the details necessary for filling out all properties of Idun’s visual objects. Type libraries can be loaded from TLB files, but indeed also from some component libraries, i.e. the DLL or EXE files that contain the components. Finally large numbers of type libraries populate the Windows Registry, and Idun therefore provides a means for reading these also.

The TLB format is an architecture dependent format, which means that a TLB file that was compiled on a Windows machine cannot be used on e.g. a Macintosh. IDL, on the other hand, provides an architecture independent way of describing the content of a COM library, and the tool also has the capability of reading this format by default.

Restricting the loading of type information to IDL or TLB files is a severe limitation. Other formats exist that describe information which could be rendered into a type library or CoClass skeleton code, and as such the tool include an extension mechanism that will let it read other formats also.

Most of the information needed to build a type library could for instance be read from a Java source file, which happened to contain Java definitions for interfaces and CoClasses. It is worth to note, however, that source files for implementation languages rarely contain all the necessary properties,
needed to describe every intricacy of every interface or CoClass. The rough outline of the type information, which was originally used to build the source file may, so to speak, be retrieved.

In Microsoft’s J++ Java compiler, for example, the CoClass skeleton code could be used to find out which interfaces the CoClasses implement and use the autogenerated code to get the definitions of interfaces, structures, enumerations etc. The GUIDs and IDL attributes are not part of the Java code, however, so getting back to the original type library still involves some manual labour.

Figure 66 lists a number of possible languages or file types that may contain descriptions of type libraries.

![Diagram](image)

**Figure 66: Input and output capabilities**

When a COM library has been defined, the work can be saved in IDL or TLB format. Furthermore a skeleton implementation of the CoClasses in the library can be generated and be saved e.g. as C++ or Delphi code. Idun can also save and load a library in its own internal format, which is needed when it has aggregates or contained CoClasses. These entities will namely not be visible in plain IDL code, and so some of the logical structure will be lost if the library is simply stored as IDL in such a case.

Depending on whether or not the interfaces are restricted to using only OLE compatible types and will use type library marshaling, it may be desirable to generate IDL code which, when compiled with MIDL, will create the source files for a proxy/stub DLL. There is a slight twist to the order of the syntactic elements of such an IDL file, as described in section 3.1.5, and this is the reason why we think the proxy/stub source should be optional.

### 5.2.6 Editing utensils

On some occasions one may want to implement a specific interface that someone has previously defined. This is for example the case when an event sink for getting updates from a COM server when some interesting property changes or when an event occurs has to be implemented. The event sink interface that Idun provides is described later in this chapter and serves as a good example of this situation. The programmer should be able to select a specific interface from another type library and get the structure for it immediately. Similarly there may be times when just one of the many CoClasses of a library is the target of interest.

One way to support this kind of flexibility in the CASE tool is to use the containment facilities as described earlier, which could be adjoined with drag-and-drop capabilities. An interface in one CoClass could then for example be selected and dropped on the library or all methods from one interface selected and dragged to another. Of course it can also be necessary to copy attributes from
one entity to the other, and so the cut, copy, and paste standard user-interface functionality is part of Idun.

Some objects will be illegal in some contexts, which means that it is not possible e.g. drag a CoClass into an interface or a method onto the library canvas. Drag-and-drop immediately allows for context sensitivity by letting the program change the icon of the mouse when dragging to indicate acceptance or rejection, whereas cut and paste does not. In the latter case the context information is provided via an error dialog box.

Sometimes one may want to build a library from the parts contained within other libraries. Since Idun has the support for inter-process drag-and-drop or cut and paste between two or more running instances, this is easily achieved.

The bottom line is that Idun provides a fine granularity that allows import of all parts of existing libraries.

5.2.7 Extending the capabilities

Once the library definition is complete, an actual implementation of the CoClasses in it can be made.

The choice of implementation language should not be restricted in any way, and it is therefore necessary to have a tool that is highly flexible, and will allow the extension of its capabilities as needed.

Since the extensions are likely to be defined after the final build of the CASE tool, and possibly in an environment where access to the source of Idun is not available, why not let the extension modules be COM components? The extensions should have access to the library that is currently contained within the CASE tool, and this suggests that the tool itself should also expose interfaces via COM. These interfaces could in turn be utilised by the plugin to generate source for the target language or to build the visual equivalent of some source file.

![Diagram of data flow between Idun and plugin](image-url)
Figure 67 shows the data flow between Idun and an extension. Before any communication can take place, the CASE tool must know that an external module is present, and thus any plugins must implement the IRegistration interface.

When a user wants to extend the capabilities of a running instance of Idun, he can one of two things. Either he uses the tool to select the plugin module via a file browser or he writes an installation program for his module, which launches Idun through automation and registers the plugin via the IPluginManager interface.

Either way, Idun invokes the register method of the CoClass for the language plugin as illustrated by the arrow at 1. The plugin returns the IRegInfo interface, which Idun can use to obtain information about the plugin such as a human readable description and a name, as shown at 2. The plugin will then be registered both with the SCM and internally with Idun, and the next time Idun is loaded, it exposes the new capability to the user.

When the user wants to generate source for a specific target language, he selects export from the menus and picks the language of choice. This causes Idun to ask the SCM for a pointer to the plugin’s IExporter interface, on which it will then invoke the export method to get the source as shown at 3.

Inside the export method, the plugin takes control and asks for the IxxxInfo interfaces of Idun, uses these to generate code as illustrated by the arrows at 4 and finally returns an ISourceFiles interface, terminating the call to IExporter::export.

Through this interface Idun will now retrieve the number of generated source files and iterate over these by calling “getChild” and getting a corresponding ISource interface for each file. The ISource interface has properties for filename, description, and linecount. Idun will use the last property to iterate over the lines in the source file, by invoking “getLine” to get each. The relationship is illustrated by the arrows at 5.

Idun will finally display a dialog box with the filenames in it, asking for a directory to save the files in.

interface ILibraryInfo : IDispatch {
    Property Control: VARIANT_BOOL;
    Property HelpContext: long;
    ...
    Property UUID: BSTR;
    HRESULT _stdcall ChildCount([out, retval] long* ChildCount);
    HRESULT _stdcall getChild([in] long index, [out, retval] IIdunInfo** index);
};

**Figure 68: Pseudo IDL syntax for ILibraryInfo**

When a plugin starts generating code, it must first ask for the outermost IIdunInfo interface of Idun. This interface simply has a property for the name and the type of the entity it represents. Using the type, the plugin can then QueryInterface after the interface that represents the rest of the entity. The first IIdunInfo interface the plugin gets will always have the type “library”, meaning that the plugin can query for ILibraryInfo.

As Figure 68 shows, this interface holds attributes for the library as well as a property that gives the number of entities in the library. The previously mentioned “getChild” method can then be used to iterate over the children of the library. Again IIdunInfo interfaces will be retrieved, and the type
property can then be used to query for the correct interface. This is true all the way down in the visual hierarchy of Idun.

IInterfaceInfo interfaces have a special attribute that indicates whether they are to be implemented in the CoClass as aggregates, contained or in the ordinary fashion. When a CoClass contains an aggregated interface, the plugin should provide an alternative implementation of IUnknown that delegates calls to the inner object’s IUnknown interface when appropriate. Furthermore the inner object should be created when the outer object is instanced and released when the outer object dies. Contained interfaces should make the plugin generate a skeleton for the CoClass, which will launch the contained interfaces on start-up and release them when the server shuts down. Furthermore it could provide a default implementation of each function within the contained interfaces that simply delegate the calls to the contained object.

The communication, which takes place when exporting code, is similar to the one that goes on when a plugin is used for importing library information. Instead of requesting the IExporter interface, Idun will ask the plugin for its IImporter interface as illustrated by the second arrow at 3 on Figure 67 and invoke the import method on it, passing a filename to be loaded as parameter. The plugin can then load the file and perhaps subsequent ones that it needs for building the visual hierarchy in Idun. The properties of the IxxxInfo interfaces can be written to as well, and the member methods “Addxxx” can be used to get a pointer to a new IIdunInfo interface that represents the new visual entity. An interface, for instance, will have methods “AddProperty” and “AddMethod”, while an enumeration will have “AddItem”.

5.2.8 Plugin implementation issues

Another strategy could have been employed in the design of the interaction between Idun and her plugin modules. The COM specification [COMSPEC95] defines the notion of an event sink interface, which is implemented by the client, but used by the server. In our terms, the plugin could have implemented a special “ParserEvents” interface, which it could then hand over to the CASE tool for use when source for a particular target language was to be generated. The event interface would have methods for generating code for each syntactic category i.e. library, CoClass, interface etc. Whenever a user would choose to export code for that language, Idun would make a recursive descent through the content of its currently displayed library and call the appropriate functions of the event interface as it happened upon elements of the different syntactic categories.

In the design of the interaction process, we have looked at different implementation languages, in order to expose the similarities. Our first idea was that this process would lead us to the definition of a general event interface, which would embrace all the peculiarities that could be expected from different languages. It became clear, however, that the expected similarities were diminishing when compared to the differences, and that the event interface would be very difficult to define. One language would require a source file for each CoClass, while another would supply a default implementation of IUnknown or IDispatch. Some languages require forward declarations of methods, while others don’t, and finally there is a general disagreement of where to put the declarations of CoClasses as well as methods.

By giving the client the full responsibility for generating the appropriate source code, we have constructed a very open model that is highly flexible. The programmer is not bound to implement a fixed set of functions other than the ones needed for initiating the communication and for letting Idun access the generated code, and can do as he pleases with regard to getting the job done. Of
course he will need to acquaint himself with the IxxxInfo interfaces and their relations to the visual equivalents of Idun, but once he has learned this, the rest should be manageable.

One way of structuring the source for such a plugin, would be to take a long look at the files that the target compiler expects for component definitions. The dynamic parts of the source would then be pin-pointed and a function, which would print the static content from top to bottom, while recursively invoking other methods for getting the dynamic content at the appropriate times, be built. Example source code for a plugin ought to be shipped with the final tool for easing a programmer’s job.

5.2.9 Reengineering

By allowing Idun to read and write both IDL and TLB files, it can effectively reengineer existing libraries. In order to provide full reengineering support the tool can also maintain the bodies of methods that have been filled out by the programmer in the skeleton source files. This allows for enhanced consistency between the graphical representation and the source code, and addresses one of the general problems with CASE tools, the CASE Gap, as described in chapter 5.2.

To facilitate this, the plugin module must define its own strategy for identifying user-defined code in an auto-generated file. When IImporter::import is invoked by Idun, the plugin is free to store the appropriate source pieces that were written by the user inside the IMethodInfo and IPropertyInfo interfaces it creates. This can be done by requesting a new empty ISourceBody interface from IMethodInfo or IPropertyInfo via the “addBody” member method. Idun will allocate a new ISourceBody and return it to the plugin, which can then set the language type by passing its own GUID, and add the source code lines.

If the user at some point in time chooses to export code in the same language again, the IExporter interface can restore the custom implementation by inserting the old pieces into the new skeleton file. It can use the GUID attached to each IMethodInfo’s or IPropertyInfo’s ISourceBody interfaces to check that the source is indeed written in its target language. Note that Idun allows a library to have many implementations of one method or property opened at once.

If the plugin needs to save additional code or information, it will have to provide its own private mechanism for this. It is hard to foresee what needs a plugin may have in these respects, and so a feature for storing arbitrary information has been omitted from Idun.

One strategy for letting the plugin get the method bodies is to mark the sections of the file where the user should fill in the blanks with special comments that are easy to identify. The plugin can then search through the old skeleton file, locating and copying the lines in between the comments. Of course this strategy requires that the user does not fiddle with the rest of the source, and in particular that he leaves the comments alone. Another strategy is to build a complete parser, but this does of course require a lot more work.

Figure 69 shows how reengineering works in Idun. The top of the figure shows the situation where Idun imports source code and, via the plugin, gets new content. At 1 Idun calls the importer passing a filename and making the plugin open the file on the disk at 2. Next the file is read by the plugin at 3 and the visual representation drawn in Idun using the IxxxInfo interfaces at 4. In the process of drawing the library, the plugin may store source code inside ISourceBody interfaces as illustrated by the arrow at 5.
When the user exports code, the plugin will retrieve data from Idun including any source inside ISourceBody interfaces and return control to Idun. This will then request the ISourceFiles and finally save them to disk.

Note that a plugin can not expect to be loaded in memory between an import and a possible export, and so should write any custom data that it wishes to reuse in a persistent storage.

5.2.10 Integration with third party tools

Idun operates at a component level meaning that it allows the developer to work with the definition of individual components rather than with the relations between them. Restricting the tool in this manner was a clear design consideration from the beginning, since there exists a vast array of object oriented CASE tools for defining object relations already.

In order to allow the extension of the capabilities within these OO editors to include working with COM objects also, Idun can in fact be fully controlled through the interfaces listed on Figure 67. Via these, an external application can launch the tool, insert a CoClass and a few interfaces and save the entire model as a binary TLB file. All this can take place behind the back of the user by launching Idun as a hidden application or it can be shown to him.

Imagine a CASE tool that displays a number of objects, some of which represent COM components. When the user chooses to edit the properties of such an object, the tool launches Idun, programmatically inserts CoClass and interfaces and displays this to the user. When all edits have been carried out, and the user wishes to return to the external CASE tool, there will be a need for some kind of notification mechanism, which will allow Idun to notify her master that this is the case.

Idun defines an event sink interface that external applications can implement in order to get event notifications from the tool when controlling it through automation. Figure 70 shows the process of connecting such an event interface to Idun: at 1 the controller must ask for Idun’s IConnectionPointContainer interface and then get the desired connection point from here. Now a pointer to the event sink can be passed onto Idun through the IConnectionPoint’s advise function at...
2, and finally at 3 Idun can invoke the methods on the event sink when appropriate. The event sink schema is discussed in section 3.3.2 and thus omitted in its full detail here for brevity.

The IDunEventSink interface defines methods that will be invoked when Idun is about to close and when different actions, e.g. the insertion of interfaces or methods, are performed on the model. This allows the other CASE tool to reread the configuration in Idun before closing it and to take special action when a user changes the model.

![Figure 70: Automating Idun from other applications](image)

Idun supports in-place activation, a term that covers a program’s ability to be controlled within another window of a host application. This ability can be added by implementing a number of interfaces, all of which are part of the OLE technology as described in [OLE95] and thus omitted here. In-place activation will let the host application expose the functionality of Idun as if it were in fact part of its own implementation.

### 5.2.11 Implementation

The description of Idun, which is presented in this paper, is not fully implemented as mentioned earlier. It has, however, been used as the basis of an actual implementation. Some parts of the original design were left out because a complete implementation was unrealistic given the time bounds set by the ultimate deadline of our thesis. In the implementation we have focused on the aspects that we found to be the most interesting, and that would take us around most of the issues involved in creating the entire tool.

Figure 60 is in fact an actual screen shot of the implemented tool, and as can be seen, we have chosen to build most of the graphical interface. The properties of an interface have been omitted since these are nothing more than get- and set methods and thus more or less correspond to the methods of an interface. Apart from this, only modules, unions and structs are not supported a fact that has little consequence for the usefulness of the tool.

Methods will not be edited through a window that embraces the logic of the current library, but the user will be forced to write them in plain IDL inside the Object Inspector. When this facility has been omitted it is due to the fact that it merely represents a convenience more than an interesting topic.

When selecting the ancestor of an interface, there are no restrictions to what can be written. Building a list of variable and interface names was left out because parsing type libraries was investigated in conjunction with the type library import module.

The tool will read type library binaries either from Registry or a disk file, but not support references to other libraries. It will not generate TLB files, but will faithfully create IDL code. We chose this strategy because it is easy to get from IDL to TLB via Microsoft’s MIDL compiler, which can be
obtained from the Web for free, and because reengineering of library descriptions is thus possible. This last fact makes the tool much more interesting to work with.

Our version of Idun supports export plugin modules, and we have actually implemented two of these in different languages in order to test it thoroughly. The first plugin is for J++ and is written in that language, while the second will generate code for Borland’s Delphi 3.0, the language in which it is itself written.

The only restrictions of the plugins is that they cannot import code, but this restriction is actually imposed by the fact that our implementation of Idun does not support the “Addxxx” methods of the IxxxInfo interfaces. On the other hand, skeleton code that supports both aggregation and containment can be expected. This functionality is one of the really interesting parts, since – at least – aggregation is a quite hairy feature to implement for intermediate COM programmers, and since they both facilitate a kind of pseudo implementation inheritance through component reuse.

The tool supports only one of the two types of containment, namely the reimplementation schema. There is no profound reason for choosing this type rather than the other, but we have only supported one of them, because implementing the other, also, would add nothing new to our understanding of the tool and be academically trivial.

As an extra feature that eases working with Idun, drag-and-drop capabilities within a single instance of the program has been implemented. Cross application drag-and-drop was left out along with cut-and-paste functionality because that was not the focus of our investigation.

External automation has not been fully implemented because the issues involved in doing so are little different from implementing the plugin capability. An external program may read the content of an open library in Idun, but not write to it, and nor force the generation of source code. Likewise the Event interface cannot be connected.

Finally the program does not support in-place activation because implementing this feature was far too time consuming given the amount of time available.

The full source and binaries of Idun and plugins can be installed from the companion CD or be downloaded from http:\www.daimi.aau.dk/~hardcore\speciale\idun.zip.

5.3 SUMMARY

Working with COM is an involved task, which requires profound knowledge of many aspects, before even the simplest component is ready for use. To remedy this problem, we have identified a number of tools, which could help the programmer in his way from modelling to implementation of components.

One utensil that may be built is a tool for creating the structure of a program, which uses components as building blocks. Such a tool is quite similar to the CASE tools that already exist for defining the structure of an object-oriented program, and therefore building a new one seems like overkill. Instead existing tools should be extensible to include the notion of components also, thus effectively allowing the designer to choose whether a class should be internal or external to the program.

A highly specialised tool is the Free-Threader, which enables legacy components to run in a multi-threaded environment. By wrapping up existing procedure calls in thread safe code, this program ensures that shared resources will not lead to inconsistencies. In practice, however, one would
probably benefit very much from rewriting the entire component instead, since such automatically generated code will seldom be efficient.

In another obscure corner of the COM toolbox, lives the **Semantic Safearray Wrapper**. This program allows the definition of complex data structures like the ones supported by the IDL "struct" keyword; only the Semantic Wrapper will do this for automation interfaces, where the struct keyword is not supported.

By building skeleton code that effectively wraps a safearray into an object, complete with nested properties, this tool lets the programmer of both client and server benefit from the bliss of automatic marshaling, while retaining the semantic information that property names implicitly contain. Of course there will be an extra overhead involved by the level of indirection in the access to the actual data. On the other hand meaningful exceptions may be thrown, when a program tries to fit a value of the wrong type into a property of the structure.

Building COM components requires knowledge of IDL and the interface mapping of the implementation language. Many of the implicated procedures can be automated by using a suitable **component CASE tool**. Such a tool will benefit from having a graphical user interface that can aid the development of new components as well as the reengineering of old ones. All properties of the elements in a library must be editable, and the tool should support aggregation as well as containment as a means of reusing legacy components.

The COM CASE tool should not be restricted to one target language, but should support the development of components in **any language**, just as the features of the tool should be programmatically available for third party programs for maximum integration.

Working with the implementation of Idun, and actually building a number of components, has made us confident that there is a need for such a tool. Not only does it speed the development process enormously, but it also improves the breadth of view, making it much easier to get a quick impression of library structures.

The example application of section 3.3.2 was actually created after the completion of Idun, and since the event sink interface existed in a predefined library, using our tool to generate skeleton code really sped up the process. The interface contained quite a number of method declarations, and adding all these by hand takes a very long time compared to using Idun.

Apart from the Explorer Spy, a number of test applications have been built, ensuring that more subtle functionality of the plugins, such as aggregation and containment, actually worked. As it is the skeleton code generated by both Delphi and Java plugins will let components aggregate and be aggregated automatically.

The fact that, once defined, a library may in fact be exported to any target language for which there is a suitable plugin is really nice. The plugins that have been implemented for J++ and Delphi will generate all the source files needed for an entire project including any make files or project source. This means that all the developer has to do is fill out the CoClass skeletons and write “make” on the command line or, in the case of Delphi, to compile the project in the IDE. Idun is even superior to Delphi’s Type Library Editor in that it allows the construction of source files based on existing libraries and exports code for aggregates and contained CoClasses as well.
6 CONCLUSION AND FUTURE WORK

This thesis has treated different aspects of component software. Firstly it has defined a set of criteria for evaluating component models and used it on commercial products. Java Beans and CORBA were treated lightly, while COM and its siblings were discussed in full detail. The criteria were used to gain an understanding of the strengths and shortcomings of COM, and CORBA served to contrast solutions with alternatives.

Shifting focus from the intricacies of component models to IDEs that support component based software development, chapter 4 introduced yet another set of criteria, this time for IDE evaluation. The criteria were used to evaluate the COM development abilities of the Delphi environment and lead to the completion of the Taxonomy document.

As a direct consequence of the intimate knowledge of COM gained through the investigation in the early sections, and the thorough examination of the Delphi IDE, the last chapter introduced a number of supportive tools for COM development. Much attention was given to the CASE tool Idun, which supports the construction and reengineering of COM libraries in potentially any implementation language. The need for such a tool was motivated, and a concrete implementation presented.

The most important points of the four sections in the thesis will be summarised below.

6.1 CRITERIA FOR EVALUATING COMPONENT MODELS

To conduct an investigation of possible criteria for component evaluation, one must look at the specific needs that the concept targets. There are a number of issues at play here, namely that of providing a language and platform independent technology, which supports distribution in a transparent manner, and allows for non-redundant reuse.

6.1.1 Language Inspired Concepts

These requirements have resulted in the definition of several commercial models, all of which base their foundations on class based object-orientation and more general programming language constructs. The first set of evaluative criteria will therefore pertain to the important issues from these foundations, in the thesis referred to as the language inspired concepts.

The notions of object, class, and interface fit well into component models with the detail in mind that interface and implementation must be separated. In a similar manner, instantiation and destruction mechanisms can easy be generalised and incorporated into the model. This is also true for providing a means for initialising and finalising components.

Turning to inheritance, the first troubles begin because of the inherent black box nature of components. It seems difficult if not impossible to define a way by which true implementation inheritance can be provided. This can be circumvented by allowing interface inheritance and delegation, but the strict sub-typing of inheritance is lost\(^\text{26}\). Calls can be delegated to any object, not only ones in an inheritance hierarchy.

\(^{26}\)There is a standing discussion of whether delegation is superior to inheritance. In [DEL87] a proof that the techniques are equally powerful is given, and the main difference between the two techniques for sharing is the strict requirements on the types for inheritance.
Encapsulating the behaviour of components is an inevitable part of CBSD, again because of the black box nature. It seems reasonable to simply support public and private members, since more subtle access specifiers tend to pertain to implementation inheritance.

Just like ordinary OO objects, components should support a means by which their interface can be accessed and introspected at runtime, and by which they can communicate. They must support the ability to be composed into new components in order to facilitate reuse and OO modelling, and should support polymorphic behaviour on interfaces. This latter facility allows components of the same base interface to be treated in a uniform manner.

Of the issues that pertain to programming languages in general are providing a homogeneous manner by which exceptions inside components can be propagated back to the client and handled there. Similarly an event mechanism should be defined to allow components to call back to their clients, when some important incident has occurred.

Since components can be expected to shut down on occasion, the model should equip each with the ability to be stored persistently. The models should conform to the design principles of persistence independence, persistent data type orthogonality, and orthogonal persistence management. This ensures language independence and maximum flexibility.

Finally the model must not hamper the definition of concurrently executing components, but specifying guards and sentinels must be left to the implementation language.

### 6.1.2 Data Related Concepts

The fact that a profound requirement for component models is that they should support non-redundant reuse, means that their nature must be somewhat similar to that of runtime libraries. These component libraries are external to their hosts, and since many clients can connect to the same component library at once, issues that pertain to data management arise.

Revision management is an issue, which component models must also address in order to provide a proper way of supporting software evolution. The introduction of new versions into running systems should be possible in a transacted manner that allows an administrator to undo his changes and restore the system to a consistent state. There must be a way by which clients using an older version can seamlessly upgrade to a new one without shutting down.

Transaction management is also needed when more clients connect to object servers that access some shared data. A global service, which could enhance the system with transacted capabilities, would therefore be necessary.

### 6.1.3 Distributed Concepts

Targeting the client/server market, component models must provide a means by which distribution is easily achieved. There should be no limitations on the architectures that a component model should support; in effect it should be fully portable. This should also be true for the components themselves, and this introduces the difficulties pertaining to activation and execution.

Ideally a component model should support both local and remote activation and local as well as remote execution. In reality, moving components across architectures is not realistic unless the components themselves are always interpreted by some runtime machine.

Retrieving object references should be completely transparent to the programmer, and a component model should therefore provide a naming mechanism, which has a location transparent facility for
activating and instantiating components. One or more network-global name-servers could handle this.

At the same time these servers could also take care of security by ensuring that only certified clients gained access to restricted components, and to balance network load by replicating components between machines and successively redirecting clients. Since such dynamic load balancing is very difficult to implement, a more static approach is acceptable.

6.1.4 Peripheral Concepts
Apart from the obvious requirements that component models seek to fulfil, there may be additional ones that are more or less important, depending on the other specific design goals of a given technology. The thesis has not made an attempt to exhaust the list, but briefly introduced two of the more common concepts.

The most important one of these is providing a framework for equipping components with a graphical user interface. A standard way of doing so will ease the use of such entities. A similar concept is providing a standard for graphical manipulation of the component’s internals, but again this can be done programmatically, and it is easy to support such functionality when the model allows components with a GUI.

6.2 Commercial Models
A number of commercial models are available for CBSD on various platforms. This thesis introduced CORBA as a model which targets and specifies solutions to all of the criteria we have listed. As a model, CORBA is very strong, but it has one inherent problem namely that its commercial implementations are not necessarily compatible. This means that ORBs from different vendors cannot communicate lest they both implement the IIOP protocol on top of their ordinary communication mechanisms. Additionally one should not expect a commercial implementation to fully support all details of the CORBA specification.

The Java Beans model can only be used with the Java language, and so fails on one of the most crucial requirements for component models, namely language independence. Furthermore only the most basic requirements are treated, and the difference between ordinary Java classes and Beans is so insignificant that we feel Java Beans does not deserve to be called a component model.

Microsoft’s component model is made up of different technologies, COM, DCOM, and OLE, which form a complete model when grouped together. The thesis provided an in-depth treatment of COM and DCOM, as well as the component related issues from the OLE technology.

6.2.1 Language Specific Concepts
The COM model defines the basics, and treats concepts like object, class, and interface as the criteria prescribe. Instantiation is done through Windows API functions, and initialisation and finalisation are supported via the ClassFactory and the Release method.

COM has single inheritance on interfaces, but allows classes to implement any number and use aggregation and containment to expose functionality from other classes. Similarly encapsulation and polymorphism are as expected, and composition can be facilitated by instantiating contained components within the constructor of the outer component. The aggregation mechanism provides an alternative, where aggregated interfaces are exposed directly as if they were native parts of their host class.
Components can be introspected via type libraries, but these need not be present in all components, which is a nuisance. Furthermore they may communicate by accessing each other as would an ordinary client.

The COM model has no exception mechanism, but relies on integer values to propagate error information back to their clients. OLE and dispatchable interfaces must be used to get a real exception mechanism, and even so that one is difficult to work with. The exception mechanism should have been part of the core model.

On the other hand, the event mechanism in COM through connectable objects is easy to use, but again it is not part of the core model. This latter critique also pertains to the persistence model, which is something that the programmer must fully support himself. There is only little help to get and no automatic persistence mechanism exist. On the other hand the model is simple and conforms largely to the three design principles. Relying on the Structured Storage technology, however, it is not efficient.

Creating concurrent components in COM is no different from creating concurrent Windows programs in general. The concurrency mechanisms of the operating system are fully available.

6.2.2 Data Related Concepts

An area lacking a lot of work in Microsoft’s component model is the data related concepts. Versioning of libraries is very poorly supported, and the revision management of the individual components must be done manually. It is not possible to commit new versions in a transacted manner, but everything must be handled manually. The fact that interfaces must never change, ensures, however that old clients will not cease to function when a new version is introduced.

A recent add-on, which will be part of the component model with NT 5.0, the MTS server supports transaction management. Unfortunately this product must be paid for separately at the time of writing.

6.2.3 Distributed Concepts

COM components are tied to the architecture on which they were built, but through DCOM it is possible to use components on other architectures with a given client. The model itself can and has been ported to other platforms.

DCOM allows components to be instantiated and executed both locally and remotely, but for the mixed case, ActiveX components are needed. Referring to components is done through GUIDs and the Registry, but in a highly inflexible manner where the target machine must be specified. There is no global name server, but location transparency is ensured through the Registry entries.

This way of organising the naming capabilities makes load balancing a decentralised issue, and will not be handled automatically. In a huge network with many components such a strategy might well be too difficult to maintain.

As the model has been defined basically for the Windows NT system, the security mechanisms of the operating system are utilised. This equips DCOM with a strong access facility, which unfortunately is very hard to use. Additionally components may themselves provide a means for restricting their use via the IClassFactory2 interface.
The distributed version of COM is where the model really differs from CORBA, which in contrast was originally devised to be a component model for distributed environments. DCOM is much weaker than CORBA in this respect, and still needs a lot of work to scale well in large networks.

6.2.4 Peripheral Concepts

The OLE technology embraces a lot of different features, building directly on top of COM. these concepts as well as those provided by the ActiveX framework make up the existing peripherals of Microsoft’s model. Graphical user interfaces can be attached to ordinary out-of-process servers, but not remoted across networks. Additionally ActiveX components may have a GUI and support property pages for graphical introspection.

Microsoft’s component model targets many of the criteria we collected, but some with more luck than others. It was developed over a long period of time, and evolved from something that was not even a component model. As a consequence it suffers from having been patched and introduced to add-ons, which did not always simplify the model. It is highly functional though and scales well from small lightweight components to full-fledged automation servers, but a recapitulation on the side of Microsoft’s to rebuild the entire model would be beneficial.

6.3 Evaluating IDEs

Developing a strategy for revealing how well an IDE supports COM development was done by defining an example application and a number of tasks that would be executed. We find the strategy good because it forces all environments through issues that they may or may not support, and thus inhibits that they are evaluated on false pretences. The Pilot Application tasks did not target enough issues, though, and many of the questions to be answered later were left open.

The investigation would definitely have been more useful, had the tasks been adapted more to the final Taxonomy document. On the other hand, it is our impression that the Taxonomy document treats the better part of issues that let a developer choose the IDE that best suits his needs.

Putting Delphi through the evaluation was an interesting experiment that revealed both strengths and weaknesses of that particular environment. Central to all COM development in Delphi is the Type Library Editor, which provides a practical way of defining the structure of COM libraries. The editor has some shortcomings, but nothing that could not be fixed in a later version.

Working with and editing the code of components in Delphi is a bliss because of the inherent reengineering capabilities provided by the Type Library Editor. The worst weakness seems to be the fact that the tool must be cheated in order to have it implement predefined type libraries.

By wrapping up COM specific details, the Delphi COM framework allows the developer to focus on the functionality of his component rather than on basically irrelevant issues. On the other hand, the fact that full access to API functions and COM specifics is given, lets the seasoned programmer extend the existing framework and even build his own.

Issues that are not dealt with by the IDE include automatic code for containment and the outer of an aggregate. It would not have been a hard task to provide such a facility, but the reason for leaving it out is probably connected to the fact that the Type Library Editor was built according to COM specifications. Had the Delphi team chosen to construct the editor to be more focused on design instead of simply providing a graphical view of type library content, this kind of functionality would have been easier to fit in.
The last major shortcoming of the IDE is the inability to provide a proxy/stub component for standard marshaled interfaces. Having to rely on MIDL and a competing C compiler is not something an IDE should need to do.

Reviewing the draft versions of the Pilot Application documents for all four IDEs revealed that Delphi is strong and will be a good choice for most purposes. There are situations when VC is superior, but in general it is much easier and as powerful to use Delphi. VB is good for building automation servers and nothing else, while VJ has a huge problem in having to supply its runtime engine. The latter two IDEs are weaker than both Delphi and VC with respect to the overall range and strength of the COM support, and we would not conduct a programming project with these as the foundation.

### 6.4 Tools

Following the IDE evaluation, identifying issues that relate directly to the COM model, and which could be simplified by using the appropriate tool was the first task, which was undertaken. A number of highly specialised utensils were introduced, and the Semantic Safearray Wrapper discussed in some detail. Common to most of these tools is that they have a narrow public, but would be quite useful in some situations.

The main topic of the tools chapter was the CASE tool Idun, which was treated in full detail. Based on shortcomings in the Type Library Editor and on features we missed in our practical work with the COM model, this tool was designed with flexibility in mind.

The main goal was to describe a utility that would allow developers to define component libraries visually, export descriptions in any language and reengineer any code that could be interpreted as defining a component library. By providing the description of the entire user interface and a framework for plugins, we feel that this goal has been quite successfully achieved. This feeling is of course strengthened by the fact that two language plugins have been coded, and most of the tool actually implemented. Using the tool ourselves to build test servers in both Delphi and Java ensured our claims that such a tool would speed up the development process.

It is perhaps unfortunate that we did not have the time to implement all aspects of the tool, and in particular we feel that reengineering would have been very interesting to work with. When implementing the export modules, minor problems in our original model were discovered and handled, and possibly other irregularities would also appear when the reengineering facilities were in fact implemented.

In a similar manner it could have been very interesting to have the tool actually be launched and used by another CASE tool to enhance that tool with component capabilities. Again the robustness of our design would have been tested this way. It is worth to note that the current implementation of Idun can in fact be launched through automation, but the properties of the visual entities can only be read, since only the export facilities have been constructed.

### 6.5 Future Work

It is our impression that components have come to stay, because they embody an abstraction, which is superior to alternative solutions such as equipping runtime libraries with distributed capabilities. This said, it is no secret that the evaluation mechanisms, the component models, and the tools discussed in this thesis are, but in their infancy. Much work is still needed in the area of CBSD, and quite possibly new inventions and demands will spawn criteria as well as actual implementations that no one are currently thinking about.
Had we had more time, completing the implementation of Idun would have been given high priority. Since the tool is quite general in nature, the specification could also be used with other component models, by changing a few of the details as appropriate. By moving up yet another level of abstraction, building a tool that could generate skeletons for any component model is perhaps even a research issue.

This utility could then provide facilities that would transparently build bridges across the component model borders, and let e.g. CORBA components talk to COM components without the designer needing to bother with the details.

Another interesting research area which we have not touched upon is that of CBSD. Building component based software is different from ordinary application construction. There are a number of new issues that must be dealt with, and possibly the design process must be augmented to incorporate components also. It may not be all classes that are suitable for being components, and so pinpointing and designing classes for being externalised is also an important issue. The difficulty lies in making the components general enough to facilitate reuse without destroying the design, while at the same time maintaining a degree of specialisation that does not seriously compromise performance.

Rumours have it that Microsoft will release the next generation of COM sometime in 1998. Termed COM+, this model should supposedly target some of the shortcomings that we have identified in this thesis. In particular the DCOM part has been given some attention, and relating COM+ to the criteria could therefore definitely be interesting.

At the same time SUN continues to enhance and promote Java Beans, and who knows: it might just become a real component model some day. The CORBA community is also flourishing and the next version of Delphi will see Visigenic’s CORBA implementation as part of the IDE. Also, the largest CORBA vendor, Iona Technologies have just announced the availability of their bi-directional COM/CORBA bridge OrbixCOMet. This bridge will let developers access CORBA components via COM interfaces and vice versa and from the IDEs of their choice.

It appears that we have not seen the end of CBSD yet, and the developers of the next millennium will definitely build component based software.

---

7 Keywords

activation ......................................................... 17
aggregation .................................................. 36;50;87
automation ................................................. 56
BeanInfo interface ............................................ 26
binding
  early ................................................................. 59
  late ................................................................. 59
  very early ......................................................... 58
CASE tool .................................................. 108
CBSD .............................................................. 1
class ........................................................... 6;49
Class Identifier ........................................ See CLSID
ClassFactory .................................................. 34
CLSID .............................................................. 34
CoClass .......................................................... 30;99
  aggregated .................................................... 99
communication ............................................. 12
component ..................................................... 1;5
Component based software development ........................................ See CBSD
component CASE tool .................................. 109
composition ..................................................... 12;24
Computer Aided Software Engineering See
  CASE tool
concurrency .................................................. 15;25;27
connectable object ........................................ 61
containment .................................................... 36;50;87;99
customisation ............................................... 19
DCE RPC ......................................................... 42
delegation ....................................................... 9
dispID ............................................................. 57
dispinterface .................................................... 57
DLL .............................................................. 30
dual ............................................................... 57
Dynamic Link Library ................................ See DLL
dynamic replacement ....................................... 16
encapsulation ............................................. 10;23;26;40;50
event sink ..................................................... 61
events ......................................................... 12;24;27
exception .................................................... 12;24;27
  handling ...................................................... 63
execution ....................................................... 17
external ......................................................... 5
finalisation .................................................. 8;23;26;49
Free-Threader ............................................ 108
global name server ........................................ 27
Globally Unique Identifier ................................ See GUID
Graphical User Interface ................................ See GUI
GUI ............................................................. 19;27
GUID ............................................................ 31
IDE ............................................................... 70
IDL ............................................................... 31;40
  CORBA ......................................................... 23
immutability .................................................. 32;49
inheritance ..................................................... 9
  implementation ............................................ 49
initialisation ............................................... 8;23;26;49
in-process ...................................................... 34
instance ......................................................... 8
instantiation ............................................... 23;26;49
Integrated Development Environment See
  IDE
<table>
<thead>
<tr>
<th>Term</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>interface</td>
<td>7:99</td>
</tr>
<tr>
<td>multiple</td>
<td>26:49</td>
</tr>
<tr>
<td>out-going</td>
<td>61</td>
</tr>
<tr>
<td>querying</td>
<td>87</td>
</tr>
<tr>
<td>Interface Definition Language</td>
<td>See IDL</td>
</tr>
<tr>
<td>Interface Repository</td>
<td>See IR</td>
</tr>
<tr>
<td>internal</td>
<td>5</td>
</tr>
<tr>
<td>introspection</td>
<td>13</td>
</tr>
<tr>
<td>Introspector</td>
<td>26</td>
</tr>
<tr>
<td>IR</td>
<td>24</td>
</tr>
<tr>
<td>IUnknown</td>
<td>31</td>
</tr>
<tr>
<td>library</td>
<td>99</td>
</tr>
<tr>
<td>Licensing Service</td>
<td>25</td>
</tr>
<tr>
<td>load balancing</td>
<td>18;25;56</td>
</tr>
<tr>
<td>location transparency</td>
<td>17;55</td>
</tr>
<tr>
<td>marshaling</td>
<td>42;50</td>
</tr>
<tr>
<td>automation</td>
<td>43</td>
</tr>
<tr>
<td>custom</td>
<td>48</td>
</tr>
<tr>
<td>standard</td>
<td>45</td>
</tr>
<tr>
<td>type library</td>
<td>43</td>
</tr>
<tr>
<td>method</td>
<td>7</td>
</tr>
<tr>
<td>Microsoft IDL compiler</td>
<td>See MIDL</td>
</tr>
<tr>
<td>MIDL</td>
<td>41</td>
</tr>
<tr>
<td>naming</td>
<td>18;49</td>
</tr>
<tr>
<td>object</td>
<td>6:49</td>
</tr>
<tr>
<td>Object Request Broker</td>
<td>See ORB</td>
</tr>
<tr>
<td>object server</td>
<td>30</td>
</tr>
<tr>
<td>ORB</td>
<td>23</td>
</tr>
<tr>
<td>out-of-process</td>
<td>34</td>
</tr>
<tr>
<td>performance</td>
<td>56</td>
</tr>
<tr>
<td>persistence</td>
<td>13:27</td>
</tr>
<tr>
<td>Persistent Object Service</td>
<td>25</td>
</tr>
<tr>
<td>platform neutral</td>
<td>26</td>
</tr>
<tr>
<td>polymorphism</td>
<td>10;24;26;39;50</td>
</tr>
<tr>
<td>portable</td>
<td>16</td>
</tr>
<tr>
<td>private</td>
<td>10</td>
</tr>
<tr>
<td>ProgID</td>
<td>32</td>
</tr>
<tr>
<td>Programmatic Identifier</td>
<td>See ProgID</td>
</tr>
<tr>
<td>property</td>
<td>7:57</td>
</tr>
<tr>
<td>protected</td>
<td>10</td>
</tr>
<tr>
<td>proxy/stub</td>
<td>87</td>
</tr>
<tr>
<td>public</td>
<td>10</td>
</tr>
<tr>
<td>reengineering</td>
<td>87</td>
</tr>
<tr>
<td>reference counting</td>
<td>87</td>
</tr>
<tr>
<td>Registry</td>
<td>41</td>
</tr>
<tr>
<td>Remote Procedure Call</td>
<td>See DCE RPC</td>
</tr>
<tr>
<td>replication</td>
<td>18</td>
</tr>
<tr>
<td>revision management</td>
<td>15</td>
</tr>
<tr>
<td>SCM</td>
<td>34</td>
</tr>
<tr>
<td>security</td>
<td>19;56</td>
</tr>
<tr>
<td>Security Service</td>
<td>25</td>
</tr>
<tr>
<td>self registering</td>
<td>87</td>
</tr>
<tr>
<td>Semantic Safearray Wrapper</td>
<td>109</td>
</tr>
<tr>
<td>Service Control Manager</td>
<td>See SCM</td>
</tr>
<tr>
<td>specialisation</td>
<td>9</td>
</tr>
<tr>
<td>taxonomy</td>
<td>70</td>
</tr>
<tr>
<td>Trader Service</td>
<td>24</td>
</tr>
<tr>
<td>transaction management</td>
<td>16;56</td>
</tr>
<tr>
<td>Transaction Service</td>
<td>24</td>
</tr>
<tr>
<td>type library</td>
<td>50;87</td>
</tr>
<tr>
<td>Universally Unique IDentifier</td>
<td>See UUID</td>
</tr>
<tr>
<td>UUID</td>
<td>See GUID</td>
</tr>
<tr>
<td>versioning</td>
<td>24:49</td>
</tr>
<tr>
<td>V-Table</td>
<td>33</td>
</tr>
</tbody>
</table>
8 References


[CPMCOM97] Kevin Sullivan, Mark Marchukov, “Compositionality Problems in Microsoft's COM”, Department of Computer Science, University of Virginia, 1997


[DP96] Erich Gamma et al. *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1996


[JAVA95] “ActiveX integration with Java”, Microsoft Corporation, 1995


[SMALLTALK80] Adele Goldberg and David Robson, *Smalltalk-80 The Language*, Addison-Wesley Publishing Company


9 Appendix A: TypeLibMarshal

The files in this appendix pertain to section 3.1.6.2 and can be found on the companion CD in directory \Appendix\TypeLibMarshal.

9.1 Server

9.1.1 Project file: prjTypeLibMarshal.dpr

program prjTypeLibMarshal;

uses
    Forms,
    unitForm in 'unitForm.pas' (Form1),
    prjTypeLibMarshal_TLB in 'prjTypeLibMarshal_TLB.pas',
    unitImpl in 'unitImpl.pas' (ITypeLibMarshal: CoClass);

{$R *.TLB}
{$R *.RES}

begin
    Application.Initialize;
    Application.CreateForm(TForm1, Form1);
    Application.Run;
end.

9.1.2 Main form: unitForm.pas

unit unitForm;

interface

uses
    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
    StdCtrls;

type
    TForm1 = class(TForm)
        Memo1: TMemo;
    private
        { Private declarations }
    public
        { Public declarations }
    end;

var
    Form1: TForm1;

implementation

{$R *.DFM}

end.
9.1.3 Implementation unit: UnitImpl.pas

unit unitImpl;

interface

uses
  ComObj, ActiveX, prjTypeLibMarshal_TLB, unitForm, SysUtils;

type
  TITypeLibMarshal = class(TAutoObject, IITypeLibMarshal)
  protected
    procedure GetPersonalData(Data: OleVariant); safecall;
  end;

implementation

uses ComServ;

procedure TITypeLibMarshal.GetPersonalData(Data: OleVariant);
var
  cCount, i: Integer;
begin
  if VarIsArray(data) then
  begin
    with Form1.Memo1.Lines do
    begin
      Add('* Person *****************');
      Add(Data[0] + ' ' + Data[1] + ' (' + IntToStr(Data[2]) + ')');
      if(VarIsArray(Data[3])) then
      begin
        cCount := VarArrayHighBound(Data[3], 1);
        i := 0;
        Add('- Children ----------');
        while (i <= cCount) do
        begin
          Add(IntToStr(i) + ': ' +
              Data[3][i][0] + ' ' +
              Data[3][i][1] + ' (' +
              IntToStr(Data[3][i][2]) + ')');
          inc(i);
        end
        Add('**************************');
      end;
      end;
    end;
  end;
end;

initialization
  TAutoObjectFactory.Create(ComServer, TITypeLibMarshal, Class_ITypeLibMarshal,
  ciMultiInstance);
end.

9.1.4 Type library wrapper file: prjTypeLibMarshal_TLB.pas

unit prjTypeLibMarshal_TLB;

{ This file contains pascal declarations imported from a type library.}
This file will be written during each import or refresh of the type library editor. Changes to this file will be discarded during the refresh process.

{ prjTypeLibMarshal Library }
{ Version 1.0 }

interface

uses Windows, ActiveX, Classes, Graphics, OleCtrls, StdVCL;

const
 LIBID_prjTypeLibMarshal: TGUID = '{5F9F4A20-5E81-11D1-B030-0020AF3BC782}';

const
 { Component class GUIDs }
 Class_ITypeLibMarshal: TGUID = '{5F9F4A22-5E81-11D1-B030-0020AF3BC782}';

type

 { Forward declarations: Interfaces }
 IITypeLibMarshal = interface;

 { Forward declarations: CoClasses }
 ITypeLibMarshal = ITypeLibMarshal;

 { Dispatch interface for ITypeLibMarshal Object }
 IITypeLibMarshal = interface(IUnknown)
  [ '{5F9F4A21-5E81-11D1-B030-0020AF3BC782}' ]
   procedure GetPersonalData(Data: OleVariant); safecall;
end;

{ ITypeLibMarshalObject }

CoITypeLibMarshal = class
  class function Create: IITypeLibMarshal;
  class function CreateRemote(const MachineName: string): IITypeLibMarshal;
end;

implementation

uses ComObj;

class function CoITypeLibMarshal.Create: ITypeLibMarshal;
begin
  Result := CreateComObject(Class_ITypeLibMarshal) as ITypeLibMarshal;
end;

class function CoITypeLibMarshal.CreateRemote(const MachineName: string): ITypeLibMarshal;
begin
  Result := CreateRemoteComObject(MachineName, Class_ITypeLibMarshal) as ITypeLibMarshal;
end;
9.2 CLIENT

9.2.1 Project file: prjClient.dpr

program prjClient;

uses
  Forms,
  unitForm in 'unitForm.pas' (Form1),
  unitKidDlg in 'unitKidDlg.pas' (frmKidDlg),
  prjTypeLibMarshal_TLB in '..\..\..\Programmer\Borland\Delphi 3\Imports\prjTypeLibMarshal_TLB.pas';

{$R *.RES}

begin
  Application.Initialize;
  Application.CreateForm(TfrmClient, frmClient);
  Application.CreateForm(TfrmKidDlg, frmKidDlg);
  Application.Run;
end.

9.2.2 Main form: unitForm.pas

unit unitForm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, ComObj, prjTypeLibMarshal_TLB;

type
  TfrmClient = class(TForm)
   GroupBox1: TGroupBox;
    editName: TEdit;
    editLastName: TEdit;
    editAge: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    GroupBox2: TGroupBox;
    listKids: TListBox;
    btnAddKid: TButton;
    GroupBox3: TGroupBox;
    btnSendData: TButton;
    editServer: TEdit;
    Label4: TLabel;
  private
    procedure btnSendDataClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure btnAddKidClick(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  public
    
private
    endl.
{ Public declarations }
end;

var
frmClient: TfrmClient;
Class_DispSafeArray: TGUID = '{5F9F4A22-5E81-11D1-B030-0020AF3BC782}';
Server: ITypeLibMarshal;     // Server Interface reference
Kids: TStringList;           // Internal representation of children data

implementation

uses unitKidDlg;

{$R *.DFM}

{----------------------------------------------------------------------
| Function: Create SafeArray of parent and child data and send it to
| the server
|----------------------------------------------------------------------
}
procedure TfrmClient.btnSendDataClick(Sender: TObject);
var
  theParent, theKids, Billy: Variant;
i: Integer;
begin
  Server := CoITypeLibMarshal.CreateRemote(editServer.Text);

  theKids := VarArrayCreate([0, (Kids.Count div 3)-1], varVariant);

  i := 0;
  while(i < Kids.Count) do
  begin
    Billy := VarArrayCreate([0, 2], varVariant);
    Billy[0] := Kids.Strings[i];
    Billy[1] := Kids.Strings[i+1];
    Billy[2] := Kids.Strings[i+2];

    theKids[i div 3] := Billy;
    i := i + 3;
  end;

  theParent := VarArrayCreate([0, 3], varVariant);
  theParent[0] := editName.Text;
  theParent[1] := editLastName.Text;
  theParent[2] := StrToInt(editAge.Text);

  Server.GetPersonalData(theParent);
end;

{----------------------------------------------------------------------
| Function: Get a reference to server interface (on adam)
|----------------------------------------------------------------------
}
procedure TfrmClient.FormCreate(Sender: TObject);
begin
  Kids := TStringList.Create;
end;

{---------------------------------------------------------------------


procedure TfrmClient.btnAddKidClick(Sender: TObject);
begin
  if (frmKidDlg.ShowModal = mrOK) then
  begin
    listKids.Items.Add(frmKidDlg.editName.Text + ' ' +
      frmKidDlg.editLastName.Text + ' (' +
      frmKidDlg.editAge.Text + ')');
    Kids.Add(frmKidDlg.editName.Text);
    Kids.Add(frmKidDlg.editLastName.Text);
    Kids.Add(frmKidDlg.editAge.Text);
  end;
end;

{----------------------------------------------------------------------
- Function: Deallocate resources
----------------------------------------------------------------------}
procedure TfrmClient.FormDestroy(Sender: TObject);
begin
  Kids.Destroy;
end.
end.

9.2.3 Dialog box form: unitKidDlg.pas

unit unitKidDlg;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, Buttons;

type
  TfrmKidDlg = class TForm
    GroupBox1: TGroupBox;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    editName: TEdit;
    editLastName: TEdit;
    editAge: TEdit;
    BitBtn1: TBitBtn;
    BitBtn2: TBitBtn;
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  frmKidDlg: TfrmKidDlg;

implementation
\{$R *.DFM\}$

end.
10 APPENDIX B: STANDARDMARSHAL

The files in this appendix pertain to section 3.1.6.3 and can be found on the companion CD in directory \Appendix\StandardMarshal.

10.1 SERVER

10.1.1 Project file: prjMarshalStruct.dpr

```pascal
program prjMarshalStruct;
uses
    Forms,
    unitForm in 'unitForm.pas' {Form1},
    unitImpl in 'unitImpl.pas',
    prjMarshalStruct_TLB in 'prjMarshalStruct_TLB.pas';

{$R *.RES}

{$R 'marshalstruct.tlb'}

begin
    Application.Initialize;
    Application.CreateForm(TForm1, Form1);
    Application.Run;
end.
```

10.1.2 Main form: unitForm.pas

```pascal
unit unitForm;

interface

uses
    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, StdCtrls;

type
    TForm1 = class(TForm)
        Memo1: TMemo;
        Label1: TLabel;
    private
        { Private declarations }
    public
        { Public declarations }
    end;

var
    Form1: TForm1;

implementation

{$R *.DFM}

end.
```
10.1.3 Server main code: unitImpl.pas

unit unitImpl;

interface

uses Windows, SysUtils, ComObj, prjMarshalStruct_TLB, unitForm;

type
   TMarshalStruct = class(TAutoObject, IMarshalStruct)
      protected
         function GetPerson(var data: _Person): HResult; stdcall;
      end;

implementation

uses ComServ;

{- Function: Implementation of IMarshalStruct.GetPerson. Add the data
  to the memobox. -}
function TMarshalStruct.GetPerson(var data: _Person): HResult;
begin
   Form1.Memo1.Lines.Add('GetPerson');
   Form1.Memo1.Lines.Add(WideCharToString(data.firstname) + ' ' +
                           WideCharToString(data.lastname) + ' aged ' +
                           IntToStr(data.age));
   result := S_OK;       // Always return OK.
end;

initialization
   TAutoObjectFactory.Create(ComServer, TMarshalStruct, Class_MarshalStruct,
                             ciMultiInstance);
end.

10.1.4 Type library wrapper file: prjMarshalStruct_TLB.pas

unit prjMarshalStruct_TLB;

{ This file contains pascal declarations imported from a type library.
  This file will be written during each import or refresh of the type
  library editor. Changes to this file will be discarded during the
  refresh process. }

{ prjMarshalStruct Library }
{ Version 1.0 }

interface

uses Windows, ActiveX, Classes, Graphics, OleCtrls, StdVCL;

const
   LIBID_prjMarshalStruct: TGUID = '{581171d4-5b61-11d1-b029-0020af3bc782}';

const
{ Component class GUIDs }
Class.MarshalStruct: TGUID = '{581171D6-5B61-11D1-B029-0020AF3BC782}';

type

{ Forward declarations: Interfaces }
IMarshalStruct = interface;

{ Forward declarations: CoClasses }
MarshalStruct = IMarshalStruct;

_GUID = record
  Data1: UINT;
  Data2: Word;
  Data3: Word;
  Data4: array[0..7] of Byte;
end;

_Person = record
  firstname: PWideChar;
  lastname: PWideChar;
  age: SYSINT;
end;

{ IUnknown Interface for MarshalStruct Object }

IMarshalStruct = interface(IUnknown)
  ['{581171D5-5B61-11D1-B029-0020AF3BC782}']
  function GetPerson(var data: _Person): HRESULT; stdcall;
end;

{ MarshalStructObject }

CoMarshalStruct = class
  class function Create: IMarshalStruct;
  class function CreateRemote(const MachineName: string): IMarshalStruct;
end;

implementation

uses ComObj;

class function CoMarshalStruct.Create: IMarshalStruct;
begin
  Result := CreateComObject(Class.MarshalStruct) as IMarshalStruct;
end;

class function CoMarshalStruct.CreateRemote(const MachineName: string): IMarshalStruct;
begin
  Result := CreateRemoteComObject(MachineName, Class.MarshalStruct) as IMarshalStruct;
end;

end.
10.1.5 Proxy/Stub def file: Proxy.def

LIBRARY Proxy.dll
DESCRIPTION ‘Proxy/Stub DLL’
EXPORTS  
  DllGetClassObject @1 PRIVATE
  DllCanUnloadNow @2 PRIVATE
  GetProxyDllInfo @3 PRIVATE
  DllRegisterServer @4 PRIVATE
  DllUnregisterServer @5 PRIVATE

10.1.6 IDL Code: MarshalStruct.IDL

typedef struct _Person {  
  [string] wchar_t *firstname;
  [string] wchar_t *lastname;
  int age;
} Person;

[odl,
  uuid(581171D5-5B61-11D1-B029-0020AF3BC782),
  version(1.0),
  helpstring("IUnknown Interface for MarshalStruct Object"),
  hidden,
  object ]
interface IMarshalStruct : IUnknown {
  import "unkwn.idl";
  HRESULT _stdcall GetPerson([in] Person *data);
};

/* Type library definition */
[
  uuid(581171D4-5B61-11D1-B029-0020AF3BC782),
  version(1.0),
  helpstring("prjMarshalStruct Library")
]library prjMarshalStruct
{
  [uuid(581171D6-5B61-11D1-B029-0020AF3BC782),
   version(1.0),
   helpstring("MarshalStructObject")
  ]coclass MarshalStruct {
    [default] interface IMarshalStruct;
  };

  interface IMarshalStruct;
}
10.1.7 Proxy/Stub makefile: makefile

**clean:**

```
cmd del *.c
del *.obj
del *.h
doidl : marshalstruct.idl
    midl marshalstruct.idl
dllldata.obj : dllldata.c
    cl /c /DWIN32 /DREGISTER_PROXY_DLL dllldata.c
marshalstruct_p.obj : marshalstruct_p.c
    cl /c /DWIN32 /DREGISTER_PROXY_DLL marshalstruct_p.c
marshalstruct_i.obj : marshalstruct_i.c
    cl /c /DWIN32 /DREGISTER_PROXY_DLL marshalstruct_i.c
```

**PROXYSTUBOBJS** = dllldata.obj \ 
    marshalstruct_p.obj \ 
    marshalstruct_i.obj

**PROXYSTUBLIBS** = kernel32.lib \ 
    rpcndr.lib \ 
    rpcns4.lib \ 
    rpcrt4.lib \ 
    uuid.lib

```
proxy.dll : $(PROXYSTUBOBJS) proxy.def
    link /dll /out:proxy.dll /def:proxy.def \
        $(PROXYSTUBOBJS) $(PROXYSTUBLIBS)
            regsvr32 /s proxy.dll
```

10.2 CLIENT

10.2.1 Project main file: Project1.dpr

**program** Project1;

```
uses
    Forms,
    prjClient in ‘prjClient.pas’ (Form1),
    prjMarshalStruct_TLB in ‘..\..\..\Programmer\Borland\Delphi
3\Imports\prjMarshalStruct_TLB.pas’;

    {$R *.RES}
```

```
begin
    Application.Initialize;
    Application.CreateForm(TForm1, Form1);
    Application.Run;
end.
```

10.2.2 Client main unit: unitClient.pas

```
unit unitClient;
```
interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  prjMarshalStruct_TLB, StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.Button1Click(Sender: TObject);
var
  iMarsh: IMarshalStruct;
begin
  iMarsh := CoMarshalStruct.Create;
end;

end.
11 APPENDIX C: AUTOMATION

The files in this appendix pertain to section 3.3.1 and can be found on the companion CD in directory \Appendix\Automation.

11.1 SERVER

11.1.1 Project file: prjAutoTest.dpr

library prjAutoTest;

uses
  ComServ,
  AutoGetSet_TLB in 'AutoGetSet_TLB.pas',
  unitAutoTest in 'unitAutoTest.pas' {AutoTest: CoClass};

exports
  DllGetClassObject,
  DllCanUnloadNow,
  DllRegisterServer,
  DllUnregisterServer;

{$R *.TLB}
{$R *.RES}

begin
end.

11.1.2 Main unit: unitAutoTest.pas

{ ----------------------------------------------------------------------
  -- Unit:       unitAutoTest
  -- Function:   A (very) simple automation server
  ---------------------------------------------------------------------- }

unit unitAutoTest;

interface

uses
  ComObj, AutoGetSet_TLB;

type
  TAutoTest = class(TAutoObject, IAutoTest)
  public
    sval: WideString;
  protected
    function Peek: WideString; safecall;
    procedure IAutoTest.Poke = IAutoTest_Poke;
    procedure IAutoTest_Poke(const s: WideString); safecall;
  end;

implementation

uses ComServ;
function TAutoTest.Peek: WideString;
begin
  if(Sval = '') then Sval := 'Hello World';
  Result := Sval;
end;

procedure TAutoTest.IAutoTest_Poke(const s: WideString);
begin
  Sval := s;
end;

initialization
  TAutoObjectFactory.Create(ComServer, TAutoTest, Class_AutoTest,
  ciMultiInstance);
end.

11.1.3 Type library wrapper file: prjAutoTest_TLB.pas

unit prjAutoTest_TLB;

{ This file contains pascal declarations imported from a type library. This file will be written during each import or refresh of the type library editor. Changes to this file will be discarded during the refresh process. }

{ prjAutoTest Library }
{ Version 1.0 }

interface

uses Windows, ActiveX, Classes, Graphics, OleCtrls, StdVCL;

const
  LIBID_prjAutoTest: TGUID = '{B98BB810-3C8A-11D1-AFE4-2A083C000000}';

const
  { Component class GUIDs }
  Class_AutoTest: TGUID = '{B98BB812-3C8A-11D1-AFE4-2A083C000000}';

type

{ Forward declarations: Interfaces }
IAutoTest = interface;
IAutoTestDisp = dispinterface;

{ Forward declarations: CoClasses }
AutoTest = IAutoTest;
{ Dispatch interface for AutoTest Object }

IAutoTest = interface(IDispatch)
    
    function Peek: WideString; safecall;
    procedure Poke(const s: WideString); safecall;
end;

{ DispInterface declaration for Dual Interface IAutoTest }

IAutoTestDisp = dispinterface

    function Peek: WideString; dispid 1;
    procedure Poke(const s: WideString); dispid 2;
end;

{ AutoTestObject }

CoAutoTest = class
    class function Create: IAutoTest;
    class function CreateRemote(const MachineName: string): IAutoTest;
end;

implementation

uses ComObj;

class function CoAutoTest.Create: IAutoTest;
begin
    Result := CreateComObject(Class_AutoTest) as IAutoTest;
end;

class function CoAutoTest.CreateRemote(const MachineName: string): IAutoTest;
begin
    Result := CreateRemoteComObject(MachineName, Class_AutoTest) as IAutoTest;
end;

end.

11.2 CLIENT

11.2.1 Project file: prjClient.dpr

program prjClient;

uses
    Forms,
    unitClient in ‘unitClient.pas’ (Form1),
    prjAutoTest_TLB in ‘..\..\..\Programmer\Borland\Delphi 3\Imports\prjAutoTest_TLB.pas’;

{$R *.RES}
begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.

11.2.2 Main unit: unitClient.pas

{ *--------------------------------------------------------------------------
  -- Unit:        unitClient
  -- Function:    An automation client
  *-------------------------------------------------------------------------- }

unit unitClient;

interface

uses Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
StdCtrls, ComObj, ActiveX, AutoGetSet_TLB;

type
  TForm1 = class(TForm)
    btnGetNames: TButton;
    Memo1: TMemo;
    btnPoke: TButton;
    btnPeek: TButton;
    editText: TEdit;
    btnLateBind: TButton;
    procedure btnGetNamesClick(Sender: TObject);
    procedure btnPeekClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure btnPokeClick(Sender: TObject);
    procedure btnLateBindClick(Sender: TObject);
  private
    function DispCheck(res: HRESULT; err: String): boolean;
  end;

var
  Form1: TForm1;
  dispInt: IDispatch;       // Global handle to automation server
  m_lcid: Integer;          // Global Locale handle

implementation

{$R *.DFM}

{ *--------------------------------------------------------------------------
  -- Function:    Get names of all member functions in automation server
  -- Parameters:  Sender: Sending VCL component
  *-------------------------------------------------------------------------- }

procedure TForm1.btnGetNamesClick(Sender: TObject);
var
  Attr: PTypeAttr;
  funcdesc: PFuncDesc;
  i: Integer;
  memid: TMEMBERID;
  name: TBStr;
v: String;
IInfo: ITypeInfo;
begin
  // Get type info of dispatch interface
  if(DispCheck(dispInt.GetTypeInfo(0, m_lcid, IInfo), 'GetTypeInfo')) then begin
    // Get attributes of disp. intf. from type info
    IInfo.GetTypeAttr(Attr);
    i := 0;
    // Loop through #functions in disp intf.
    while(i < Attr.cFuncs) do begin
      // Get function descriptor for next function
      IInfo.GetFuncDesc(i, funcdesc);

      // Get corresponding member ID
      memid := funcdesc.memid;

      // Get human readable name of this member
      IInfo.GetDocumentation(memid, @name, nil, nil, nil);

      // Print data in a nice manner
      V := Name;
      if(Length(Name) < 8) then V := V + #9;
      V := V + #9 + IntToStr(funcdesc.memid) + #9;
      if(Length(IntToStr(funcdesc.memid)) < 8) then V := V + #9;
      V := V + '#Params: ' + IntToStr(funcdesc.cParams);

      // Add nice lines to memo box
      Memo1.Lines.Add(v);

      // Deallocate func desc.
      IInfo.ReleaseFuncDesc(funcdesc);
      funcdesc := nil;
      inc(i);
    end;
  end;
end;

{ -----------------------------------------------------------------------------
  -- Function:   Raise an exception if an OLE error occurred
  -- Parameters: res: Ole return result value
  -- err: optional custom error message
  -- Note:       This function has the side effect of raising an exception
  -----------------------------------------------------------------------------}
function TForm1.DispCheck(res: HRESULT; err: String): boolean;
var
  Sval: String;
begin
  case res of
    E_OUTOFMEMORY: SVal := 'Out of memory.';
    
E_INVALIDARG: SVal := 'One or more of the arguments is invalid.';
TYPE_E_IOPERROR: Sval := 'The function could not read from the file.';
TYPE_E_INVDATAREAD: Sval := 'The function could not read from the file.';
TYPE_E_UNSUPFORMAT: Sval := 'The type library has an old format.';
TYPE_E_INVALIDSTATE: Sval := 'The type library could not be opened.';
TYPE_E_WRONGTYPEKIND: Sval := 'Type mismatch.';
TYPE_E_ELEMENTNOTFOUND: Sval := 'The element was not found.';
DISP_E_BADPARAMCOUNT: Sval := 'The number of elements provided DISPPARAMS
is different from the number of arguments accepted by the method or property.';
DISP_E_BADVARTYPE: Sval := 'One of the arguments in rgvarg is not a valid
variant type.';
DISP_E_EXCEPTION: Sval := 'The application needs to raise an exception. In
this case the structure passed in pexcepinfo should be filled in.';
DISP_E_MEMBERNOTFOUND: Sval := 'The requested member does not exist, or the
call to Invoke tried to set the value of a read-only property.';
DISP_E_NOMOREDARGS: Sval := 'This implementation of IDispatch does not
support named arguments.';
DISP_E_OVERFLOW: Sval := 'One of the arguments in rgvarg could not be
coerced to the specified type.';
DISP_E_PARAMNOTFOUND: Sval := 'One of the parameter dispatch IDs does not
correspond to a parameter on the method. In this case puArgErr should be set to
the first argument that contains the error.';
DISP_E_TYPEMISMATCH: Sval := 'One or more of the arguments could not be
coerced. The index within rgvarg of the first parameter with the incorrect type
is returned in the puArgErr parameter.';
DISP_E_UNKNOWNINTERFACE: Sval := 'The interface ID passed in riid is not
IID_NULL.';
DISP_E_UNKNOWNLCID: Sval := 'The member being invoked interprets string
arguments according to the locale ID (LCID), and the LCID is not recognized. If
the LCID is not needed to interpret arguments, this error should not be
returned.';
DISP_E_UNKNOWNNAME: Sval := 'One or more of the names were not known.' +
'The returned array of DISPID contains
DISPID_UNKNOWN for each entry that corresponds to an unknown name.';
end;

if(res <> S_OK) then
begin
  Application.MessageBox(PChar(sval), Pchar(err), mb_OK);
  Result := False;
end
else
  Result := True;
end;

{ ----------------------------------------------------------------------}
{ -- Function: Call the 'Peek' function, using late binding
{ -- Parameters: Sender: Delphi VCL caller
{ ---------------------------------------------------------------------- }

procedure TForm1.btnPeekClick(Sender: TObject);
var
  Params: TDispParams;
  Res: Variant;
  Names: array[0..0] of TBStr;
  dID: array[0..0] of Integer;
begin
  // Set entry to known name
  Names[0] := 'Peek';
// Get dispID corresponding to ‘Peek’
DispCheck(dispInt.GetIDsOfNames(GUID_NULL, @Names, 1, m_lcid, @dID),
    ‘IDispatch.GetIDsOfNames’);

// If an error occurred, forget it.
if(dID[0] > -1) then
begin
    // Create an empty dispparams structure
    Params.rgvarg := nil;
    Params.rgdispidNamedArgs := nil;
    Params.cArgs := 0;
    Params.cNamedArgs := 0;

    // Call ‘Peek’ using the obtained dispID
    dispInt.Invoke(dID[0], GUID_NULL, m_lcid,
        DISPATCH_METHOD, Params, @Res, nil, nil);

    // Add result to memo box
    Memo1.Lines.Add(Res);
end;
end;

{ ----------------------------------------------------------------------
-- Function:   Initialize global variables
-- Parameters: Sender: Delphi VCL caller
-----------------------------------------------------------------------
procedure TForm1.FormCreate(Sender: TObject);
var
    punk: IUnknown;
begin
    // Get locale ID
    m_lcid := GetUserDefaultLCID;

    // Create automation server using somewhat early binding
    dispInt := CoAutoTest.Create;
end;

{ ----------------------------------------------------------------------
-- Function:   Call the ‘Poke’ function using earlier binding
-- Parameters: Sender: Delphi VCL caller
-----------------------------------------------------------------------
procedure TForm1.btnPokeClick(Sender: TObject);
var
    Params: TDispParams;
    ArgList: array[0..1] of TVariantArg;
begin
    // Fill entry #0 of arglist with data
    ArgList[0].vt := VT_BSTR;
    ArgList[0].bStrVal := StringToOleStr(editText.Text);

    // Fill dispparams structure with arglist
    Params.rgvarg := @ArgList;
    Params.rgdispidNamedArgs := nil;
    Params.cArgs := 1;
    Params.cNamedArgs := 0;

    // Call poke method on disp. intf.
DispCheck(dispInt.Invoke(2, GUID_NULL, m_lcid,
               DISPATCH_METHOD, Params, nil, nil, nil), 'Poke');

end;

{ ----------------------------------------------------------------------
   -- Function:   Test late binding, the nice way
   -- Parameters: Sender: Delphi VCL caller
   ---------------------------------------------------------------------- }

procedure TForm1.btnLateBindClick(Sender: TObject);
var
  v: Variant;
begin
  // Create object using late binding
  v := CreateOleObject('AutoGetSet.AutoTest');

  // Set value of automation server
  v.Poke('Late binding works!');

  // Read value of automation server
  Memo1.Lines.Add(v.Peek);
end;

end.
12 Appendix D: Explorer Spy

The files in this appendix pertain to section 3.3.2 and can be found on the companion CD in directory \Appendix\ExplorerSpy

12.1.1 Project file: MassaJrLib.dpr

program MassaJrLib;

uses
    Forms,
    ComServ,
    unitForm in 'unitForm.pas' {frmMain},
    MassaJrLib_TLB in 'MassaJrLib_TLB.pas',
    WebBrowserEventsImpl in 'WebBrowserEventsImpl.pas' {WebBrowserEvents: CoClass},
    SHDocVw_TLB in 'E:\Programmer\Borland\Delphi 3\IMPORTS\SHDocVw_TLB.pas';

{$R *.TLB}
{$R *.RES}

begin
    Application.Initialize;
    Application.CreateForm(TfrmMain, frmMain);
    Application.Run;
end.

12.1.2 Main unit: unitForm.pas

unit unitForm;

interface

uses
    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
    StdCtrls, ComCtrls, SHDocVw_TLB, WebBrowserEventsImpl, ActiveX, ComObj;

type
    TfrmMain = class(TForm)
        btnConnect: TButton;
        btnDisconnect: TButton;
        StatusBar1: TStatusBar;
        memoOutput: TMemo;
        procedure btnConnectClick(Sender: TObject);
        procedure FormCreate(Sender: TObject);
        procedure btnDisconnectClick(Sender: TObject);
        procedure FormCloseQuery(Sender: TObject; var CanClose: Boolean);
    private
        { Private declarations }
        procedure SetFormCaption;
        procedure Disconnect;
    public
        { Public declarations }
        IE: IDispatch;     // Handle to Internet Explorer main interface
        IWeb: IWebBrowserApp; // Handle to WebBrowserApp interface in IE
Sink: TWebBrowserEvents; // Event sink
CP: IConnectionPoint;   // Connection point in IE for Event sink
Cookie: Integer;        // Cookie to identify sink connection
end;

var
  frmMain: TfrmMain;

const
  APPNAME = 'Explorer Spy';
  DEFAULTURL = 'http://www.daimi.aau.dk/';

implementation

{$R *.DFM}

procedure TfrmMain.btnConnectClick(Sender: TObject);
var
  Enum: IEnumConnectionPoints;
  CPC: IConnectionPointContainer;
  Fetched: LongInt;
  Dummy, Flags: OleVariant;
begin
  if(IE = nil) then
  begin
    StatusBar1.SimpleText := 'Loading Internet Explorer';
    Application.ProcessMessages;
    IE := CreateOleObject('InternetExplorer.Application');
    StatusBar1.SimpleText := ''; 
    if(IE <> nil) then
    begin
      IE.QueryInterface(IWebBrowserApp, IWeb);
      IWeb.StatusText := 'Obeying '+APPNAME;
      IWeb.StatusBar := True;

      // Get connection point container
      OleCheck(IWeb.QueryInterface(IConnectionPointContainer, CPC));

      // Get connection point enumerator
      OleCheck(CPC.EnumConnectionPoints(Enum));

      // Get connectionpoint (there is just 1)
      Enum.Next(1, CP, @Fetched);

      // Create and install the event sink
      Sink := TWebBrowserEvents.Create;
      OleCheck(CP.Advise(Sink, Cookie));
      IWeb.Visible := True;

      // Set caption
      SetFormCaption;

      // Disable button
      btnConnect.Enabled := False;
      btnDisconnect.Enabled := True;
    end;
  end;
end;
// Navigate to default homepage
flags := navNoReadFromCache;
IWeb.Navigate(DEFAULTURL, flags, dummy, dummy, dummy);
end
else
  Application.MessageBox('Internet Explorer wouldn’t start',
    'Error', mb_OK);
end;

procedure TfrmMain.FormCreate(Sender: TObject);
begin
  IE := nil;
  IWeb := nil;
  btnDisconnect.Enabled := False;
  btnConnect.Enabled := True;
end;

procedure TfrmMain.SetFormCaption;
begin
  Self.Caption := AppName + ' snooping ...';
end;

procedure TfrmMain.btnDisconnectClick(Sender: TObject);
begin
  Disconnect;
  btnDisconnect.Enabled := False;
  btnConnect.Enabled := True;
end;

procedure TfrmMain.Disconnect;
begin
  // Cleanup Internet Explorer
  if(IWeb <> nil) then
  begin
    // Uninstall event sink
    OleCheck(CP.Unadvise(Cookie));

    // Release interface pointers
    IWeb := nil;
    IE := nil;
  end;
end;

procedure TfrmMain.FormCloseQuery(Sender: TObject; var CanClose: Boolean);
begin
  Disconnect;
  CanClose := True;
end;
end.

12.1.3 Event sink unit: WebBrowserEventsImpl.pas

unit WebBrowserEventsImpl;
{ Implementation unit for CoClasses.

This pascal unit was autogenerated by Idun on 26-04-98
using the type library for MassaJrLib version 1.0
(Massa Junior Library) }

interface

uses

ComObj, MassaJrLib_TLB;

type

{ CoClass declaration for WebBrowserEvents }
TWebBrowserEvents = class(TAutoObject, DWebBrowserEvents)
protected
    procedure BeforeNavigate(const URL: WideString; Flags: Integer; const
TargetFrameName: WideString; var PostData: OleVariant; const Headers:
WideString; var Cancel: WordBool); safecall;
    procedure NavigateComplete(const URL: WideString); safecall;
    procedure StatusTextChange(const Text: WideString); safecall;
    procedure ProgressChange(Progress, ProgressMax: Integer); safecall;
    procedure DownloadComplete; safecall;
    procedure CommandStateChange(Command: Integer; Enable: WordBool); safecall;
    procedure DownloadBegin; safecall;
    procedure NewWindow(const URL: WideString; Flags: Integer; const
TargetFrameName: WideString; var PostData: OleVariant; const Headers:
WideString; var Processed: WordBool); safecall;
    procedure TitleChange(const Text: WideString); safecall;
    procedure FrameBeforeNavigate(const URL: WideString; Flags: Integer; const
TargetFrameName: WideString; var PostData: OleVariant; const Headers:
WideString; var Cancel: WordBool); safecall;
    procedure FrameNavigateComplete(const URL: WideString); safecall;
    procedure FrameNewWindow(const URL: WideString; Flags: Integer; const
TargetFrameName: WideString; var PostData: OleVariant; const Headers:
WideString; var Processed: WordBool); safecall;
    procedure Quit(var Cancel: WordBool); safecall;
    procedure WindowMove; safecall;
    procedure WindowResize; safecall;
    procedure WindowActivate; safecall;
    procedure PropertyChange(const szProperty: WideString); safecall;
end;

implementation

uses ComServ, unitForm, SysUtils;

{ Method implementations for CoClass WebBrowserEvents. Modify at will
-------------------------------------------
procedure TWebBrowserEvents.BeforeNavigate(const URL: WideString; Flags:
Integer; const TargetFrameName: WideString; var PostData: OleVariant; const
Headers: WideString; var Cancel: WordBool); safecall;
var
    BoolStr: String;
begin
    if (Cancel) then BoolStr := 'True' else BoolStr := 'False';
}
frmMain.memoOutput.Lines.Add('BeforeNavigate(URL='+URL+',Flags='+IntToStr(Flags)+',TargetFrameName='+TargetFrameName+'+Headers='+Headers+',Cancel='+BoolStr+')');
end;

procedure TWebBrowserEvents.NavigateComplete(const URL: WideString); safecall;
begin
  frmMain.memoOutput.Lines.Add('NavigateComplete(URL='+URL+')');
end;

procedure TWebBrowserEvents.StatusTextChange(const Text: WideString); safecall;
begin
  frmMain.memoOutput.Lines.Add('StatusTextChange(Text='+Text+')');
end;

procedure TWebBrowserEvents.ProgressChange(Progress: Integer; ProgressMax: Integer); safecall;
begin
  frmMain.memoOutput.Lines.Add('ProgressChange(Progress='+IntToStr(Progress)+',ProgressMax='+IntToStr(ProgressMax)+')');
end;

procedure TWebBrowserEvents.DownloadComplete; safecall;
begin
  frmMain.memoOutput.Lines.Add('DownloadComplete');
end;

procedure TWebBrowserEvents.CommandStateChange(Command: Integer; Enable: WordBool); safecall;
var
  BoolStr: String;
begin
  if (Enable) then BoolStr := 'True' else BoolStr := 'False';
  frmMain.memoOutput.Lines.Add('CommandStateChange='+IntToStr(Command)+',Enable:'+BoolStr+');
end;

procedure TWebBrowserEvents.DownloadBegin; safecall;
begin
  frmMain.memoOutput.Lines.Add('DownloadBegin');
end;

procedure TWebBrowserEvents.NewWindow(const URL: WideString; Flags: Integer;
const TargetFrameName: WideString; var PostData: OleVariant; const Headers:
WideString; var Processed: WordBool); safecall;
var
  BoolStr: String;
begin
  if (Processed) then BoolStr := 'True' else BoolStr := 'False';
  frmMain.memoOutput.Lines.Add('NewWindow(URL='+URL+',Flags='+IntToStr(Flags)+',TargetFrameName='+TargetFrameName+'+Headers='+Headers+',Processed='+BoolStr+');
end;

procedure TWebBrowserEvents.TitleChange(const Text: WideString); safecall;
begin
  frmMain.memoOutput.Lines.Add('TitleChange(Text='+Text+')');
end;

procedure TWebBrowserEvents.FrameBeforeNavigate(const URL: WideString; Flags: Integer; const TargetFrameName: WideString; var PostData: OleVariant; const Headers: WideString; var Cancel: WordBool); safecall;
var
  BoolStr: String;
begin
  if (Cancel) then BoolStr := 'True' else BoolStr := 'False';
  frmMain.memoOutput.Lines.Add('FrameBeforeNavigate(URL='+URL+',Flags='+IntToStr(Flags)+',TargetFrameName='+TargetFrameName+',Headers='+Headers+',Cancel='+BoolStr+')');
end;

procedure TWebBrowserEvents.FrameNavigateComplete(const URL: WideString); safecall;
begin
  frmMain.memoOutput.Lines.Add('FrameNavigateComplete(URL='+URL+')');
end;

procedure TWebBrowserEvents.FrameNewWindow(const URL: WideString; Flags: Integer; const TargetFrameName: WideString; var PostData: OleVariant; const Headers: WideString; var Processed: WordBool); safecall;
var
  BoolStr: String;
begin
  if (Processed) then BoolStr := 'True' else BoolStr := 'False';
  frmMain.memoOutput.Lines.Add('FrameNewWindow(URL='+URL+',Flags='+IntToStr(Flags)+',TargetFrameName='+TargetFrameName+',Headers='+Headers+',Processed='+BoolStr+')');
end;

procedure TWebBrowserEvents.Quit(var Cancel: WordBool); safecall;
var
  BoolStr: String;
begin
  if (Cancel) then BoolStr := 'True' else BoolStr := 'False';
  frmMain.memoOutput.Lines.Add('Quit(Cancel='+BoolStr+')');
end;

procedure TWebBrowserEvents.WindowMove; safecall;
begin
  frmMain.memoOutput.Lines.Add('WindowMove');
end;

procedure TWebBrowserEvents.WindowResize; safecall;
begin
  frmMain.memoOutput.Lines.Add('WindowResize');
end;

procedure TWebBrowserEvents.WindowActivate; safecall;
begin
  frmMain.memoOutput.Lines.Add('WindowActivate');
end;
procedure TWebBrowserEvents.PropertyChange(const szProperty: WideString);
safecall;
begin
   frmMain.memoOutput.Lines.Add('PropertyChange('+szProperty+')');
end;

initialization
   TAutoObjectFactory.Create(Comserver, TWebBrowserEvents,
Class_WebBrowserEvents, ciMultiInstance);
end.

12.1.4 Type library wrapper file: MassaJrLib_TLB.pas

unit MassaJrLib_TLB;
{
This file contains pascal declarations imported from a type library.
This file will be written during each import or refresh of the type
library editor. Changes to this file will be discarded during the
refresh process. }
{
Massa Junior Library }
{
Version 1.0 }

interface

uses Windows, ActiveX, Classes, Graphics, OleCtrls, StdVCL;

const
   LIBID_MassaJrLib: TGUID = '{C96C9B35-B0FE-11D1-B0E1-0020AF3BC782}';

const
   { Component class GUIDs }
   Class_WebBrowserEvents: TGUID = '{C96C9B37-B0FE-11D1-B0E1-0020AF3BC782}';

type

   { Forward declarations: Interfaces }
   DWebBrowserEvents = interface;
   DWebBrowserEventsDisp = dispinterface;

   { Forward declarations: CoClasses }
   WebBrowserEvents = DWebBrowserEvents;

   { Event interface for WebBrowserEvents control }

   DWebBrowserEvents = interface(IDispatch)
   [
   '{EAB22AC2-30C1-11CF-A7EB-0000C05B8E0B}']
   procedure BeforeNavigate(const URL: WideString; Flags: Integer; const
TargetFrameName: WideString; var PostData: OleVariant; const Headers:
WideString; var Cancel: WordBool); safecall;
procedure NavigateComplete(const URL: WideString); safecall;
procedure StatusTextChange(const Text: WideString); safecall;
procedure ProgressChange(Progress, ProgressMax: Integer); safecall;
procedure DownloadComplete; safecall;
procedure CommandStateChange(Command: Integer; Enable: WordBool); safecall;

}
procedure DownloadBegin; safecall;
procedure NewWindow(const URL: WideString; Flags: Integer; const TargetFrameName: WideString; var PostData: OleVariant; const Headers: WideString; var Processed: WordBool); safecall;
procedure TitleChange(const Text: WideString); safecall;
procedure FrameBeforeNavigate(const URL: WideString; Flags: Integer; const TargetFrameName: WideString; var PostData: OleVariant; const Headers: WideString; var Cancel: WordBool); safecall;
procedure FrameNavigateComplete(const URL: WideString); safecall;
procedure FrameNewWindow(const URL: WideString; Flags: Integer; const TargetFrameName: WideString; var PostData: OleVariant; const Headers: WideString; var Processed: WordBool); safecall;
procedure Quit(var Cancel: WordBool); safecall;
procedure WindowMove; safecall;
procedure WindowResize; safecall;
procedure WindowActivate; safecall;
procedure PropertyChange(const szProperty: WideString); safecall;
end;

{ DispInterface declaration for Dual Interface DWebBrowserEvents }

DWebBrowserEventsDisp = dispinterface
['{EAB22AC2-30C1-11CF-A7EB-0000C05BAE0B}']
procedure BeforeNavigate(const URL: WideString; Flags: Integer; const TargetFrameName: WideString; var PostData: OleVariant; const Headers: WideString; var Cancel: WordBool); dispid 100;
procedure NavigateComplete(const URL: WideString); dispid 101;
procedure StatusTextChange(const Text: WideString); dispid 102;
procedure ProgressChange(Progress, ProgressMax: Integer); dispid 108;
procedure DownloadComplete; dispid 104;
procedure CommandStateChange(Command: Integer; Enable: WordBool); dispid 105;
procedure DownloadBegin; dispid 106;
procedure NewWindow(const URL: WideString; Flags: Integer; const TargetFrameName: WideString; var PostData: OleVariant; const Headers: WideString; var Processed: WordBool); dispid 107;
procedure TitleChange(const Text: WideString); dispid 113;
procedure FrameBeforeNavigate(const URL: WideString; Flags: Integer; const TargetFrameName: WideString; var PostData: OleVariant; const Headers: WideString; var Cancel: WordBool); dispid 200;
procedure FrameNavigateComplete(const URL: WideString); dispid 201;
procedure FrameNewWindow(const URL: WideString; Flags: Integer; const TargetFrameName: WideString; var PostData: OleVariant; const Headers: WideString; var Processed: WordBool); dispid 204;
procedure Quit(var Cancel: WordBool); dispid 103;
procedure WindowMove; dispid 109;
procedure WindowResize; dispid 110;
procedure WindowActivate; dispid 111;
procedure PropertyChange(const szProperty: WideString); dispid 112;
end;

{ WebBrowserEventsObject }

CoWebBrowserEvents = class
  class function Create: DWebBrowserEvents;
  class function CreateRemote(const MachineName: string): DWebBrowserEvents;
end;
implementation

uses ComObj;

class function CoWebBrowserEvents.Create: DWebBrowserEvents;
begin
  Result := CreateComObject(Class_WebBrowserEvents) as DWebBrowserEvents;
end;

class function CoWebBrowserEvents.CreateRemote(const MachineName: string): DWebBrowserEvents;
begin
  Result := CreateRemoteComObject(MachineName, Class_WebBrowserEvents) as DWebBrowserEvents;
end;

double.

13 Appendix E: Taxonomy

The files in this appendix pertain to section 4.1.2 and can be found on the companion CD in directory \Appendix\Taxonomy.

The following pages list the questions that make up the taxonomy document as of 27/4-1998. The latest revision is available from http://www.data.dti.dk/cot/case3/taxonomy/taxomain.asp.

Following the taxonomy document are the answers to the investigation of Delphi.

13.1 Taxonomy Document

1. Component Server

1.1. Types of components?
   1.1.1. Support for visual Controls
   1.1.2. Support for non-visual Controls

1.2. Types of servers
   1.2.1. Can the tool generate/create In-process servers?
   1.2.2. Can the tool generate/create Out-of-process servers?
   1.2.3. How can you make transition between in-process and out-of-process servers?
     1.2.3.1. You can do it through your IDE by clicking in some menu (where)?
     1.2.3.2. Changing the code by hand?
     1.2.3.3. Restarting the project and moving the code manually?
     1.2.3.4. Other:
   1.2.4. Can Multiple Components be contained in the same server?
   1.2.5. Are there any limit on the number of different components?

1.3. Component construction (tool support)

1.3.1. How is the COM components created?
   1.3.1.1. Not using wizard. Comments:
   1.3.1.2. using wizard. Comments:
     1.3.1.2.1. Which code-generation-wizards exist (list names and usages)?
     1.3.1.2.2. Description of the available options or generating the code (parameters, code-templates etc.)
     1.3.1.2.3. Description of the architecture of the generated code?
     1.3.1.2.4. Is it easy to customize the generated code?
     1.3.1.2.5. Does the wizards allow for reverse-engineering? To answer this question you will answer the following
     1.3.1.2.5.1. Are customizations in the generated code preserved when reverse-engineering the code?
1.3.1.2.5.2. Is it possible to change the fundamental strategies for the generated code?

1.3.1.2.5.3. Evaluation of the maintenance of the generated code (customizations, additions, changes, etc.). What is it allowed to change in the generated code while still retaining the possibility of reverse-engineering it?

1.3.1.2.6. Evaluation of the quality of the generated code

1.3.1.2.6.1. Performance, stability, "nice" and consistent coding standard?

1.3.1.2.6.2. Comments:

1.3.2. Handcoding

1.3.2.1. Is it easy?

1.4. Support for interfaces

1.4.1. Which types of interfaces does the tool support:

1.4.1.1. Custom?

1.4.1.2. Dispatch?

1.4.1.3. Dual?

1.4.2. How do you define an interface?

1.4.3. Is it possible to implement multiple interfaces on the component?

1.4.3.1. If Yes, How do you do that?

1.4.4. Predefined interfaces:

1.4.4.1. Is possible to implement predefined interfaces, meaning implementing interfaces defined by other people?

1.4.4.2. What do you use as foundation for the predefined interface?

1.4.4.2.1. Type lib?

1.4.4.2.2. IDL?

1.4.4.2.3. Source code?

1.4.4.2.4. other

1.4.4.3. How do you use a predefined interface?

1.4.5. Interface maintenance

1.4.5.1. Is changes in a published interface allowed

1.4.5.2. Insert new methods and/or parameters?

1.4.5.3. Change existing methods and/or parameters?

1.4.5.4. Delete methods and/or parameters?

1.4.5.5. What are the consequences on the generated component?
1.4.6. Interface optimizations: Which type of interface optimization are supported?
   1.4.6.1. None ?
   1.4.6.2. Tear-of interfaces, How: ?
   1.4.6.3. Other:

1.5. Reference counting
   1.5.1. A description of aspects of reference counting
      1.5.1.1. In/out-parameters ?
      1.5.1.2. Aggregation ?
   1.5.2. How is it implemented in the tool ?
      1.5.2.1. Is AddRef and Release called automatically by the tool ?
      1.5.2.2. Must AddRef and Release be called manually by the developer ?
      1.5.2.3. What happens when the reference count drops to 0 ?
      1.5.2.4. How does the tools mechanism for implementing reference counting affect flexibility etc?

1.6. Threading
   1.6.1. Which Threading models are supported ?
      1.6.1.1. Not specified ?
      1.6.1.2. Single ?
      1.6.1.3. Appartment ?
      1.6.1.4. Free ?
      1.6.1.5. Other:
   1.6.2. Multi threaded components:
      1.6.2.1. Can a single component be multi-threaded?
      1.6.2.2. If no, is the support embedded in the language or is it near API ?

1.7. How is error handling supported by the tool
   1.7.1. Implementation of Error interfaces
      1.7.1.1. ISupportError ?
      1.7.1.2. IObjectError ?
   1.7.2. Direct support for HRESULT
      1.7.2.1. Use of existing HRESULT’s ?
      1.7.2.2. Interface dependent HRESULTS ?
      1.7.2.3. Does the tool provide support for HRESULT creation ?
      1.7.2.4. Is it easy to create/use HRESULTS ?
1.7.3. Are there mapping between HRESULTS and native language exceptions?
   1.7.3.1. Manual mapping?
   1.7.3.2. Automatic mapping?

1.7.4. Describe in words how the mapping works:

1.8. Data types
   1.8.1. How are the development tools data-types mapped to COM’s data types?
      1.8.1.1. Simple datatypes?
      1.8.1.2. Strings?
      1.8.1.3. Predefined types?
      1.8.1.4. User-defined types?
      1.8.1.5. Object parameters?
      1.8.1.6. Variants?
      1.8.1.7. SafeArrays?

1.9. Marshalling
   1.9.1. How can marshalling be handled:
      1.9.1.1. Don’t know!? 
      1.9.1.2. OLE32.DLL?
      1.9.1.3. Own Proxy/Stub DLL?
         1.9.1.3.1. Can it be attached to the component DLL?
      1.9.1.4. Directly made by tool?
      1.9.1.5. Work around?

1.10. How does the server support typelibraries
   1.10.1. IDL Integration?
      1.10.1.1. IDL/ODL?
   1.10.2. Implicit creation of typelibraries?
   1.10.3. Integration of typelibraries as resource in server?
   1.10.4. Are errors in the type library able to "cheat" the tools?

2. Compiling and Distribution
   2.1. Version compatibility
      2.1.1. Manual?
      2.1.2. Automatic?
   2.2. Interface compatibility
2.2.1. Is it possible to change an interface and keep the same IID?

2.3. Target types
2.3.1. Interpreted?
2.3.2. Optimization?

2.4. How does the tool support registration of components?
2.4.1. Components are selfregistering? If Yes then
2.4.1.1. DLL RegSvr32?
2.4.1.2. EXE: selfregistering when runned once?
2.4.1.3. EXE: selfregistering when compiled?
2.4.2. Registration is done through
2.4.2.1. Installation?
2.4.2.2. Has to be done manually?
2.4.3. Does the tool support unregistration of components?

2.5. Support for Component Categories?
2.5.1. Is it possible to define and implement your own component categories?
2.5.1.1. How?

3. Reuse of Controls
3.1. Support for COM clients
3.1.1. Creating and deleting components
3.1.1.1. Is it easy to create new component?
3.1.1.2. Is it easy to delete existing component?
3.1.1.3. Does the tool update registry entries for deleted components after recompilation?
3.1.1.4. Does the tool release component instances?
3.1.2. COM Library Support
3.1.2.1. Does the tool allow direct access to COM functionality, like calling COM functions direct (CoCreateInstance, CreateInstance, …)?
3.1.2.2. Does the tool encapsulate COM functionality indirect COM support?
3.1.2.3. Does the tool provide COM Helper Classes?
3.1.2.4. Is it possible to combine direct and indirect COM support?
3.1.3. Acquisition of interfaces
3.1.3.1. Is it possible to call QueryInterface directly?
3.1.3.2. Does the tool encapsulate QueryInterface?
3.1.3.3. Reference counting?
3.1.4. How is component binding supported

3.1.4.1. Early binding?

3.1.4.1.1. VTable binding?
3.1.4.1.2. Dispid binding?

3.1.4.2. Late binding?

3.1.5. Error handling

3.1.5.1. Direct?
3.1.5.2. Indirect?

3.2. Aggregation

3.2.1. Can the tool control COM-aggregation for the objects? If yes, Supported methods

3.2.1.1. Not aggregatable?
3.2.1.2. Is the support Wizard oriented (GUI)?
3.2.1.3. Is Manual programming Easy/Difficult?

3.3. Delegation

3.3.1. Support for delegation:

3.3.1.1. Manual?
3.3.1.2. Tool?

4. Development Environment support

4.1. Description and evaluation of supporting tools/facilities in the development tool

4.1.1. Object browser?

4.1.2. Syntax help/info for COM-components and methods herein?

4.1.3. Context sensitive help?

4.1.4. Support for mandatory interface prototypes?

4.1.4.1. Manually made?

4.1.4.2. Made by the tool. Filling out the slots?

4.1.5. Debugging

4.1.5.1. Which facilities exist for debugging COM-components?

4.1.5.2. Is it possible in the development to debug form client into component?

4.1.5.3. If any, what are the differences between in-process and out-of-process servers?

4.2. Evaluation of the development tool

4.2.1. A description of the work load
4.2.1.1. Does the COM-based part of the development integrate nicely and consistently into the tool?

4.2.1.1.1. Is it easy to find out how to do COM-based development in the tool?

4.2.1.1.2. Is the COM-based development in the tool similar to non-COM-based development?

4.2.1.2. Evaluation of whether it is easy for novices/experts with respect to the COM-development in general/the development tool specifically to do COM-based development in the tool.

5. Development Processes

5.1. Is it possible to develop and test at the same time?

5.1.1. Will recompilation require change of server GUID and IID’s?

13.2 TAXONOMY ANSWERS FOR DELPHI

(1.1.1) You can make ActiveX controls from all native Delphi visual controls i.e. buttons, forms, grids etc. Delphi will automatically provide a skeleton file in which you will simply fill out the implementations of properties and methods.

(1.1.2) All kinds of COM components from IUnknown to IDispatch based ones can be made.

(1.2.1) Yes
(1.2.2) Yes
(1.2.3.1) No
(1.2.3.2) No
(1.2.3.3) You must create a new project to make the transition. Delphi supports a wizard that will initially let you specify the kind of server you want to create. To change the type you will start a new project of that kind and then include the implementation units from the old project.

(1.2.3.4) No
(1.2.4) Yes
(1.2.5) No
(1.3.1.1) No
(1.3.1.2) Yes
(1.3.1.2.1)

Delphi has seven wizards that you must acquaint yourself with, when you want to start making components. Six of these are shown on Figure 38, and the seventh is simply the ordinary executable wizard.

To choose between in-process and out-of-process you will select either

Application: Starts a new empty project for an ordinary executable (EXE) • ActiveX Library: Starts a new empty project for a library (DLL). The name of the wizard is misleading in that simple COM
servers that do not implement IDispatch or any ActiveX specific interface will also be created this way.

When a new application type has been defined you will then select a new component type. This can be one of the following:

**Automation Object:** A simple COM component that can derive from IUnknown, IDispatch, and a number of others including the ones that you have defined yourself. Following the definition of the name of your new component, the wizard will open the Type Library Editor as shown on Figure 40 (TypeLibEd) and insert an interface and CoClass definition for you.

**ActiveForm:** An ActiveX control that represents an entire visual form, on which you can drop native Delphi controls. In all other respects, this is an ActiveX control.

**ActiveX Control:** ActiveX control that derives from a native Delphi control other than a form. This can be a button, a grid etc.

**Property Page:** A Delphi wrapper for the property pages technology

**Type Library:** Opens the Type Library Editor with an empty library. You will typically not use this wizard when creating component, but rather define them through the Automation Object Wizard

(1.3.1.2.2) When you first create a new component using the Automation Object wizard, you will be asked for the name of the new control and how it should be allowed to be instanced. The possible choices are single, multiple or internal. Internal specifies that no external application is allowed to use the component, whereas single and multiple are self-explanatory.

The Type Library Editor will be opened next, and from here you may set all kinds of properties on the component. These correspond to flags and attributes in IDL. There does not seem to be any restrictions on the attributes, but you can only create methods, properties, interface, CoClass and enumerations from the editor. Struct, unions and modules can be shown by opening an existing type library, but not added.

(1.3.1.2.3) Once the library is fully defined, you will register it on the button in the upper right corner of the Type Library Editor. This will create the necessary entries in the Windows Registry and create a number of source files. Each CoClass will get its own unit, which contains skeleton source for all methods and properties in the implemented interfaces as well as initialisation code for a corresponding Class Factory. Furthermore, a file containing pascal definitions of CoClass GUIDS, Interfaces, constants, structures and unions. This file has ".TLB" appended to its name.

(1.3.1.2.4) You cannot alter the _TLB file, since this will be auto-generated each time the type library changes via the editor. All other code can be customised as you see fit.

(1.3.1.2.5) If you redefine method or property definitions in the Type Library Editor following an implementation of some of them, your code will not be harmed.

(1.3.1.2.5.1) Yes

(1.3.1.2.5.2) No. Delphi has its own way.

(1.3.1.2.5.3) The only file you cannot customise to your own preferences is the _TLB type library wrapper file.

(1.3.1.2.6) Overall the generated code is ok, and performs fine. There is a problem, though.
(1.3.1.2.6.1) Sometimes the Type Library Editor changes its definitions of methods and properties in the _TLB file. The one minute you will have code that looks like:

```delphi
function MyFunc(V1: Integer): Boolean;
```

After a refresh, however you may end up with something that looks like:

```delphi
function MyFunc(V1: Integer; var V2: Boolean): HRESULT;
```

The CoClass skeleton declarations are not updated, and so you end up with a project that cannot compile unless you manually redefine the CoClass skeleton files. We have not been able to pinpoint the situation in which this problem occurs.

(1.3.1.2.6.2) The Type Library editor has a number of known bugs that will eventually be fixed by Borland. A report is available from Borland’s homepage.

(1.3.2) If you want to define the entire project yourself, there is some work to do, but it is not impossible. You will have to define the project file, the CoClass skeleton files and the type library by hand, IDL, or Type Library Editor. The CASE Tool IDUN autogenerates these files from a type library description.

(1.3.2.1) It is not worth the while, but can be done.

(1.4.1.1) Yes

(1.4.1.2) Yes

(1.4.1.3) Yes

(1.4.2) Use the Type Library Editor to add a new interface. Then add methods and properties as appropriate. An interface declaration in Delphi syntax will then be created automatically. You cannot add interfaces to the _TLB file yourself, since this is created from the editor.

(1.4.3) Yes

(1.4.3.1) Add a new interface via the editor. Select the CoClass to implement it. Select the "members tab". Right click the window. Select "insert interface" from the pop-up-menu.

(1.4.4.1) It is difficult. Delphi will let you open .TLB files, which will result in a new instance of the Type Library Editor showing the library. Drag-and-drop is somewhat supported between instances of the editor, but usually crashes it. You can add references to existing type libraries (on the "members" tab of the library), but for some reason this changes nothing with respect to what you can implement.

(1.4.4.2) It is possible to "cheat" Delphi into believing that a type library on the disk does in fact belong to a native Delphi project. Delphi always includes the type library as a Windows ressource in the final object server by means of its SR compiler directive. It uses the name of the project to identify the type library binary on the disk, and so if you save that .TLB file under the appropriate name, your troubles are over. You will just have to implement the CoClass skeleton files yourself.

(1.4.4.2.1) Yes and cheat as above.

(1.4.4.2.2) Possible, but you will need the MIDL compiler and cheat as above.
(1.4.4.2.3) No
(1.4.4.2.4) No
(1.4.4.3) Since it will ultimately be part of the project as any other type library defined through the wizards, there is no difference here.
(1.4.5.1) Yes. Delphi does not distinguish between published and non-published interfaces.
(1.4.5.2) Yes
(1.4.5.3) Yes
(1.4.5.4) Yes
(1.4.5.5) When adding entities, corresponding Pascal declarations will be created. When you remove an entity, e.g. an interface, the CoClass units will not be cleaned up, but the _TLB file will.
(1.4.6.1) Yes
(1.4.6.2) ??
(1.4.6.3) No
(1.5.1) ??
(1.5.1.1) ??
(1.5.1.2) ??
(1.5.2) Delphi will implicitly perform and AddRef(), when assignment is made on an interface variable as for example in

```pascal
var
    MyCalc: ICalc;
begin
    MyCalc := CreateComObject(CLASS_CALC) as ICalc;
    ...
end;
```

In the example the "as" keyword performs an implicit "QueryInterface" for the ICalc interface. "CreateComObject" is a wrapper function for "CoCreateInstance", but Delphi does not force you to use that wrapper function. The above code could also have been written:

```pascal
var
    MyCalc: ICalc;
begin
    CoCreateInstance(CLASS_CALC, nil, CLSCTX_INPROC_SERVER, ICalc, MyCalc);
    ...
end;
```

(1.5.2.1) Yes
(1.5.2.2) No, lest he uses the API functions.
(1.5.2.3) The component is released.
(1.5.2.4) They don't: you can choose the strategy of your preference.

(1.6.1.1) -
(1.6.1.2) Yes
(1.6.1.3) No
(1.6.1.4) No
(1.6.1.5) No

(1.6.2) Delphi has a native wrapper class for Windows threads, the TThread class. You can use this in any Delphi program, including components.

(1.6.2.1) Yes

(1.6.2.2) You can also use the API if you prefer.

(1.7.1) All COM components in Delphi descend from the TComObject class, which implements ISupportErrorInfo as well as IUnknown.

(1.7.1.1) Yes
(1.7.1.2) No

(1.7.2) Delphi wraps HRESULTS into native exceptions.

(1.7.2.1) Yes, the constants are declared in the include file "ActiveX.pas"
(1.7.2.2) You can create your own as needed.
(1.7.2.3) No.
(1.7.2.4) Easy to use, harder to define.

(1.7.3) Yes, as described above.

(1.7.3.1) Yes
(1.7.3.2) Yes

(1.7.4) Components that return errorcodes or exceptions through IErrorInfo interfaces, will have their errors translated into native Delphi exceptions. A base class for these are the EOleException exception class.

(1.8.1) See below

(1.8.1.1) The following table lists the mapping of simple data-types

<table>
<thead>
<tr>
<th>COM Constant</th>
<th>Delphi Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VT_I2</td>
<td>SmallInt</td>
<td>2 byte signed int</td>
</tr>
<tr>
<td>VT_I4</td>
<td>Integer</td>
<td>4 byte signed int</td>
</tr>
<tr>
<td>VT_R4</td>
<td>Single</td>
<td>4 byte real</td>
</tr>
<tr>
<td>VT_R8</td>
<td>Double</td>
<td>8 byte real</td>
</tr>
<tr>
<td>VT_CY</td>
<td>Currency</td>
<td>Currency</td>
</tr>
<tr>
<td>VT_DATE</td>
<td>TDateTime</td>
<td>Date</td>
</tr>
</tbody>
</table>
### Software Components and Development Tools

<table>
<thead>
<tr>
<th>VT_DISPATCH</th>
<th>IDispatch</th>
<th>IDispatch FAR*</th>
</tr>
</thead>
<tbody>
<tr>
<td>VT_ERROR</td>
<td>SCODE</td>
<td>SCODE</td>
</tr>
<tr>
<td>VT_BOOL</td>
<td>WordBool</td>
<td>True=-1, False=0</td>
</tr>
<tr>
<td>VT_UNKNOWN</td>
<td>IUnknown</td>
<td>IUnknown FAR*</td>
</tr>
<tr>
<td>VT_DECIMAL</td>
<td>TDecimal</td>
<td>16 byte fixed point</td>
</tr>
<tr>
<td>VT_I1</td>
<td>ShortInt</td>
<td>signed char</td>
</tr>
<tr>
<td>VT_UI1</td>
<td>Byte</td>
<td>unsigned char</td>
</tr>
<tr>
<td>VT_UI2</td>
<td>Word</td>
<td>unsigned short</td>
</tr>
<tr>
<td>VT_UI4</td>
<td>UINT</td>
<td>unsigned long</td>
</tr>
<tr>
<td>VT_I8</td>
<td>Comp</td>
<td>signed 64-bit int</td>
</tr>
<tr>
<td>VT_UI8</td>
<td>LargeUINT</td>
<td>unsigned 64-bit int</td>
</tr>
<tr>
<td>VT_INT</td>
<td>SYSINT</td>
<td>signed machine int</td>
</tr>
<tr>
<td>VT_UINT</td>
<td>SYSUINT</td>
<td>unsigned machine int</td>
</tr>
<tr>
<td>VT_HRESULT</td>
<td>HResult</td>
<td>HResult</td>
</tr>
</tbody>
</table>

(1.8.1.2) Strings are mapped as follows:

<table>
<thead>
<tr>
<th>COM Constant</th>
<th>Delphi Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VT_BSTR</td>
<td>WideString</td>
<td>16 bit wide Automation string</td>
</tr>
<tr>
<td>VT_LPSTR</td>
<td>PChar</td>
<td>Null terminated C-style</td>
</tr>
<tr>
<td>VT_LPWSTR</td>
<td>PWideChar</td>
<td>Null terminated 16 bit C-style</td>
</tr>
</tbody>
</table>

(1.8.1.3) Predefined types are not possible to create via the Type Library editor. However, if a type library that contains structs of typedefs is loaded, these will translated in the _TLB file. Structs will be mapped into records and simple typedefs will be substituted. Enums will be mapped into constants.

(1.8.1.4) Only enums can be created, and these are mapped as described above.

(1.8.1.5) Will be mapped as described in 1.8.1.3. When a type is not simple (e.g. struct), its name will simply be put in as the datatype in the parameter list.

(1.8.1.6) Variants are supported by Delphi, and here known as "OleVariant".

(1.8.1.7) Safearrays will be mapped into the TSafeArray class, which is a full wrapper for safearrays.

(1.9.1.1) See below.

(1.9.1.2) Yes, if you provide dual interfaces or ones based on the automation types.

(1.9.1.3) Yes, but you must use MIDL to get the proxy/stub files and then a C-compiler to build it.
(1.9.1.3.1) No, use MIDL as described above.
(1.9.1.4) No.
(1.9.1.5) No.
(1.9.2) You can provide your own implementation of IMarshal if you care.
(1.10.1) No
(1.10.1.1) No.
(1.10.2) Through the Type Library Editor.
(1.10.3) Yes
(1.10.4) No, the compiler will (and does) complain.
(2.1.1) Yes
(2.1.2) No
(2.2.1) Yes
(2.3.1) You can build COM servers that will need the Delphi runtime libraries. These servers are very small, compared to ones that link the runtime libraries into them.
(2.3.2) The optimisations that Delphi supports for all programs does also apply to components.
(2.4.1) Delphi components that are out-of-process will register themselves automatically when executed as if they were ordinary programs. In-process servers can be registered through REGSVR32.EXE.
(2.4.1.1) Yes
(2.4.1.2) Yes
(2.4.1.3) No
(2.4.2.1) No
(2.4.2.2) Yes
(2.4.3) Yes. In-process servers may be unregistered either from the menues in Delphi or via REGSVR32.EXE, using the /u option. Out-of-process servers can be registered by running them once and unregistered again by supplying the switch "/unregserver" on the commandline.
(2.5.1) Delphi has no direct support for grouping components into categories.
(2.5.1.1) -
(3.1.1.1) Yes: use the wizards.
(3.1.1.2) Yes: use the Type Library Editor.
(3.1.1.3) Yes, but not until the component has been re-registered.
(3.1.1.4) Yes.
(3.1.2.1) Yes, but you will not get the benefits of automatic reference counting if you do so.
(3.1.2.2) Yes, as described in 1.5.2.
(3.1.2.3) Yes. There are a number of classes, encapsulating simple COM servers, automation servers, and even Class Factories.

(3.1.2.4) Yes.

(3.1.3.1) Yes.

(3.1.3.2) Yes, using the "as" operator.

(3.1.3.3) Implicit through assignment or explicit via AddRef and Release.

(3.1.4.1) Yes.

(3.1.4.1.1) Yes.

(3.1.4.1.2) Yes.

(3.1.4.2) Yes.

(3.1.5) Error handling is supported through Delphi exceptions. The mapping from HRESULTs or IErrorInfo is implicit, but can also be used explicitly.

(3.1.5.1) Yes.

(3.1.5.2) Yes.

(3.2.1) All components will have native support for being the inner of an aggregate. If you want to create a component that aggregates another one, you will have to override the default implementation of QueryInterface.

(3.2.1.1) -

(3.2.1.2) No.

(3.2.1.3) Yes. The implementation of QueryInterface can be defined as shown below. You will also need to create the aggregates in the initialization procedure. This procedure is automatically called by the Class Factory, when a new instance of a Delphi COM class is made.

```delphi
procedure TClass2.Initialize;
var
  Master: IUnknown;
begin
  if (Controller = nil) then
    Master := Self
  else
    Master := Controller;

  { Initialize aggregated components }
  OleCheck(CoCreateInstance(Class_ItemInfo, Master, CLSCTX_INPROC_SERVER, IUnknown, punkInner));
end;

function TClass2.ObjQueryInterface(const IID: TGUID; out Obj): Integer;
begin
  if GetInterface(IID, Obj) then
    Result := S_OK
  else if (pUnkInner.QueryInterface(IID, Obj) = S_OK) then
Result := S_OK
else
  Result := E_NOINTERFACE;
end;

(3.3.1.1) Yes. It is quite simple to add delegates to a COM class: Simply create the instances in the Initialize procedure and forward the calls in the method implementations as shown below. TClass3 contains CoClass Calc, and delegates calls on the ICalc interface as shown in the implementation of Modulus.

procedure TClass3.Initialize;
begin
  { Launch contained components }
  Class3ICalc := CreateComObject(Class_Calc) as ICalc;
end;

function TClass3.Modulus(v1: Integer; v2: Integer): Integer; safecall;
begin
  Result := Class3ICalc.Modulus(v1, v2);
end;

(3.3.1.2) No.

(4.1.1) The Type Library Editor is useful for browsing type libraries, and will let you import these from the Windows Registry or from a file.

(4.1.2) There is no way to use a help-file attached to a component from Delphi, but help-strings will be added as comments in auto-generated code.

(4.1.3) Available on most API functions and some standard interfaces.

(4.1.4) Delphi provides default implementations of IUnknown, IDispatch, ISupportErrorInfo, and some other interfaces. You can override these implementations at will.

(4.1.4.1) Is possible

(4.1.4.2) No slots: the implementations are hidden in super-classes.

(4.1.5) For in-process servers, you can specify an application that will be used to debug the component. You can then debug as you would ordinary Delphi applications, using breakpoints and watches in your component. To debug an out-of-process server, it must be launched before any clients of it. If you launch it from Delphi, you can then set breakpoints and watches as usual.

(4.1.5.1) The ordinary Delphi debugger.

(4.1.5.2) Yes. You can have multiple instances of Delphi running, and can connect them as described above.

(4.1.5.3) See 4.1.5

(4.2.1) -

(4.2.1.1) Yes. All Delphi applications are made using wizards, and so are COM components. When you have defined the type library, the rest is ordinary Delphi programming.
(4.2.1.1.1) The help files of Delphi in general lack examples, but by being a little stubborn, it is not so hard.

(4.2.1.1.2) Yes. Apart from the Type Library Editor, which is easy to use.

(4.2.1.2) When you have figured out to use the wizards, and have prior knowledge of the COM entities (interface, CoClass, etc.), it is easy to use Delphi. An expert may override Borland’s standard implementations of interfaces and provide his own. All COM support is wrapped up in Delphi units that are shipped with the tool in both compiled and source-code form. It is therefore a question of reading the source and patching it appropriately.

(5.1) Yes.

(5.1.1) No.