Combining Visual Modelling with Visual Programming for CORBA Component Development

Frank Stefan Bühler

De Montfort University Leicester
Faculty of Computing Sciences and Engineering
De Montfort University, The Gateway
Leicester LE1 9BH
England

3 Volumes (Thesis and Appendices)

THESIS (VOLUME 1)

Dissertation submitted for the degree Doctor of Philosophy

May 2002
Abstract
One of the many issues that has arisen in the process of using efficient and state-of-the-art techniques and software architectures is the introduction of object-/component-oriented methods. Another general trend is the integration of visual elements and techniques in tool development environments.

This thesis identifies several approaches for improving the effectiveness of current development by introducing different visual programming techniques. One promising idea, in order to decrease the complexity of software development and to improve the efficiency for novice and inexperienced users, is the creation of a complete visual system that makes use of either visual modelling or visual programming techniques and thus reduces the gap between different representations and paradigms. The goal of this dissertation is to investigate why there has been little research in the field of visual object-oriented programming and the integrated combination of visual modelling with visual programming in order to find new ways to improve this situation. Of specific interest is the improvement and support of component-based development, as this will play a central role in modern software development in the near future.

A prototype system for the development of CORBA components has been developed that integrates visual programming with visual modelling. In particular, a new visual programming language called VOOPL has been invented. The language is based on the processor idea, which is directly related to a visual control-flow paradigm. The visual modelling approach is based on standard UML models (namely class diagrams plus specific stereotypes). The system has been tested by informal usability trials. The results of evaluations indicate that this is an effective approach for novice and inexperienced users but less so for professional users. One key problem identified is the lack of a standardised visual programming language that would decrease the learning curve for a complete visual (object-oriented) approach. The visualisation of code elements through the use of so-called processors has proven to be accepted by users. Processor code is useful for showing the overall code structures but less efficient for entering control logic. Therefore, a hybrid approach (i.e. the combination of visual and textual programming) is suggested. The findings of this research clearly show that there are benefits when combining visual modelling with visual programming.
Acknowledgements

I would like to thank Paul Reeves and Sarah Lingard for their support and fruitful discussions during the long period of research. Special thanks to my supervisors Paul Luker and Michael Callaghan for giving me the opportunity to carry out this PhD work. I am grateful for their support and motivation.

During the last two years I have benefited tremendously from the many discussions with my colleagues at Dresdner Bank AG, Frankfurt/Main (Germany). They taught me what is important for real-life projects and broadened my horizons in many different ways. In addition, I would like to thank those colleagues who participated in the evaluation studies. Finally, the work would have never been possible without the encouragement and tolerance of my wife Anja Zink-Bühler and my lovely children.

Declaration

I hereby declare that this PhD thesis is a record of work undertaken by myself and is not being submitted concurrently in candidature for any other degree.

Frank Bühler
Leicester
February 27, 2001
Table of Contents

1 Introduction ........................................................................................................... 13
  1.1 Motivation and Research Context ................................................................ 13
  1.2 Statement of Research Methods Applied ...................................................... 14
  1.3 Overview of the Thesis .................................................................................. 16
2 Background Information ......................................................................................... 18
  2.1 Influence of Visual Technology ................................................................... 18
  2.2 Object Technology ....................................................................................... 19
    2.2.1 Introduction ............................................................................................... 19
    2.2.2 Basic Concepts ......................................................................................... 20
    2.2.3 Common Pitfalls of OO development ....................................................... 22
  2.3 Component Technology .................................................................................. 23
    2.3.1 Introduction ............................................................................................... 23
    2.3.2 CORBA ...................................................................................................... 24
      2.3.2.1 Introduction .......................................................................................... 24
      2.3.2.2 Basic Concepts ..................................................................................... 25
      2.3.2.3 Overview of Commercial Products ..................................................... 27
      2.3.2.4 Common Pitfalls .................................................................................. 30
  2.4 Visual Modelling .............................................................................................. 31
    2.4.1 Introduction ............................................................................................... 31
    2.4.2 Basic Concepts of UML ............................................................................ 31
    2.4.3 Current Trends and Possible Improvements ............................................ 35
  2.5 Visual Languages ............................................................................................. 37
    2.5.1 Overview of Visual Languages ................................................................ 37
    2.5.2 Overview of Visual Programming Languages ......................................... 38
    2.5.3 Graphical Notations for Program Design ................................................ 42
    2.5.4 Visual programming (VP) requirements and problems ............................ 46
    2.5.5 Visual Object-Oriented Programming ..................................................... 47
    2.5.6 The definition of a VPL ........................................................................... 50
    2.5.7 Vampire : A Visual Metatool for Programming Iconic Environments ....... 53
    2.5.8 VLCC: A Compiler-Compiler for Visual Languages ............................... 55
    2.5.9 Claims and Limitations for VPLs ............................................................. 56
3 Related Research in the VL Field ........................................................................... 58
  3.1 Overview .......................................................................................................... 58
  3.2 Survey 1: Review of Experimental Findings from Related Topics .................... 58
  3.3 Survey 2: Visual Object-Oriented Programming Languages ......................... 64
    3.3.1 Overview ................................................................................................. 64
    3.3.2 Vista ......................................................................................................... 64
    3.3.3 Ispel ......................................................................................................... 66
    3.3.4 VIPR ........................................................................................................ 67
    3.3.5 Prograph .................................................................................................. 69
Combining Visual Modelling with Visual Programming for CORBA Component Development

4 Overview of Different Approaches

4.1 Approach 1: Visual Meta Builder
4.1.1 Background and Motivation
4.1.2 Basic Idea
4.1.3 Language Design
4.1.4 Introduction to the Approach
4.1.5 The Underlying Theory
4.1.5.1 Program Representation by a Graph
4.1.5.2 Implementation and Representation of a Graph Node

4.2 Approach 2: Extending UML for Dialog Processing
4.2.1 Background and Motivation
4.2.2 Basic Idea
4.2.3 Additional UML diagrams
4.2.4 Dynamic Modelling
4.2.4.1 Pre-defined Application Structures
4.2.4.2 Statecharts For Structuring an Application
4.2.5 Development Environment and VOODE models

4.3 Approach 3: Framework-based Development
4.3.1 Background and Motivation
4.3.2 Basic Idea
4.3.2.1 Design Aspects of the VOOPPL-I Editor

4.4 Approach 4: Component-based Development
4.4.1 Background and Motivation
4.4.2 What Can Be Improved?
4.4.3 Basic Idea

4.5 Critical Discussion of the Different Approaches

4.6 Justification For the Approach to Combine Visual Modelling With Visual Programming

5 VOODE/VOOPL-I for CORBA

5.1 Objectives
5.2 Introduction and Motivation

5.3 Design Aspects of VOODE/VOOPL-I
5.3.1 Processor Metaphor and VOOPL-I syntax
5.3.2 Design aspects of the VOOPL-I editor and its integration in Rational Rose
5.3.3 Visual Feedback
5.3.4 The VOODE/VOOPL-I architecture / code generation
5.3.5 UML - VOOPL-I Mapping
5.3.6 Working with VOODE/VOOPL-I
5.3.6.1 How to model a CORBA component?
5.3.6.2 The VOOPL-I editor
5.3.6.3 IDL file generation
5.3.6.4 Implementation of CORBA server mainline
5.3.6.5 Implementation of CORBA methods
5.3.6.6 Code Generation
List of Figures

Figure 1. The "4 + 1" view of architecture [QUATRANI-1998], [KRUCHTEN-2000] .................................... 33
Figure 2. Overview of the Rational Unified Process [RATIONAL-2000, © Rational Corp.] .................. 34
Figure 3. Construction of iconic sentences [CHANG-1987]................................................................. 38
Figure 4. Different graphical program notations .................................................................................. 44
Figure 5. A sample processor. ............................................................................................................. 45
Figure 6. Classification of visual object-oriented programming [BURNETT-1995, p. 5] .......................... 48
Figure 7. Basic "architecture" of a Visual Object-Oriented Development Environment (VOODE) ........ 49
Figure 8. The terminal NAPEs Start, End, If (Cond.), Connector, Print. ............................................. 52
Figure 9. One sample grammar production. ........................................................................................ 53
Figure 10. A visual timer construct and a list processor [BURNETT-1995, p 210]. ............................... 65
Figure 11. Representation levels in Ispel [GRUNDY-1991] .................................................................. 67
Figure 12. The definition of a main procedure with a while loop....................................................... 68
Figure 13. The definition of a class point. ......................................................................................... 69
Figure 14. The substitution metaphor. ............................................................................................... 74
Figure 15. The key-keyhole metaphor. ................................................................................................ 75
Figure 16. The building blocks metaphor. .......................................................................................... 75
Figure 17. Architecture of the visual environment. .......................................................................... 76
Figure 18. A processor icon represents a function call. ..................................................................... 77
Figure 19. An icon represents an object's method call. .................................................................... 78
Figure 20. VPL-C++: class/object handling plus event handling. ..................................................... 78
Figure 21. The general architecture of the transformation process. ................................................... 80
Figure 22. Sample application processor. ......................................................................................... 89
Figure 23. Overall structure of an application. .................................................................................. 90
Figure 24. Definition of the control/data flow. .................................................................................. 91
Figure 25. Example of the definition of a code block (sub-processors). ........................................... 92
Figure 26. General architecture of form specifications. .................................................................... 92
Figure 27. Actor-Form diagram. ........................................................................................................ 93
Figure 28. Designing the form using control element processors. .................................................... 94
Figure 29. Form-Use case diagram. .................................................................................................. 94
Figure 30. Linking the dialog panel to domain/architecture classes. .................................................. 95
Figure 31. Defining the method Calculation::Add(). ...................................................................... 96
Figure 32. Visual representation of a dialog-oriented application (static view). .................................... 97
Figure 33. Single panel application based on IlvDialog. ................................................................. 98
Figure 34. Single panel application based on IlvApplication. .......................................................... 99
Figure 35. A sample statechart "Application". .................................................................................. 101
Figure 36. A sample statechart "CWinApp". .................................................................................... 102
Figure 37. A sample statechart "CCalculatorDlg". ............................................................................ 102
Figure 38. A sample statechart "Dialog Processing". ....................................................................... 103
Figure 39. VOODE framework and sample "slot charts". ............................................................... 104
Figure 40. Defining the action "load calculation object". ................................................................ 104
Figure 41. Defining the method VOODE_Container::AddButtonCB(). ........................................ 105
Figure 42. Description files needed to maintain the definition links. ................................................ 105
Figure 43. Using inheritance with a base class. .............................................................................. 106
Figure 44. Using delegation to use a base class .............................................................................. 107
Figure 45. The complete visual object-oriented development system ............................................ 108
Figure 46. Necessary VOODE components. .................................................................................... 109
Figure 47. The layout of a standard framework. ............................................................................... 111
Figure 48. Defining the method Calculation::Add(). ...................................................................... 111
Figure 49. Sample dialog panel to select/add a class method with attributes. ................................. 112
Figure 50. How visual code may be linked to a framework slot...............................................................113
Figure 51. Sample function slot............................................................................................................114
Figure 52. Sample main slot ........................................................................................................................115
Figure 53. Sample textual code .............................................................................................................116
Figure 54. Corresponding visual code ..........................................................................................................116
Figure 55. Representing pre-defined code parts and objects........................................................................117
Figure 56. Navigational links to the VOODE/VOOPL-1 editor .................................................................118
Figure 57. Sketch of the VOODE/VOOPL-1 editor .......................................................................................118
Figure 58. Mapping the Visual (Interaction) Model to the Textual (Interaction) Model ..........................127
Figure 59. A sample "Add" processor with three attributes...........................................................................128
Figure 60. Design of the Visual Editor .........................................................................................................130
Figure 61. The VOODE/VOOPL-1 architecture .........................................................................................133
Figure 62. The translation step from processor code to IDL ........................................................................134
Figure 63. The pre-defined CORBA component framework within Rose ..............................................135
Figure 64. The CORBA component within Rose .........................................................................................136
Figure 65. Main entry to start VOODE/VOOPL-1 editor ...........................................................................137
Figure 66. Main menu of the VOODE/VOOPL-1 editor ............................................................................137
Figure 67. Processor-based visual IDL definition .........................................................................................138
Figure 68. Typical server mainline code (C++) .............................................................................................139
Figure 69. Mapping typical server mainline code to processors ..............................................................140
Figure 70. Visual definition of CORBA Server mainline: OnServerInit ....................................................140
Figure 71. Visual definition of a CORBA implementation method ............................................................141
Figure 72. Overview of Implementation Files .............................................................................................142
Figure 73. D.A. Norman's three models ......................................................................................................145
Figure 74. Background knowledge ............................................................................................................150
Figure 75. CORBA Component: managing an array of long values .........................................................158
Figure 76. VPL-C: Defining the project .......................................................................................................207
Figure 77. Defining the application's structure ...........................................................................................207
Figure 78. Defining a function and the main entry ......................................................................................208
Figure 79. Programming the main function ...............................................................................................208
Figure 80. The linkage of the VOODE/VOOPL-1 Editor to UML diagrams ..............................................221
Figure 81. The invocation of the VOODE/VOOPL-1 Editor from class diagrams ...................................222
Figure 82. The invocation of the VOODE/VOOPL-1 Editor from sequence diagrams ................................223
Figure 83. The additional specification of the CORBA server mainline ...................................................224
Figure 84. The IDL specification of the component ...................................................................................225
Figure 85. The invocation of the VOODE/VOOPL-1 Editor ....................................................................226
Figure 86. The extension of Rose to include additional features ...............................................................227
Figure 87. Overview of the User Interface .................................................................................................229
Figure 88. CRC Cards for Capturing Business Entities .............................................................................230
Figure 89. Panel Definitions .......................................................................................................................231
Figure 90. Static View: Object Definition and Modelling ..........................................................................232
Figure 91. Dynamic View: Visual Programming with Processors ............................................................233
Figure 92. Framework View: Slot Definitions .............................................................................................234
Figure 93. Overview of Method Specifications .........................................................................................235
Figure 94. The Visual Age for C++ development environment ...............................................................238
Figure 95. The visual builder tool ..............................................................................................................239
Figure 96. Definition of a new visual part ....................................................................................................239
Figure 97. The composition editor ............................................................................................................240
Figure 98. The definition of the user interface ...........................................................................................241
Figure 99. The setting panel for control elements .....................................................................................241
Figure 100. The definition of a default button .............................................................................................242
List of Tables

Table 1. Important concepts of object-oriented development .......................................................... 21
Table 2a. Important concepts of CORBA development (part 1) ..................................................... 25
Table 2b. Important concepts of CORBA development (part 2) ..................................................... 26
Table 3a. List of JBuilder 4 Features (part 1) ................................................................................. 28
Table 3b. List of JBuilder 4 Features (part 2) ................................................................................. 29
Table 4. List of important UML elements. .................................................................................... 32
Table 5. List of important CASE tool features ............................................................................. 35
Table 6. Visual Programming (cf. [SHU-1988]) ............................................................................ 39
Table 7. A chronology of influential computer languages. ............................................................. 41
Table 8. Main areas in the Department of Computer Science at DMU [DMU-2000]. .................. 41
Table 9. Survey of graphical program notations. ......................................................................... 43
Table 10. Commercial products including a VPL ........................................................................ 64
Table 11. The different connection types within VisualAge for C++. ......................................... 71
Table 12. Overview of different approaches identified by this research. .................................... 72
Table 13. The basic node modelling language of VMB. ............................................................... 87
Table 14. Predefined implementation contexts ............................................................................ 129
Table 15. Predefined implementation contexts. ......................................................................... 131
Table 16. Visual feedback coming from the tree view. ................................................................. 132
Table 17. Visual feedback coming from the workplace ............................................................... 132
Table 18. Overview of UML/VOOPL-1 mapping. ....................................................................... 134
Table 19. Design of user trials 1. ................................................................................................. 148
Table 20. Questionnaire. ............................................................................................................. 149
Table 21. Background knowledge of subjects. ............................................................................ 150
Table 22. Results for Task 1. ...................................................................................................... 152
Table 23. Results on Task 2 after training. .................................................................................. 153
Table 24. Results on motivation to use a language after Task 2. .............................................. 154
Table 25. Background knowledge of subjects. ............................................................................ 156
Table 26. Design of user trials 2. ............................................................................................... 157
Table 27. Judgement on Task 1 completion. ................................................................................ 159
Table 28. Judgement on Task 2 completion. ................................................................................ 160
Table 29. Judgement on Task 3 completion. ................................................................................ 160
Table 30. Judgement on Task 1 completion. ................................................................................ 161
Table 31. Judgement on Task 2 completion. ................................................................................ 162
Table 32. Judgement on Task 3 completion. ................................................................................ 162
Table 33. Important terms. ......................................................................................................... 183
Table 34. Abbreviations .............................................................................................................. 184
Table 35. List of connection types ............................................................................................... 237
Table 36: The basic node modelling language. ........................................................................... 268
Table 37. Task 1: Readability of VPL programs. ............................................................................ 281
Table 38. Task 1: Feedback of students. ...................................................................................... 282
Table 39. Task 2: C/C++/Java feedback of students. ................................................................. 283
Table 40. Task 2: Prograph feedback of students. ...................................................................... 284
Table 41. Task 2: VMB-C feedback of students. ....................................................................... 285
Table 42. Problems identified in task 1 (design A) ................................................................. 287
Table 43. Problems identified in task 2 (design A) ................................................................. 288
Table 44. Problems identified in task 3 (design A) ................................................................. 289
Table 45. Problems identified in task 1 (design B) ................................................................. 290
Table 46. Problems identified in task 2 (design B) ................................................................. 291
Table 47. Problems identified in task 3 (design B) ................................................................. 292
1 Introduction

1.1 Motivation and Research Context

Though many researchers (e.g. [SHU-1988], [GLINERT-1990], [ICHIKAWA-1990]) claim the benefits of visual languages, more than 30 years of research has led to many academic but only a few commercially available visual programming systems (e.g. Prograph [PICTORIUS-2000], LabView [NI-2000], and HP-VEE [HESEL-1998]). Furthermore, visual modelling languages like UML [UML-2000] have become, among other things, an important trend in commercial settings [FOWLER-1997]. Nowadays, CASE tool vendors provide many code generation facilities for various string languages (e.g. C++, Java, Visual Basic) including Round-trip Engineering features. To the best of the author's knowledge no academic or commercial system exists that makes use of both visual modelling and visual programming.

The goal of this dissertation is to investigate why this is the case in order to find new ways to improve this situation, thus trying to combine visual modelling with visual programming. Of specific interest is the improvement and support of component-oriented development as this will play a central role in modern software development in the near future. In order to evaluate the ideas in some depth CORBA has been chosen as the candidate component technology.

The study has also been motivated by the general trend in OOA/D/P, distributed computing and visual tools. There has been some research on visual object-oriented programming systems over the last 10 years (e.g. VIPR [CITRIN-1995], VISTA [SCHIFFER-1998]). Though OO, and especially Java as the main OOP language, has now become the major accepted paradigm for creating most of the new systems (even mission critical applications running on the host) the question arises why hasn't the visual language community made the effort to investigate the requirements for visual object-oriented programming languages and to create more powerful visual environments to use them. Since the underlying vision of this research has been to combine visual modelling with visual programming a new approach has been developed in order to learn more about the tradeoffs of such systems.

The key claims to be evaluated by this research are whether VPLs (Visual Programming Languages) are better than string languages per se, as claimed by many
VL (Visual Language) researchers, whether users will perform better in a complete visual environment that makes use of both visual modelling and visual programming, and how visual systems may reduce complexity with respect to CORBA component development.

Since UML is the dominant visual modelling notation, the research should focus on the standard use of UML, the development of a new visual system and the integration of both technologies. The investigations should be supported by a prototype system that demonstrates the usefulness and effectiveness of this approach.

1.2 Statement of Research Methods Applied

After setting out the research goals and studying the existing VL literature it soon became apparent that it is quite difficult to follow a strict scientific approach since research traditionally carried out in the HCI field or the study methods of Psychology of Programming are generally those of experimental psychology ignoring the real programming world. It was important to the success of the research to find a way to restrict the scope of the research so that it can be kept flexible enough to elaborate new ideas and at the same time be specific enough to be able to draw concrete conclusions from the work.

This section states what kind of research methods have been applied and why they have been chosen. This section therefore serves as background information in order to give the reader a basic understanding of the subsequent chapters.

The first step of this work was a survey of existing visual programming languages and the identification of existing claims for such languages by VL researchers. The conclusions from these activities were that there are only a few commercially available visual programming languages and the languages developed by researchers are limited to specific domains. Thus, it was decided to create a new visual language environment to test some ideas. The Visual Meta Builder which enabled the definition of different iconic languages, was developed. This was action research to deal with a concrete problem, viz. the definition of a visual programming language based on the control-flow paradigm. The underlying concept was proven by building a prototype system and making step-by-step adjustments to it. One outcome was a paper published in the Compsac '98 proceedings [COMPSAC-1998]. It was
shown that the translation of any context-free language into an iconic language is straightforward and may be supported by the Visual Meta Builder system.

Although the first step was successful, various shortcomings were identified. First of all it became clear that a complete visual programming language based on the so-called processor idea is not sufficient in order to build a large application system. One severe drawback, for instance, is the lack of efficient navigation techniques, and a process model which guides the user in building the intended system. Furthermore, the system did not directly address component-oriented development which is one of the major ways for reducing complexity. Finally, user evaluations showed that it is very hard to prove whether a visual programming language is in fact any better than any other string language.

Therefore the next step was to think of a way in which the VPL could be extended to include object-oriented features. In addition, the incorporation of object-oriented modelling aspects has been considered. This activity was supported by a survey of existing methodologies and notations (mainly OMT [RUMBAUGH-1991] and RUP/UML [QUATRANI-1998]). This was necessary to obtain information about the various notations and to get more confidence in the research carried out. At this time, the author looked for a technique to integrate VP (Visual Programming) with OT (Object Technology) as this was one important focus of the research. The result of this step led to two new ideas. It has been of great importance to discover how relationships and software architectures could be made more visible. The first idea was a top-down approach starting from the modelling and software specification standpoint. The proposal has been to make use of state diagrams and to integrate a VPL. This approach concentrated on the design and implementation of dialog-oriented programs. The second idea was a bottom-up approach aimed at inventing a general framework description language and applying the so-called slot idea in order to link visual code to the framework.

These new ideas were evaluated by a (constructed) case study based on some screen mock-ups. This research method allowed the study of various aspects in some depth and, most importantly, within a limited time scale. A simple adding machine was chosen as a candidate application. The top-down approach was shown to be quite useful though the limitation to dialog-oriented applications was not acceptable.
framework approach was judged to be more generic. However, it turned out that a universally applicable framework language is hard to achieve.

A subsequent step was to think of a way to emphasise the modelling aspects and to reduce the complexity for code implementation. As an application framework could be thought of as a system of pre-fabricated components that may be extended by other components, the idea was born to focus on component-based software development. The fourth and last approach was to evaluate the creation of a system that enables the visual modelling and implementation of object-oriented software components. CORBA [OMG-1995] was chosen as a candidate component technology due to its leading position in the market.

Once the core features of the proposed system had been identified, a prototype system was developed through a series of static and dynamic prototyping steps. The outcome was reported at the Symposium on Visual Languages 2000 [VL-2000]. The experimental style used for the prototype development led to a system, which was then evaluated by usability trials. One specific problem was the difficulty in finding subjects with the required skills. In addition, the usefulness of such a system is in general very difficult to show. Thus, it was decided to carry out informal user trials and to record the experiences made. In this respect, the research results reported in this document could form the basis for more in depth evaluations. However, the usefulness and effectiveness of combining Visual Modelling with Visual Programming for CORBA Component Development was shown by this approach.

1.3 Overview of the Thesis

Next, an overview of the thesis is given. The organisation of this document is as follows. Chapter 1 introduces the motivation for this research and gives a statement of the type of research methods that have been applied and the reasons why they have been chosen. Chapter 2 is an overview of different technologies of importance to this research and serves as background information to the reader. It is important to consider related research and to provide a link to it. This is the focus of chapter 3. Since the start of this research in 1995/6 several ideas have been developed by the author. In chapter 4, an overview of these different approaches is presented. Chapter 5 addresses in depth the final approach, which is called VOODE/VOOPL-1 for CORBA.
Usability problems have been investigated by usability trials. This is reported in chapter 6. Conclusions are drawn from the achieved results and described in chapter 7.

Important terms are described in Appendix A. Appendix B provides some abbreviations that are used in this document. Appendices C through G contain further information on VMB/VPL-C. VOOPL-1 processor sketches and a UML description file used by VOODE/VOOPL-1 are presented in appendices H and I. The ideal integration of the visual editor into Rose is addressed in appendix J. As a supplement to this thesis the following topics are described in appendix K through M: initial design of VOODE's user interface (3rd approach), activity diagram for the Rose Business Process Link, and Visual Age for C++. Appendix N contains the published work. Finally, appendices O through Q contain the observations and problems identified in usability trials as well as the Powerpoint slides which were used for the usability evaluations.
2 Background Information

2.1 Influence of Visual Technology

Since the wider availability of graphical workstations, the influence of visual technology has become more and more relevant. Two important related areas are visual user interfaces and integrated development environments.

As Ambler [AMBLER-1989] stated, Smalltalk first introduced not only a new language, extending the object-oriented approach of Simula67, but also a new and highly visual user interface. The user-interface makes intensive use of multiple windows. The idea was to provide large virtual windows with modeless switching between windows and therefore between functions.

Besides multiple windows, the concept of multiple views, first introduced by Pecan [REISS-1990], became more important. The use of multiple views provides a means to represent different aspects of commonly shared internal data.

Evaluations undertaken by Reiss and others concluded that users are limited by inherent one-dimensionality associated with the underlying textual languages [AMBLER-1989]. Reiss developed the Garden system which is able to accept descriptions of visual as well as textual programming objects. Garden uses a common internal representation model, i.e. an object-oriented environment complete with inheritance.

There is quite a lot of influence of visual technology on programming environments. One example concerns visual editing as a means of syntax-directed editing. Each of the visual editing systems which have been developed in the past are template-oriented to some degree. Ambler stated that the earliest approaches to visual programming consisted of visual editors for traditional imperative textual languages [AMBLER-1989].

Other examples are visual query languages (e.g. a prototype titled "VisualMelvyl" which represents a model of visual interfaces for an online public access catalogue). Besides diagrammatic visual query languages there are a number of systems which allow the use of non-diagrammatic queries, e.g. one system which was developed by Del Bimbo et al. in which the query is basically a simplified picture of the desired results [BIMBO-1992].
Further areas which are influenced by visual techniques are systems that use programming-by-demonstration, programming-by-rehearsal, and programming-with-example techniques. These systems make use of visual methods due to their natural applicability. It is considered easier for a system developer to perform a process by visual means than to describe textually how to perform the process. One example is ThinkPad, a declarative, graphical, programming-by-demonstration language and environment developed by Rubin et al. [RUBIN-1985], [MYERS-1986].

Furthermore, the visual paradigm also influences our culture [HORN-1998]. Horn believes that human-beings are going to develop a "visual language". By this he means that our language includes more than just text but images and symbols. Mattaini [MATTAINI-1993] argues that "throughout human history, people have used images for perceiving and communicating complex information holistically". He adds that due to the massively parallel processing capabilities the brain can absorb and integrate a tremendous amount of information simultaneously and, therefore, visual images play increasingly important roles in contemporary disciplines and professions that deal with complex data.

From the previous arguments it follows that visual technology has influenced the way that current development environments work. Visual techniques such as multiple windows and direct manipulation successfully used within GUI are now state-of-the-art and have directly influenced the work of this research.

### 2.2 Object Technology

#### 2.2.1 Introduction

When the author started the PhD work, the title of the proposed investigation was "Development of a Visual OO-Development Environment (VOODE) integrating an object modelling language which supports a visual object-oriented programming language based on the processor idea".

At this time, the often cited *method wars* between the competing different object-oriented methodologies and notations (such as OMT [RUMBAUGH-1991], [DERR-1995], OOSE [JACOBSON-1994], BOOCH [BOOCH-1995], Fusion [COLEMAN-1994], [MALAN-1995], VMT [TKACH-1996] and others) were in progress.
until the Unified Modeling Language (UML) [UML-1999], [UML-2000] was presented to
the OMG, and finally accepted and standardised in 1997. However, a modelling
language is not enough to develop complex systems. In order to control the iterative
and incremental life cycle of the software development process an extensive set of
guidelines is necessary that address both the technical and the organisational aspects.
In 1996, Rational launched the Rational Objectory Process which has now been
superseded by the Rational Unified Process [RATIONAL-1999].

Another trend has been the use of visual elements and techniques in
development environments. The term “visual” has also become one of the many
important marketing words in the OO community. It stands for “ease-of-use” though this
is not true in all cases. Sometimes it just means that the software is running under a
window-oriented graphical surface. Current object-oriented languages like C++ or Java,
to name the obvious, play an important role in current day program development
[NIELSEN-1997], [ROGERS-1997]. Compilers for these languages are integrated in a
more or less comfortable graphical environment.

As the applications which are built by these tools become increasingly complex,
new ways are required to enable the production of more robust object-oriented
programs, which have to be written in a shorter time. The question that remains is
“what technique (or combination of techniques) is necessary to make the inherent
structures of object-oriented applications more visible in order to reduce their
complexity?”. Another question is “to what extent can visual object-oriented
programming help in this?”. 

In the next two sections the basic concepts of object technology and its
important pitfalls are mentioned. These sections will help identify some of the problems
the research tries to solve.

2.2.2 Basic Concepts

There are several basic concepts that distinguish object-oriented programming from
procedural programming [B.MEYER-1988], [B.MEYER-1997]. One important aspect is
the idea of defining objects or classes that pass messages. A class may be seen as
being another name for an abstract data type (ADT) and an object as an instance of an
abstract data type. In object-oriented programming an action or method is initiated by a
message sent to a specific object.
The objects contain both data and methods that will manipulate or change the data. Since other objects do not have direct access to or knowledge of the data this important concept is called *encapsulation*.

In addition to message passing, object-oriented programming added more mechanisms to ADTs. One is called *generalisation/specialisation* which allows classes and objects to share the same code. Another is called *polymorphism* which allows the shared code to be dynamically executed according to the current object type and late binding rules. A complete list of object-oriented development concepts are described in the following table.

<table>
<thead>
<tr>
<th>Concept of Object-Oriented Development</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td>A software entity that responds to a set of messages.</td>
</tr>
<tr>
<td>Abstraction</td>
<td>Definition of the appropriate data and state for an object. &quot;Correct&quot; mapping of the real-world to objects in order to solve the problem.</td>
</tr>
<tr>
<td>Encapsulation</td>
<td>An object should hide its implementation. All it shows to other objects is its interface (i.e. public methods).</td>
</tr>
<tr>
<td>Instantiation</td>
<td>Creation of a specific instance of an object which contains the same &quot;behaviour&quot; but its own data.</td>
</tr>
<tr>
<td>Inheritance</td>
<td>A more general classification used to express common behaviour and data of objects, coined derived classes (subclasses) and base classes (superclasses). There are two types of inheritance: interface inheritance (i.e. inheriting the interface but not the implementation) and implementation inheritance (i.e. inheriting the interface and the implementation).</td>
</tr>
<tr>
<td>Specialisation</td>
<td>Each class needs to add its own data, interface and implementation. Thus, a user has to specialise each class.</td>
</tr>
<tr>
<td>Polymorphism</td>
<td>Different classes may respond to the same messages.</td>
</tr>
<tr>
<td>Type Checking and Message Binding</td>
<td>There are two principles. With &quot;static binding&quot; (or early, compile-time binding) the compiler decides which object is called. &quot;Dynamic binding&quot; (or late, run-time binding) allows the decision of which method to call to be made at run-time.</td>
</tr>
<tr>
<td>Composition</td>
<td>An instance of an object may be composed of other instances.</td>
</tr>
<tr>
<td>Containment</td>
<td>An object may hold a relationship to one or many other objects.</td>
</tr>
<tr>
<td>Association</td>
<td>A (navigable) relationship between objects so that an object has the capability to call any of the public methods of another object.</td>
</tr>
</tbody>
</table>

Table 1. Important concepts of object-oriented development.
There are basically three different programming paradigms used in object-oriented technology. These are called object-based, class-based and object-oriented programming [LEE-1997]. Object-based languages (like ADA [ADA-2000]) support the terminology of objects, and message passing. Class-based programming (like CLU [CLU-2000]) includes all the mechanisms of object-based languages plus the class mechanism. If an inheritance structure is also included then this language is called an object-oriented programming language. In addition, self-recursion is supported.

In addition to OOP, several OO notations (e.g. OMT [RUMBAUGH-1991], BOOCH [BOOCH-1994], UML [RATIONAL-1999], [LEE-1997], SADIE [PONT-1996] etc.) show the way from problem statements through modelling to complex application development/programming. UML has now become the quasi-standard. Many different academic and commercial CASE tools have been developed to apply these techniques. UML will be described in more detail in section 2.4.2.

2.2.3 Common Pitfalls of OO development

Many benefits are claimed for object-oriented development. These include, for instance, faster development, reuse of previous work, modular architecture, better management of complexity and a better mapping to the problem domain [WEBSTER-1995]. However, real-life projects teach that the development of object-oriented applications is still fairly complex and time-consuming. There are many reasons for this. For instance, applying the OO programming style is much more difficult than using the 'simple' procedural programming style as OOP requires more powerful and complex techniques. For example, permitted associations between different objects are not clear to many programmers. One basic problem is how an object can be accessed and which methods may be executed.

In addition, when CASE tools and code generators (or specific bridges) are used, these tools quite often generate many lines of code which are difficult to understand and maintain. While the modelling is done within the CASE tool the core programming is done within a different development environment. The reason is that round-trip engineering doesn't really work at present. Thus, there is a break in the development process as the integration and consistent traceability is still fairly weak. In addition, the systems that are built are not self-documenting and are difficult to maintain.
Webster [WEBSTER-1995] has described many pitfalls which may occur during object-oriented development. He has classified them into the following different categories: conceptual pitfalls (e.g. moving to object-oriented programming for the wrong reasons, thinking objects will solve all problems), political pitfalls (e.g. underestimating the resistance, overselling the technology), management pitfalls (e.g. not defining and using an effective methodology, assuming linear development), analysis and design pitfalls (e.g. underestimating the need for analysis and design, forcing a new paradigm on users), environment, language and tool pitfalls (e.g. deploying the wrong environment in-house, selecting the wrong programming language), class and object pitfalls (e.g. using inheritance badly, having the base class do too much or too little), coding pitfalls (e.g. testing objects for equality or identity, consuming memory inadvertently), quality assurance pitfalls (e.g. thinking about testing after the fact), and reuse pitfalls (e.g. not investing in reuse, allowing too many connections).

For this research, the coding and environment, language and tool pitfalls are of most interest. These will be described in more detail in chapter 4 when different visual approaches are outlined which are aimed at reducing these pitfalls.

2.3 Component Technology

2.3.1 Introduction

As modern applications become more sophisticated, their development increases in its complexity, and the time needed for its completion. Clearly new methods are required to facilitate a more rapid production of robust programs. 'Component Software' [SZYPERSKI-1998], [OVUM-1998] is an important and emerging area in the software field that addresses this problem. It describes the aim of application/system developers to create software systems that are built from individual components. Components are nontrivial software parts which conform to and provide a set of interfaces. They directly address the support of reusability/extensibility which failed in OO though there was a strong belief in this in the early 90s. Within component technology there are two general approaches named component-oriented development and component-based development. The latter one addresses the vision that a complete application may be assembled by just selecting and connecting existent components [NOD-2000].
There are several component technologies of current importance such as ActiveX [MS-1998], (D)COM [MS-2000], JavaBeans [SUN-1997], [ONEIL-1998], EJB [SUN-2000], or CORBA [OMG-1995]. In this PhD thesis, the focus is on the design and implementation of fine and large grained CORBA objects or components [MOWBRAY-1997b]. In this context, a component is a reusable software unit with an implementation and a set of separate interfaces which a client may use in order to request the services provided.

One important trend that enforces component-oriented development is client/server computing. This has created a deep paradigmatic shift in the industry since this resulted in the replacement of monolithic mainframe applications with applications split across client and server lines [ORFALI-1996]. A server-side object is a small-grained component that provides a public interface to other client objects. As CORBA from the OMG is an important and widely used technology, this research concentrates on CORBA, however the results of this research are not limited to this technology. The basics of CORBA in addition to common pitfalls are explained in more detail in the following sections.

2.3.2 CORBA

2.3.2.1 Introduction

One of the many issues that have arisen from the introduction of object-oriented programming techniques is the way distributed computing [DC-2000] is treated. In the past, client/server systems made use of different approaches such as RPC (DCE-RPC) or socket programming based upon TCP/IP. However, this caused many problems since it required low-level programming. Thus, more generic solutions have been investigated. One of the important technologies is CORBA which provides a language and platform independent way to connect different systems. It is therefore known as an enabling middle-ware technology. Since its inception in 1989 by the OMG (Object Management Group), CORBA has become the industry standard for the development of distributed object-oriented applications, as it promotes language and platform independent technologies. One particular strength of the CORBA technology and its architecture OMA (Object Model Architecture) [OMG-1995] [OPENGROUP-2000] is that it provides powerful mechanisms for legacy systems integration.
In summary, CORBA allows the interconnection of objects and applications, regardless of:

- the computer language of the applications that provide or use the objects
- the machine architecture of the computers involved
- the geographical location of the computer (connection through the Internet)

### 2.3.2.2 Basic Concepts

CORBA was designed to create an open *object infrastructure*. This is achieved by the definition of a global bus for distributed components based on the *Common Object Request Broker Architecture*. This *bus* or Object Request Broker (ORB) facilitates the interoperation of distributed objects by the specification of a platform and language independent interface language called *IDL* (Interface Definition Language). CORBA objects are packaged as binary components so that remote clients can be accessed via method invocations. The language and compiler used to create server objects are totally transparent to clients. The following list (see table 2a, 2b) names important CORBA concepts. Due to space limitations, it is not possible to treat the topic in full depth.

<table>
<thead>
<tr>
<th>CORBA Concept</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORBA IDL</td>
<td>To achieve language, site and platform independence, an <em>interface</em> to the objects is written in a (strongly typed) specification language called CORBA IDL or for short IDL. Though IDL allows multiple inheritance, unions, enumerations, constants, sequences, operations with input and output parameters, and compiler directives it is not a programming language.</td>
</tr>
<tr>
<td>IDL repository</td>
<td>Interface repository</td>
</tr>
<tr>
<td>Basic Object Adaptor (BOA)</td>
<td>The BOA is an interface of the ORB (and the BOAImpl class) implementation. It provides an interface to access remote objects. CORBA IDL to Java Language mapping specifies, for instance, a minimal Java API based on the BOA. This interface separates the BOA implementation from the ORB implementation while maintaining backward compatibility.</td>
</tr>
</tbody>
</table>

Table 2a. Important concepts of CORBA development (part1).
**Portable Object Adaptor (POA)**
A CORBA server may have one or more Portable Object Adapters arranged in the form of a tree. The POA implementation is used to dispatch the requests to their servants. It is intended to be more portable than BOA. The POA's specification introduces some new vocabulary and also changes the CORBA inheritance approach.

**Common Object Services (COS)**
CORBA Services (e.g. Naming Service, Event, Trader, Lifecycle and Property Services).

**CORBA Facilities**
CORBA Common Facilities specifications are services that many applications may share, but which are not as fundamental as the Common Object Services. To date, several specifications such as the Internationalisation and Time, and Mobile Agent Facility specifications are available.

**Objects By Value (OBV)**
A CORBA data type that combines aspects of interfaces, inheritance, methods, null-semantics with aspects of 'structs'. This is one of the requirements for RMI-IDL bridging.

**ORB**
Network software based on CORBA specification.

**Internet Inter-Operability Protocol (IIOP)**
Network protocol, specific protocol is RMI over IIOP for Java. Unlike RPC, this protocol invokes a method on a remote object.

**Tie mechanism**
This mechanism makes use of a delegation implementation class. This is transparent to the client application.

**Object reference**
Object references serve as proxy objects on the client side and encapsulate host name, port number, and object key.

**Object implementation**
This refers to the client-side object implementation.

**Interoperable Object Reference (IOR)**
This is a reference made from the object reference when a domain boundary is crossed. IORs may be stringified and be published to the naming service.

**Dynamic Invocation Interface (DII)**
Dynamic creation and invocation of requests.

**Dynamic Skeleton Interface (DSI)**
Delivering a request from ORB to an implementation without having skeletons.

**Dynamic Management of ANY's (DynAny)**
Code generated for the type has not been compiled with object implementation.

| Table 2b. Important concepts of CORBA development (part 2). |
Generally, CORBA components are developed using a set of tools, e.g. a CASE tool (such as Rational Rose [ROSE-1998]) and an IDE (such as Visual Café, JBuilder, Visual C++ etc.) plus an ORB product (such as Orbix [IONA-2000] or Visibroker [INPRISE-2000]).

The development of a CORBA component requires several steps. In the following, a brief and simplified outline is given. When a CORBA component is being developed, first of all, the business objects are identified, and modelled with UML (Unified Modeling Language) and a CASE tool. Then, the components and their interfaces are specified. The next step is to refine the interface model so that IDL (Interface Definition Language) files may be generated from the business model. The IDL files are then taken to create the stub and skeleton code and to map/transform the interface definitions to the target programming language (e.g. Java, C++). This job is done by the IDL compiler which is part of every ORB product. After this the implementation classes are created and refined. In order to keep the UML model consistent, the implementation classes are reverse engineered. Then, the required methods and attributes are designed and further specified, ideally using the CASE tool and an IDE. When this is done, the behaviour of the components can be implemented. Next, the code for the server mainline, which instantiates the CORBA objects, is written. Finally, the source code is compiled and the server executable is built and registered. After the successful completion of all these steps, the server code may be tested and client applications written.

### 2.3.2.3 Overview of Commercial Products

There are quite a few CORBA compliant object broker implementations on the market [ORFALI-1996]. For instance, there is SOM from IBM, Visibroker from Inprise, and Orbix/OrbixWeb from IONA. In addition to an ORB product, implementation tools are needed. The following list contains products which are used in current development.

- Microsoft Visual C++
- Symantec Visual Café
- Borland/Inprise J Builder
- Borland/Inprise C++ Builder
- IBM Visual Age for Java
In the following, JBuilder is taken as a candidate tool. The following feature list (see table 3a, 3b) shows the claims made by Inprise [JBUILDER-2000]:

<table>
<thead>
<tr>
<th>Developer Productivity</th>
</tr>
</thead>
<tbody>
<tr>
<td>100% Pure Java™ applications, applets, and JavaBeans® with no proprietary code or markers</td>
</tr>
<tr>
<td>Visual Java 2 two-way designers and wizards for drag-and-drop JFC/Swing application development</td>
</tr>
<tr>
<td>100% pure Java IDE hosted on Java 2 SDK 1.3</td>
</tr>
<tr>
<td>XML-based project manager with new JPX project file format</td>
</tr>
<tr>
<td>Advanced syntax highlighting for XML, WML, IDL, JSP™, and XSL, CSS style sheets</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Debugging (Additional Developer Productivity Features)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debugging of any JDK with Java 2 JPDA debug API support</td>
</tr>
<tr>
<td>ToolTip Expression Insight™ with detailed structure view of member instances</td>
</tr>
<tr>
<td>Multi-platform and Remote debugging for debugging complex distributed applications on a variety of platforms</td>
</tr>
<tr>
<td>Thread deadlock, stalls and race conditions detection on precise error location</td>
</tr>
<tr>
<td>Native JSP debugging with full breakpoint, watches, evaluation and context information support</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Integrated Team Development (Additional Debugging Features)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Revision browser for displaying history information of source versions</td>
</tr>
<tr>
<td>Visual source-level display of differences between source revisions</td>
</tr>
<tr>
<td>Conflict resolution for reconciling source versions between workspace and repository</td>
</tr>
<tr>
<td>Integrated Version Control System with support for update, merge, add, and check-out</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rapid Internet Development (Additional Team Development Features)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full support for Servlet 2.2/JSP 1.1 standard</td>
</tr>
<tr>
<td>Executes JSP and Servlet in the built-in Web-server</td>
</tr>
<tr>
<td>InternetBeans™ Express presentation components for rapidly creating Web driven Internet applications based on DataExpress™ database components</td>
</tr>
<tr>
<td>CodeInsight, ErrorInsight, ToolTip Evaluation for JavaServer Pages (JSP) embedded Java</td>
</tr>
<tr>
<td>Remote execution and debugging of Servlet</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Open Database Support (Additional Internet Development Features)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataExpress™ data access components with JDBC database connectivity, including support for Master-Detail relationships, Picklists, Lookups, Multiple Table Joins, and Transactions</td>
</tr>
<tr>
<td>Pure Java JDataStore™ 4 Development License for high performance data caching and compact persistence of data, objects, and arbitrary files</td>
</tr>
<tr>
<td>DataExpress Source Code for control and flexibility in building your Pure Java database applications</td>
</tr>
</tbody>
</table>

Table 3a. List of JBuilder 4 Features (part 1).
Rapid J2EE™ compliant development and deployment (Additional Database Features)

Visual creation of Enterprise JavaBeans® (EJBs), 100% compliant to the latest J2EE standard

Two-way editing of deployment descriptor for home and remote interface, container transactions, security roles and data sources

Entity Bean Modeler to create Container Managed Persistence (CMP) and Bean Managed Persistence (BMP) entity beans including home and remote interfaces as well as primary key classes

Dynamic Hot-deploy, to deploy/undeploy/redeploy EJBs to container without shutting down or restarting

Inprise™ Application Server™ 4.1 Development License, a complete EJB 1.1 implementation

Integrated WebLogic® Server 5.1 development support

Table 3b. List of JBuilder 4 Features (part 2).

Though this list is impressive, it is interesting to note how users feel about the tool. The following two reviews are presented to show different judgements of the same tool [JBUilder-2000].

Review JBuilder 3.0 by Mr. Kansler from The Netherlands, Apr 26, 2000. He says: "I'm currently an intern at Sun Microsystems. I work with JBuilder Enterprise for a program that I have to write. I admit that for the first two weeks I hated JBuilder. But once I got really going with it, I came to love it. The good structure of the IDE, the fast compiler and debugger, code completion and other fine enhancements to the Java 2 language make JBuilder 3 an excellent product."

Review JBuilder by Mr. Erlambi, Aug 10, 2000. He states: "I don't have a lot of experience with other Java IDE's, but I have used Visual Studio for development fairly extensively and JBuilder is simply not on-par. I find bugs regularly, it is slow, and has memory leaks (Yesterday it told me I was out of memory, and it was the only thing open on my machine with 256 MB RAM). On a brighter note, the two way visual development can be nice, wish it worked with arrays of GUI objects though."

What are the conclusions from this section? Firstly, it is important to note that many features are included in current IDEs for CORBA component development. The time needed to get used to the basic concepts is quite high (approximately 2-4 weeks). Secondly, the current tools contain many (useful) "helpers", named assistants, wizards, inspectors etc. However, none of the available commercial tools for CORBA
component development enable the usage of a VPL. This is somewhat surprising since visual programming (VP) as claimed by VL researchers can greatly enhance the strengths of proven techniques like object-oriented programming and modelling. This is especially true when the design model is very precisely defined so that the generated and refined code can be closely tied to the design model and round-trip engineering techniques can be used. The key idea of this baseline model is that "the system is the model" and "the design appears to be executable". This aspect is readdressed in section 2.4 when visual modelling is considered.

2.3.2.4 Common Pitfalls

There are many problems and pitfalls when trying to understand CORBA. First of all, since IDL is independent of the programming language an understanding of the mapping to the target language is required. Next come the problems in understanding the various CORBA concepts which take many weeks to master. In addition, there are many product-specific problems which hinder the application projects. One example is the language independent communication between client and server. Project experiences taught the author that a Java client built with ORB product A will not necessarily be able to talk to a C++ server written with ORB product B. Other problems are related to the IDL interfaces and the necessary generation steps. Unfortunately, an interface specification is not as stable during the software life-cycle as expected and thus many problems derive from unsynchronised project changes. One useful workaround is the use of XML strings as parameters but this causes the loss of type checking by the IDL compiler. An important question of this research is "how can CORBA development be facilitated by the use of visual techniques?".

Many problems are reported in newsgroups and discussion groups. These range from very simple questions (e.g. on how to write code fragments, how to catch CORBA exceptions, deployment aspects, CORBA types) to severe problems encountered in a specific environment.

As the reader may see from the following statement there are even more fundamental problems. "Unfortunately, current component technologies suffer from several problems. Over years of evolution and also during customisation for reuse, components are affected by changes arising from new functional and non-functional requirements, new software architectures and component deployment strategies, new versions of a computing environment that affect component communication, etc. Due to
those changes, components soon start suffering from complexity explosion, performance degradation and redundant implementations of similar features. The cumulative effect of this uncontrolled growth becomes prohibitive to reuse [NOD-2000].

One approach to solving this problem is the use of generative techniques. Using generative techniques, custom components can be produced from elements that can be considerably simpler than the executable components themselves, keeping the complexity of an evolving component system under control. Both generative and component-based approaches require changes in analysis and design methods, particularly, to move their focus from single systems to product lines [EISENECKER-2000].

The conclusion the author has drawn from the current approaches and tools is that the structures and relationships have to be made more visible and user tasks have to be facilitated. This is especially of importance for novice and inexperienced users. The research aim is guided by these assumptions and investigates the extent to which visual programming may help in this.

2.4 Visual Modelling

2.4.1 Introduction

During the 1990s different competing object-oriented methodologies and notations existed. This is often referred to as the “Method War”. The most popular methods were OMT by Rumbaugh, Booch method by Booch, and OOSE by Jacobson. These notations were “unified” to the so-called UML. “The Unified Modeling Language (UML) is a language for visualizing, specifying, constructing and documenting the artifacts of software systems. It is a general-purpose modelling language that can be used with all major object methods and applied to all application domains” [UML-2000].

2.4.2 Basic Concepts of UML

The first public draft (version 0.8) of UML was introduced in October 1995. The first commercially relevant version was 1.1 which was also presented to the OMG in September 1997 [QUATRANI-1998] and finally adopted in November 1997. At present,
the current UML version is 1.3. UML 1.4 is under way and is expected to become available in 2001.

The following table gives an overview of the important elements in UML 1.3/1.4.

<table>
<thead>
<tr>
<th>UML element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use Case</td>
<td>Use cases provide a means to capture system requirements and may be used to communicate with the end users and domain experts.</td>
</tr>
<tr>
<td>Use Case Diagrams</td>
<td>Use case diagrams present a high level view of how the system is used as viewed from an outsider's (actor's) perspective.</td>
</tr>
<tr>
<td>Actor</td>
<td>An actor is someone or something that interacts with or uses the system. An actor may provide input to and receive information from the system. The actor is external to the system and has no control over the use cases.</td>
</tr>
<tr>
<td>Scenario</td>
<td>A scenario describes one path through the flow of events for a use case.</td>
</tr>
<tr>
<td>Class</td>
<td>A class is an abstraction of real-world items and may be presented as a set of objects that share a common structure and common behaviour (i.e. the same attributes, operations, relationships and semantics).</td>
</tr>
<tr>
<td>Class Diagrams</td>
<td>A class diagram is used to describe possible systems or components made of classes.</td>
</tr>
<tr>
<td>Stereotype</td>
<td>A stereotype represents the sub-classification of a UML model element. Some stereotypes are already predefined, but new stereotypes may be added to represent new modelling types.</td>
</tr>
<tr>
<td>Packages</td>
<td>A package is a general-purpose model element that organises model elements into groups.</td>
</tr>
<tr>
<td>Sequence Diagrams</td>
<td>These diagram types provide a way to look at a scenario in a time-based order.</td>
</tr>
<tr>
<td>Associations</td>
<td>There are several possible relations, which may be modelled (e.g. bi-directional association, aggregation, reflexive, package, and inheritance relationships).</td>
</tr>
<tr>
<td>Activity Diagrams</td>
<td>Activity diagrams provide a way to model the workflow of a business process. They may also be useful to model object flows.</td>
</tr>
<tr>
<td>State Diagrams</td>
<td>State diagrams are important for describing the dynamics of an object.</td>
</tr>
</tbody>
</table>

Table 4. List of important UML elements.

UML is also a helpful means for designing the System Architecture. "Establishing a sound architectural foundation is absolutely essential to the success of an object-oriented project" [BOOCH-1995]. "The architecture of a proposed system does not appear in a flash. It takes exploration of the use cases, a proof-of-concept
prototype, an architectural baseline, and other efforts during the Inception and Elaboration phases" [JACOBSON-1999]. The Unified Process as described in [JACOBSON-1999] goes beyond mere object-oriented analysis and design. It defines the complete software development life cycle as a component-based process that is use-case driven, architecture-centric, iterative, and incremental.

Within the proprietary RUP specification of Rational that is based on the Unified Process, the following views are defined [QUATRANI-1998]:

- **Logical View** which addresses the functional requirements of the system.
- **Component View** which concerns itself with the actual module organisation.
- **Process View** which focuses on the run-time implementation structure of the system.
- **Deployment View** which involves the mapping of software to processing nodes.
- **Use Case View** which validates the logical, process, component and deployment views.

The different dimensions of the concurrent views are depicted in the following figure.

```
Logical View
Functionality

Component View
Software Management, Reuse, Portability

Use Case View
Understandability, Usability

Process View
Performance, Availability, Fault Tolerance

Deployment View
Performance, Availability, Fault Tolerance, Scalability, Delivery and Installation
```

**Figure 1.** The "4 + 1" view of architecture [QUATRANI-1998], [KRUCHTEN-2000].

One important question is why UML and visual modelling is considered to be such a big advantage. An answer to this was given by Edward Tufte [TUFTE-1983]. He
noted in "The Visual Display of Quantitative Information" that "graphics reveal data". By this statement he means that certain complex sets of data convey far more information than raw data itself. The Unified Modeling Language (UML) may be seen as a means to visualise a software system by different models. As Terry Quatrani in [QUATRANI-1998] stated "visual modelling is a way of thinking about problems using models organized around real-world ideas". The Convergence of the modelling languages led to the use of common tools and methods. As Booch stated in [QUATRANI-1998] "the industry-wide standards offered the promise of achieving the true interoperability and reuse of software long sought". Furthermore, "UML was designed with extensibility in mind, so it can adapt to new issues as they arise" [JACOBSON-2000].

In addition to a notation, a software development process is needed. Besides the V-Model [BUND-2000] and the Catalysis approach [DSOUZA-1998], [DSOUZA-2000], the Rational Unified process is the most prominent. It is an iterative/incremental, architecture-centric, and requirements-driven process. The following figure provides an overview of the main phases and workflows.

**Figure 2. Overview of the Rational Unified Process**

[RATIONAL-2000, © Rational Corp.].
There are many different CASE tools available on the market. Important products are Rational Rose [RATIONAL-1999], Together/J [TSOFT-2000], Popkin's System Architect [POPKIN-2000], Innovator [MID-2000], and StP [AONIX-2000]. These tools support most parts of the current UML version.

The following table summarises some important features of current CASE tools:

<table>
<thead>
<tr>
<th>Product feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UML 1.3 features</td>
<td>Most CASE tools claim to support UML 1.3. However, this doesn't mean that all UML elements are really adequately supported. One example is OCL.</td>
</tr>
<tr>
<td>Support for component-oriented development</td>
<td>Most tools offer specific CORBA and EJB code generators and bridges offered by technology partners.</td>
</tr>
<tr>
<td>Design Patterns</td>
<td>Selection of available design patterns (e.g. from Gamma [GAMMA-1995]) and associating them to existing UML classes.</td>
</tr>
<tr>
<td>Data Modelling (object-relational mapping)</td>
<td>Support for logical data models and mapping mechanism to relational data bases.</td>
</tr>
<tr>
<td>Round-trip Engineering</td>
<td>Most CASE tool vendors claim to support round-trip engineering for all major programming languages. However, many specific language features may not be represented in UML.</td>
</tr>
</tbody>
</table>

Table 5. List of important CASE tool features.

At present there is a movement away from CASE tools towards I-CASE tools as there was in the 80s for procedural CASE tools. This means that many (commercial) requirements result in new features such as data modelling, web modelling and business process modelling.

2.4.3 Current Trends and Possible Improvements

Observations from practice show that not all UML elements and diagrams are used. Many projects need a mixed or hybrid approach since the corporate data is stored in relational databases which have to be accessed by COBOL or PL/1 applications. This also results in concurrent top-down/bottom-up approaches and UML models which do not reflect the implementation. In addition, Round-Trip-Engineering does not work for all types of projects and languages.

Other concerns are that UML is too informal and works only during the analysis phase and for documentation issues. Kent Beck et al. are working on special constraint
diagrams called Spider diagrams to improve this situation [KENT-2000], [VL-2000].
Finally, UML (and UML/RT) was not invented as a Visual Programming Language
[UML-2000]. Thus, UML models are not "Turing computational", i.e. is not powerful
enough to specify an executable program.

There are several techniques for improving object-/component-oriented
development. One category may be called the generator approach. This approach tries
to include powerful code generation facilities within a development tool. An example of
this category are EJB bridges (e.g. ArcStyler [ARCSTYLER-2000] or Inline [INLINE-
2000]). Another candidate of this category is BITPlan's generic approach [BITPLAN-
2000] which uses a UML analysis model containing the business logic as a basis. A
Rational Rose Addin called SmartGenerator is then used to create and maintain the
code framework.

Another important approach is the so-called single source technology. The
basic idea is that the UML model and the code entered are kept in sync all of the time.
This means that if a user enters a new method or adds a new attribute the
 corresponding changes are immediately updated in the UML model. One example of
this category is TogetherSoft's Together/J. This technology is also "simulated" by an
XMI bridge for Rational Rose and Visual Age for Java.

Another trend is visual programming (VP) used for the development of
JavaBeans components, ActiveX controls, and with query by example (QBE)
approaches (e.g. MS Access). Many visual programming languages have been
developed up to now but not many have reached a mature state and become a
commercially available tool. The reasons for this are mainly that powerful workstations
have been very expensive in the past and visual interaction and programming has only
just recently received more attention. The author started this research with the
assumption that combining the strengths of proven techniques (such as object-oriented
programming) and visual programming will yield a significant improvement in program
design and development. Object technology (OT) has emerged to such a degree that it
now plays an important role in program development. Some OO promoters even call it
the most important evolution (revolution) of the 1990s [LEE-1997]. This may also be
true for Component-Based Software Engineering. Thus it is worth investigating the
implications for visual object-oriented programming and what benefits they could bring
to present day development problems. In the next section, background information on visual languages is presented.

2.5 Visual Languages

2.5.1 Overview of Visual Languages

As Chang [CHANG-1987] and many others stated, the term *visual language* means different things to different people. One meaning is that it is a language for processing visual information whereby the objects handled by the language are visual. Another meaning is that the language itself is visual, i.e. visual expressions are used to define operations. In short, the first type is called a *visual information processing language*, the second type is called a *visual programming language*.

Further, Chang distinguishes four different types of visual languages which all deal with generalised icons (i.e. object or process icons). An object icon consists of two parts, namely the logical part (the meaning) and the physical part (the image). A process icon represents an action, or a computational process. The distinction between an object icon and a process icon depends both on context and interpretation [CHANG-1987].

Shu [SHU-1986] classifies visual programming languages according to the following three dimensions: 1. *visibility* (adequacy in visualisation), 2. *language level* (adequacy in representing processes by procedural or nonprocedural means), and 3. *language scope* (adequacy in representing objects for different applications). Thereby, the level of a language is inversely related to the amount of detail a user has to specify in order to achieve a task.

There are different degrees of abstraction used to represent the intended computation. For instance, state transition diagrams are used as a visual programming language to design and specify user interfaces. Other applications make use of graphical (iconic) representations, such as dataflow diagrams [HILS-1992], HIPO charts, action diagrams, or Nassi-Shneiderman diagrams [CHANG-1987].

In most cases, visual languages are limited to special purposes. A further distinction is made to identify iconic languages. Chang gives an informal definition, as follows: "Iconic languages are visual languages that use icons extensively or
exclusively." Further, he adds that an iconic language could be any of the following four visual language types: 1. a language that supports visual interaction, 2. a visual programming language, 3. a visual information processing language, or 4. an iconic visual information processing language. An iconic sentence has definite syntactic rules governing its construction and definite semantic rules governing its representation. Finally, an iconic language is composed of these iconic sentences.

The following figure gives an example of iconic sentences.

![Diagram of iconic sentences](image)

**Figure 3.** Construction of iconic sentences [CHANG-1987].

### 2.5.2 Overview of Visual Programming Languages

"There is an increasing expectation that visualisations in various domains will help communicate details of complex processes, structures and flows. In the field of computing, visual programming languages are expected to provide these features in an easy to understand way" [OBERLANDER-1999].

Visual programming languages are not primarily designed for use by expert programmers [BLACKWELL-1998]. Graphical representations are expected to be more appropriate for inexperienced computer users. This has been demonstrated in studies by Cunniff & Taylor [CUNNIFF-1987], as well as being an explicit claim when programming languages are promoted for use by novices without any empirical verification [BONAR-1987]. Nardi on the other hand, has strongly criticised the notion of any programming language being suitable for novices or "end-users" [NARDI-1993].

The development of VPLs can be traced back to research carried out by Sutherland [SUTHERLAND-1963] and Smith [SMITH-1977]. Specific diagrammatic notations for programming have been first described by Fitter and Green...
However, as Alan Kay in [SHU-1988] remarked, "there is no consensus on what visual programming is". Different authors describe different classification schemes. Kay suggests the categories which are depicted in the following table.

<table>
<thead>
<tr>
<th>Visual Environment</th>
<th>Visualisation of ...</th>
<th>Visual coaching</th>
</tr>
</thead>
<tbody>
<tr>
<td>data or information about data</td>
<td>program and/or execution</td>
<td>software design</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Visual Languages</th>
<th>Visual Programming Languages ...</th>
<th>for handling visual information</th>
<th>for supporting visual interactions</th>
</tr>
</thead>
<tbody>
<tr>
<td>diagrammatic systems</td>
<td>iconic systems</td>
<td>form systems</td>
<td></td>
</tr>
</tbody>
</table>

Table 6. Visual Programming (cf. [SHU-1988]).

Though the term "visual" is widely used, it is important to describe exactly how Visual Programming (VP) differs from traditional textual programming [BURNETT-2000]. A language is called a visual language if it handles visual information or supports visual interactions. Chang [CHANG-1987] calls a language visual when objects handled by the language are visual, the language processes visual information, or if the language itself is visual. The language is called a visual programming language if it is actually used for programming with visual expressions (e.g. iconic sentences). Different classification schemes for VP have been published, for example, Chang [CHANG-1987], Myers [MYERS-1986], and Shu [SHU-1986], [SHU-1988]. It is important to notice that the definitions contained in those works differ significantly. A good classification of visual object-oriented languages is presented in [BURNETT-1995]. There follows a more verbal description of what "visual programming" means.
A system may be called a visual programming system if:

- it covers important aspects of programming (i.e. syntax and semantics).
- it makes use of graphical techniques (for instance, the program design is simplified through the use of images and pictorial representations).
- ideally, the visual representation is not just a picture of the logical program structure but of the executable program itself (one example is VisaVis [POSWIG-1996]).
- the user interaction contains both textual and graphical elements.
- the aim of a system developer is to make use of non-verbal human abilities (i.e. right hemisphere tasks). Non-verbal and visio-spatial abilities are primarily located in the right hemisphere of the brain. Language and logical abilities are concentrated in the left hemisphere.

It is important to mention that the VL community has not been particularly interested in explaining why a VPL should be better than a string-based language. However, there is some interest coming from the cognitive psychology community [GREEN-2000]. Green's work classifies programming languages as information structures and emphasises that the structure of the notation matches the structure of the task and that it is less important whether text or diagrams are used. Cognitive dimensions provide a broad-brush analysis of programming notations and environments, and have been applied to both textual and graphical notations [GREEN-1989], [GOOD-2000b].

Another important point is that there are not as many visual languages defined as there are in the "classical" programming area. In the 70s and 80s a lot of different text languages were invented. The author also created two string languages called FHTROB2 [BUEHLER-1989] and PPL (Portable Programming Language)

Information exists on about 2350 past and present computer languages up until 1995 [KINNERSLEY-1995]. Konrad Zuse [ZUSE-2000] may have developed the first real computer programming language, called "Plankalkuel" (ca. 1945). According to Sammet [KINNERSLEY-1995], [SAMMET-2000], over 200 programming languages were developed between 1952 and 1972, but she considered only about 13 of them to be significant.

¹ Not published work.
These most influential languages are mentioned in the following table.

<table>
<thead>
<tr>
<th>Year</th>
<th>Language</th>
<th>Year</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>1957</td>
<td>FORTRAN</td>
<td>1978</td>
<td>CSP</td>
</tr>
<tr>
<td>1958</td>
<td>ALGOL</td>
<td>1978</td>
<td>FP</td>
</tr>
<tr>
<td>1960</td>
<td>LISP</td>
<td>1980</td>
<td>dBASE II</td>
</tr>
<tr>
<td>1960</td>
<td>COBOL</td>
<td>1983</td>
<td>Smalltalk-80</td>
</tr>
<tr>
<td>1962</td>
<td>APL</td>
<td>1983</td>
<td>Ada</td>
</tr>
<tr>
<td>1962</td>
<td>SIMULA</td>
<td>1983</td>
<td>Parlog</td>
</tr>
<tr>
<td>1964</td>
<td>BASIC</td>
<td>1984</td>
<td>Standard ML</td>
</tr>
<tr>
<td>1964</td>
<td>PL/I</td>
<td>1986</td>
<td>C++</td>
</tr>
<tr>
<td>1966</td>
<td>ISWIM</td>
<td>1986</td>
<td>CLP(R)</td>
</tr>
<tr>
<td>1970</td>
<td>Prolog</td>
<td>1986</td>
<td>Eiffel</td>
</tr>
<tr>
<td>1972</td>
<td>C</td>
<td>1988</td>
<td>CLOS</td>
</tr>
<tr>
<td>1975</td>
<td>Pascal</td>
<td>1988</td>
<td>Mathematica</td>
</tr>
<tr>
<td>1975</td>
<td>Scheme</td>
<td>1988</td>
<td>Oberon</td>
</tr>
<tr>
<td>1977</td>
<td>OPS5</td>
<td>1990</td>
<td>Haskell</td>
</tr>
</tbody>
</table>

**Table 7.** A chronology of some influential computer languages [SAMMET-2000].

---

### Programming

- Object-oriented programming, (main language: Java)
- Functional programming (main language: Haskell)
- Logic programming (main language: Prolog)
- Procedural/Imperative programming (main language: C)

#### Programming languages

- Semantics; Compilers; Interpreters

### Mathematical Specification

The specification of software systems (including computer programs) using rigorous methods. Logic and set theory provide the basis. Other calculi are used according to the type of specification (e.g. lambda calculus for functional programming and denotational semantics; CCS for concurrent specifications).

### Hardware and Robotic Systems

- Artificial Intelligence Approaches

### Human Computer Interface Design

- Multimedia; Graphics; Internet

### Systems Programming

- Operating systems; Embedded systems

### Networks and Protocols

- Wide area and local

**Table 8.** Main areas in the Department of Computer Science at DMU [DMU-2000].

Finally, visual languages aren't relevant in the academic curriculum. As an example the Department of Computer Science [DMU-2000] which is part of the Faculty of Computing Sciences and Engineering at DMU (Leicester, UK) is taken. It provides substantial input to the Computer Science, Software Engineering and Multimedia Computing undergraduate Diets and to the Masters courses in Information Technology and Human Computer Systems. Specific research areas include Computational...
Intelligence, Computer Imaging and Real Time Safety Critical Systems. The main areas are mentioned in table 8. In the author's opinion the reason why VP has not entered the curriculum is mainly due to the long tradition of programming lectures and their focus on teaching algorithms, data structures and other related issues which would be less relevant for a VPL.

2.5.3 Graphical Notations for Program Design

At the end of the 1980s [CHATTRATICHART-2000] the term “graphical programming” disappeared from the literature and evolved into “visual programming”. Until this time graphical programming meant flowchart representations, including Nassi Shneiderman and other diagram types. This kind of programming also included the visual specification of programs.

Nowadays, the term visual programming is widely accepted and used. There are four paradigms/approaches that may be distinguished [VL-1997]:

1. Controlflow
   A visual representation is used to express typical control flow constructs found in conventional programming languages. This may also include parallel or non-deterministic control flow. As mentioned in [VL-1997, p 13], considerable training is required to learn the constructs. Examples of this type are Create, CWave, SeeDo, and VIPR.

2. Dataflow
   The control of dataflow languages is mainly driven by the computation of data. The visual system determines when data is ready to be computed. Also, external events are represented as external controls. Examples are Prograph, and “Show and Tell”.

3. Equation-form-based
   Visual languages following the equation-form-based approach are very similar to dataflow languages. The main difference is that these kind of language considers data as a single discrete value and not as a continuous stream. Examples are Formulate, and Forms/3.

4. Rule-based
   This type of visual language is based on the idea of states and a set of rules that transform one state to some other state. Each rule has a precondition to decide
which transformation is to be carried out. If several rules are applicable at any time then the general policy is that it doesn't matter which transformation is taken. Examples are Altaira, Cocoa, RoadSurf, and AgentSheets.

However, little research was carried out to decide whether one paradigm was better than another [VL-1997, p 18]. Thus, this research could not be based on a proven paradigm. However, the author is convinced that most programmers are more acquainted with control flow languages than other types because of their dominant use in education and professional settings.

Within the design of software, different levels (e.g. system and program design) can be distinguished. For each level special notations are necessary. In some of the early research, the usefulness of graphical notations was investigated. A good survey of early graphical notations for program design was given by L. Tripp [TRIPP-1988]. In the following, some important aspects of graphical notations are presented.

Tripp distinguishes three major categories, viz. (1) box and line notations, (2) box notations, and (3) line notations. A box notation makes use of rectangles or some other enclosed figures whereby a textual description is contained in the box in order to represent the intended functionality. Within a line notation, no boxes are used and the program scheme consists only of lines and textual labels. A box and line notation is a combination of both schemes. Here are examples for the different notations (see table 9):

<table>
<thead>
<tr>
<th>Type of notation</th>
<th>Name</th>
<th>Author/Contributor</th>
</tr>
</thead>
<tbody>
<tr>
<td>line notation</td>
<td>Dimensional Flowchart</td>
<td>Witty (1977)</td>
</tr>
<tr>
<td>box notation</td>
<td>Flowblocks</td>
<td>Grouse (1977)</td>
</tr>
<tr>
<td></td>
<td>Box Chart</td>
<td>Jonssson (1987)</td>
</tr>
<tr>
<td></td>
<td>Lindsey Chart</td>
<td>Lindsey (1977)</td>
</tr>
<tr>
<td></td>
<td>SPDM Diagram</td>
<td>Marca (1979)</td>
</tr>
<tr>
<td></td>
<td>FP Diagrams</td>
<td>Pagan (1987)</td>
</tr>
<tr>
<td>box and line notation</td>
<td>Rothon Diagrams</td>
<td>Brown (1983)</td>
</tr>
<tr>
<td></td>
<td>Ferstl Chart</td>
<td>Ferstl (1987)</td>
</tr>
<tr>
<td></td>
<td>PAD</td>
<td>Futumura et al. (1981)</td>
</tr>
<tr>
<td></td>
<td>Compact Chart</td>
<td>Hanata et al. (1980)</td>
</tr>
<tr>
<td></td>
<td>SSD Diagram</td>
<td>Kanada et al. (1980)</td>
</tr>
<tr>
<td></td>
<td>Structure Chart</td>
<td>Chyou (1984)</td>
</tr>
<tr>
<td></td>
<td>Dorsan Chart</td>
<td>Dorsan et al. (1972)</td>
</tr>
<tr>
<td></td>
<td>Schematic Logic</td>
<td>Jensen et al. (1979)</td>
</tr>
</tbody>
</table>

Table 9. Survey of graphical program notations.
The following figure gives an impression of the different program schemes.

Figure 4. Different graphical program notations.

As Tripp [TRIPP-1988] stated, all notations are very similar as regards their capabilities, and are strongly influenced by the basic control structures of structured programming. Differences occur in the following aspects:

- graphic formulation
- presence/absence of parallel control structures
- design for a special language, e.g. Pagan (for a functional programming language)

Tripp also claimed that a trend towards visual programming (VP) could be observed. Due to the development of new display technologies and more powerful computers (mainly cheap PCs and workstations) VP has become more important and new interesting systems have been built. However, box and/or line notations in particular are no longer currently acceptable. Thus, other visual abstractions are necessary. In the following, higher levels of visualisations are described.

A visual software system which overcomes this problem is Vista, a visual multi-paradigm language integrated within a comfortable development environment. Vista supports "fundamental software engineering principles" during programming, such as adequate notation, modularisation, and weak coupling. Further, it provides capabilities for the construction of event-driven and data-transformation systems [BURNETT-1995]. It makes use of processors as shown in the following figure.
In the author's opinion, Vista is a very good example of how visual programming can be used in a very effective way to help construct applications on a very abstract level. A more detailed description of Vista is given in section 3.3.2.

Apart from the positive aspects of visual techniques in general and VPL in particular there are also some limitations mentioned in the literature. One difficulty derives from the limited screen space and is especially relevant for complex systems. The question is, how many visual primitives/icons can be visible on the screen at the same time. Another problem is related to the possible ambiguity of various icons and their intended meaning. However, this may be solved in various ways (e.g. as in Windows applications through the use of context-dependent text labels appearing after a certain time, or well-designed status lines which give a qualified and context-sensitive feedback). In general, an icon should be designed in such a way that the text indicates the semantics of the icon. The space problem is also solvable, e.g. by the introduction of 3D control elements or more advanced navigation techniques (e.g. fisheye views).
2.5.4 Visual programming (VP) requirements and problems

In accordance with the previous definition visual programming (VP) may be characterised as follows:

- VP is the use of graphical techniques.

- It takes into account the need for programmers to communicate with computers using both graphics and text, and for computers to communicate with programmers.

- Attempt to exploit humans' non-verbal capabilities.

- It can be applied to "all" aspects of programming (at least in theory).

It is interesting to note, that the trend in using icons (or pictures) is contrary to the history of human beings' writing. Two thousand years ago, Egyptian traders used visual describing mechanisms to write an invoice. Nowadays, our invoices are text-based with some visual elements (e.g. tables). The main reason for this is that the space needed to write an invoice is reduced. Another reason it that text has proven to be easier to understand by the user. Due to resource limitations programming started on a very low level but is now moving to a more complex and abstract level. Since the availability of cheaper graphical workstations visual elements are becoming more and more important. Therefore, one can argue that the reason why visual programming techniques haven't been greatly employed in current development environments is due to computing and technology restrictions rather than the inappropriateness of the concepts and technology itself.

Most visual systems are based on icons. In order to be successful, they should be based on an easy-to-use metaphor. The "hard bit" is to find icons that are derived from the user's world and experiences. This difficulty is apparent when the various different desktop icons are considered.

Only a few of them are standardised and commonly accepted (one example is certainly the "Open File" icon). In earlier days, IBM tried to solve some of the user interaction problems by defining standard elements and limiting the variety of the surface of an application within its SAA strategy [SAA-1991]. Despite today's Windows GUI Style Guides the look and feel of Windows applications is not really converging.
In the programming world, it is very difficult to find icons of higher granularity which are commonly understandable and which may be used more effectively. It is important to realise that it is not sufficient just to turn textual elements into icons as this would result in an exploitation of icons. The variations of textual elements would have to be translated into a set of different icons. For example, the visual representation of "if <cond> then <a> else <b>" could result in many different icons depending on the translation technique. Another possibility is the use of visual adornments. However, this may lead to code ambiguity.

Another problem is related to the layout of the different icons. If icons are defined and are used in a flexible way then the problem of crossing lines arises. In addition, the reference to these icons becomes more and more complex. It becomes clear that advanced techniques are necessary to avoid these problems. Further, most visual systems do not possess the best performance.

The composition of different icons (i.e. combining them into one logical unit) has the shortcoming that the meaning may become ambiguous to the user. One way to avoid this problem is to make use of "property sheets". In this way icons with similar behaviour may be unified and the differences may only be evident through different attributes. The idea of this research is therefore to find (higher-level) pictorial representations based on patterns (i.e. textual programming code) and to mix textual and visual elements together.

Visual systems are in general not effective at representing the overall application structure. One of the many problems is to find techniques that follows the 7±2-rule [MILLER-1956]. This means that it is possible for a normal user to identify 7 ± 2 items at once very easily. In general, a visual system should be built in such a way that the cognitive load is reduced. These are some of the many psychological aspects with respect to VP.

2.5.5 Visual Object-Oriented Programming

"Visual object-oriented programming (VOOP) refers either to a VPL that supports the object-oriented paradigm, or the use of a visual environment for a textual object-oriented language" [BURNETT-1995]. Commercial examples of the latter are for example Visual Studio for C++, JBuilder, VisualCafé, Visual Age for Java, KAWA, and Forté. In this thesis however, the interest lies on VOOPL within a visual environment.
This is quite often termed a “completely visual object-oriented language” [CITRIN-1995, pp 67] in [BURNETT-1995].

Examples of academic systems are:

- VIPR – a completely visual approach that tries to explicitly represent the dynamic behaviour of object-oriented programs and that avoids the yo-yo problem [BUDD-1997].
- HI-VISUAL [HIRAKAWA-1990] - though not explicitly classified as a VOOPL, HI-Visual has some interesting object-based features.

The term *visual object-oriented programming* is used within two distinct contexts, either as the use of an object-oriented programming language in a window-oriented graphical environment, or as the use of an environment with graphical tools. The following figure shows the relationship to the object-oriented programming paradigm.

![Classification of visual object-oriented programming](image)

*Figure 6. Classification of visual object-oriented programming [BURNETT-1995, p. 5].*

Rephrasing the previous definition, *object-oriented programming* (OOP) is the act of modelling systems in terms of objects. The basic concepts of object-oriented programming are data abstraction, instantiation, composition, and specialisation [BOOCH-1994]. As Rumbaugh and others claim, "object-oriented modeling and design promote better understanding of requirements, cleaner designs, and more maintainable systems" [RUMBAUGH-1991].
Visual programming is the use of graphical techniques and takes into account the need for programmers to communicate with computers using both graphics and text, and for computers to communicate with programmers. Unlike "conventional" string-based programming (i.e. text-based programming) techniques, visual programming represents an "attempt to exploit our non-verbal capabilities". Furthermore, it can be applied to "all aspects of programming" [SHU-1988 p. 1 and p. 9].

Visual object-oriented programming tries to combine OOP with visual techniques. This PhD work concentrates on the inclusion of a visual syntax for component-oriented programming. This may be done by mapping object-oriented concepts to visual elements which may strongly rely on a textual object-oriented language.

What is the motivation to combine OOT with VPL? Basically, there are two reasons (cf. [BURNETT-1995, p171]): OO/CT promotes support of reusability and extensibility, VPL brings the "accessibility" (i.e. directness) of VP.

The idea of this research is to seek a method based on VP which may be integrated within a known and effective OO methodology. The author decided to take CASE technology and UML as a starting point and to work out how VP could be integrated within it. With the advent of (distributed) component technology, CORBA was judged to be a further baseline for the proposed work. The following figure shows the basic architecture of the intended system.

**Figure 7.** Basic "architecture" of a Visual Object-Oriented Development Environment (VOODE).
The scope of the research should not be too limited to enable investigation into an open and flexible method. In contrast to other VP systems, the proposed system should not be limited to narrow domains. Several basic ideas to achieve this were identified and will be described in later sections. In the next section, an approach for the definition of VPLs is introduced.

### 2.5.6 The definition of a VPL

Generally speaking, VPLs may be based upon various methodologies. In the literature, various grammars are described in order to define a VPL. The most important models are "Positional Grammars" (for short PGs) [COSTAGLIOLA-1997], "Picture Layout Grammars" (PLGs) [GOLIN-1991a], [GOLIN-1991b], "Relation Grammars" (RGs) [CRIMI-1991], and "Constraint Multiset Grammars" (CMGs) [MARRIOTT-1994]. Depending on the complexity, an effective definition for an intended VPL could be based on a PG as described in [COSTAGLIOLA-1997].

A context-free positional grammar is a six-tuple and defined as

\[
PG = (N, T, S, P, POS, PE)
\]

where

- **N** is a finite nonempty set of nonterminals
- **T** is a finite nonempty set of terminals, with \(N \cap T = \emptyset\)
- **S** denotes the starting nonterminal
- **P** is a finite nonempty set of productions
- **POS** is a finite set of binary relation identifiers
- **PE** is a pictorial evaluator

**Grammar objects** are either nonterminals or terminals. They are characterised by an image (so-called "n attaching-point entities", NAPEs for short) and a set of attributes, named syntactic attributes. There may be one or more relations \(R\) between grammar objects.
Each production in P has the form
\
A \rightarrow x_1 \ R_1 \ x_2 \ ... \ R_{m-1} \ x_m \ \Delta
\

where

A denotes a nonterminal
x_i denotes a grammar object
R_j represents a sequence of the form \(< REL_1^{h_1}, REL_2^{h_2}, ..., h_n >\) with n \geq 1
\Delta is a rule which synthesises the syntactic attribute values of A from those of x_i

A subset of the definitions for a hypothetical VPL is given below.

PG_{VPL_{Test}} = (N, T, S, P, POS, PE)

where

N = \{ body, if_stmt, print_stmt, slot_code, stmt\}
T = \{Start, End, Connector, If (Cond.), Print\}
S = slot_code
POS = \{JOINT\}

with JOINT denotes a relation of the type JOINT(h, k)
and is defined as follows:

Given two NAPEs denoted by x and y, the relation
"x JOINT(h,k) y" holds
iff the attaching point h of x is connected to the attaching
point k of y.\(^2\)

\(^2\) iff stands for "if and only if".
The terminal NAPEs are visually described below.

**Figure 8.** The terminal NAPEs Start, End, If (Cond.), Connector, Print.

---

3 A more general definition is [COSTAGLIOLA-1997]: "Given the production $A \rightarrow x_1 R_1 x_2 \ldots R_m x_m A$, if $\text{JOINT}(h, k)$ occurs in $R_n$, then the attaching point $h$ of $x_{n+1}$ is connected to the attaching point $k$ of $x_{n+1}$."
Finally, a sample grammar production is shown graphically.

![Diagram showing a sample grammar production](image)

**Figure 9.** One sample grammar production.

Given the formal definition of PGs, it is possible to build a parser based upon a given grammar.

In [COMPSAC-1998], the author described another method of how textual languages may be translated into an equivalent visual representation by the means of graphical techniques (and the Visual Meta Builder tool). This will be described in section 4.1. In sections 2.5.7 and 2.5.8, two systems that help construct visual systems are outlined.

### 2.5.7 Vampire: A Visual Metatool for Programming Iconic Environments

One obstacle to the design of new visual programming languages is the effort required to design and implement the language "under consideration". As this experimental approach doesn't always result in a usable system, it is very beneficial to make use of a rapid prototyping system, such as Vampire [MCINTYRE-1995], which "allows language designers to explore new concepts efficiently".
**Vampire** is an acronym and stands for Visual Metatools for Programming Iconic Environments. The system was inspired by a graphical reasoning tool (viz. Furnas' BITPICT system [FURNAS-1990], [FURNAS-1991]) to help create iconic programming languages for a variety of domains. Not only is the language creation process object-oriented but Vampire, based on a rule-based graphical system, supports the construction of languages which are themselves object-oriented. Attributed graphical rules, similar to those used in BITPICT and ChemTrains [BELL-1993], are the vehicle for designing the visual language.

A Vampire rule frame consists of two sides. On the left side, it contains graphical elements (i.e. constraint text and constraint graphics) "which are matched against a runtime workspace" [MCINTYRE-1995]. The right side is the action side, which contains action text and action graphics. The textual areas contain Smalltalk expressions, the graphical components icons. When a match of the left side is found, the screen area is transformed to the contents of the right side. In this respect, it is directly comparable to production rules used during the specification of textual languages. To ease the task of designing the rules, a rule editor and an icon editor are used.

After the semantics of the iconic visual programming language have been defined, a program may be created through selections on pull-down menus. Thus, Vampire doesn't use a drag&drop mechanism, or any other advanced user interaction styles found in many other related systems. Using the run-time system, the program may be executed either in a static or dynamic execution mode. In the latter mode, execution is explicitly started by the user.

It can be concluded that Vampire is a powerful system, which promotes the discovery of new visual languages by reducing the amount of lines of code necessary to implement and test a new idea. McIntyre states that the system has some weaknesses in that it is not able to restrict legal syntax in a language and the interaction style is not very advanced. However, as the system may adapt to new language paradigms in a very flexible way, the benefits of Vampire outweigh its lack of abilities. Another meta tool presented in section 4.1 tries to overcome these problems, and shows an alternative approach.
2.5.8 VLCC: A Compiler-Compiler for Visual Languages

As described in the previous section it is very complex and time-consuming to create and test a new visual language. Instead of just creating a tool that eases the design of a new visual language there is also the possibility to create a so-called compiler-compiler that not only creates the VPL but also generates the complete environment for the intended system. Several generation tools are mentioned in the literature. For example, DiaGen [MINAS-1995] is able to create diagram-based editors for visual program construction, SPARGEN [GOLIN-1993] is a visual language compiler that supports additional action routines written in C++, PROGRES [REKERS-1996] is a tool for generating both programming environments and parsing algorithms, and VisPro [ZHANG-1998] is a toolset for developing diagrammatic VPLs in a way similar to lex/yacc. Another tool is the Visual Language Compiler-Compiler (or VLCC in short) [COSTAGLIOLA-1995] which is described in more detail in the following.

VLCC is a powerful tool for the automatic generation of visual language environments. Through VLCC a designer may implement a visual language in a YACC-like fashion by specifying

- the appearance of the tokens of the language. Bitmaps are used to define the visual representation of the tokens.
- the syntax of the visual language. It is described via a particular grammatical formalism which can be both alphanumerical (positional grammars, i.e. YACC-like notation) or visual.
- the semantics of the language. It is determined by associating semantic rules to each positional or visual grammar production.

VLCC consists of two editors: a graphical editor for the design of the visual tokens (named symbol editor) and a graphical/alphanumerical editor for the definition of the syntax and semantics (named production editor). It generates C++ source code which is integrated with pre-built code and which has to be compiled in order to produce the final graphical environment which implements the desired visual language.

The final environment is a graphical editor presenting a palette with the visual tokens as specified in the symbol editor, and a menu for the typical editing functions.
like "copy", "save", etc. The menu includes the command "compile" whose behaviour is defined by the syntax and semantic specification given in the production editor.

The user of a VLCC-generated visual environment arranges the visual tokens on the screen to form a visual sentence and, then, compile them. If the syntax and semantics are correct, the command "compile" will provide the user with the semantics for the sentence, otherwise it will show an error message. So far, very simple debugging capabilities have been included in the generated final environment, but this is one of the many future directions of VLCC. According to the authors, sample visual environments (e.g. for complex flowcharts, dataflow diagrams) have been automatically generated through the use of VLCC.

The use of these generated environments show that it is hard to create real useful and efficient environments. One problem of the VLCC approach is that it is not possible to add attributes to visual elements. Furthermore, no advanced interaction methods are implemented. Therefore, the author could not base the research on VLCC.

2.5.9 Claims and Limitations for VPLs

"The advent of visual programming languages (or VPLs) brought with it many claims for their potential benefits" [GOOD-2000]. In the following, often cited claims are mentioned.

VPLs are said to make structures such as control and/or data flow more apparent [CUNNIFF-1987]. These findings are mainly based on diagrammatic languages that make use of flowchart representations.

There are also many cognitive aspects mentioned in the literature. For instance, VPLs may support forward and backward reasoning [TRAFTON-1991], [ANJANEYULU-1992], and act as a memory aid [MERILL-1993]. They are thought to allow programmers to use conceptual models closer to their own mental models, make use of the brain's pattern recognition capabilities [SHU-1988] and even make "better use of the right half of the brain, which is needlessly at rest and underutilized for the purpose of computing". Other claims are that VPLs provide information at a more intuitive and humanlike level [CHANG-1986]. Mattaini in [MATTAINI-1993] reports that research indicates that visual computer programming produces better comprehension and accuracy than do traditional programming languages based on words. Subjects
who studied control flow programs tended to perform better on questions requiring information about the details of a program, or the order in which events occur [GOOD2000b]. Blackwell made the claim in [BLACKWELL-2000] that “the human visual system is optimised for multi-dimensional data” and that text-based “computer programs are one-dimensional, not utilising the full power of the brain.” However, most of these claims are made “without citing scientific evidence.”

One topic which has lately received more attention than others is the suitability of visual programming languages for novice programming. The positive motivation stems from the belief that VPLs free the student from the syntactic complexity of text-based programming languages [MYERS-1990] and provide the novice with explicitly represented data and control flow (both graphically and spatially) [CUNNIFF-1987].

Another important argument for visual programming is that it provides support for "direct manipulation", which gives the user an impression of more direct program construction rather than following an abstract design [MYERS-1990].

On the other hand, the author has identified several shortcomings for applying VP. The following list provides an overview:

- Learning visual languages is sometimes harder than learning a textual language
- There is no (agreed) standard for a VPL and its tool environment
- There is no standardised and accepted set of icons (e.g. one symbol for print which can represent write, printf etc., cf. open icon in Windows applications)
- Mixture of text and visual elements is unclear. What is a good design?
- "Poor" languages and "wrong" approaches due to the neglection of the visual environment (i.e. wrong focus of previous research)
- No linkage to visual modelling
- Screen space limitations
- Limited navigation techniques

It can be concluded that the many claims made for the benefits which VPLs might offer, very little empirical evidence exists to back up those claims. Furthermore, most aspects are not directly related to more complex programming processes. In addition, the author could not find specific evaluations for component-based development that rely on visual programming.
3 Related Research in the VL Field

3.1 Overview

In the following, an overview of related research is given. This is done by providing two different surveys. These surveys shall justify and motivate the focus of the research.

The first survey reviews experimental findings from related topics. The objective of this survey is to identify existing results which may be applied to this research. It follows a survey of academic and professional visual programming languages which are related to this research. The main focus lies on visual object-oriented languages found in the literature and the status they have reached.

3.2 Survey 1: Review of Experimental Findings from Related Topics

Though this research focuses mainly on visual object-oriented programming language aspects this work has also relationships to other research areas. The most important inter-relations are with Human-Computer Interaction (HCI) and Applied Cognitive Psychology/Psychological Research. In the following, some important findings from these fields that are relevant for this research are mentioned. Of course, this cannot be a full treatment of all aspects of these areas. For this research, the debate about metaphors and diagrammatic representations is of some importance, and of special interest are the references in the literature that relate to visual object-oriented programming languages.

For more than 20 years, computer-based information has been presented in a textual form as well as graphically. Generally speaking, the claim and justification for this is that the use of metaphors in a graphical form makes them easier to learn, to understand and to apply. One rule of thumb is: “Designers of systems should, where possible, use metaphors that the user will be familiar with.” [FAULKNER-1998, p. 89]. The usefulness of metaphors is also stated in many textbooks for undergraduates: “Very few will debate the value of a good metaphor for increasing the initial familiarity between user and computer application” [DIX-1998, p. 149]. With respect to visual programming languages it is important to mention that diagrams (and diagram-based metaphors) are believed to assist with abstract reasoning [BLACKWELL-1998].
Practitioners and researchers in Human-Computer Interaction (HCI) believe that the value of diagrammatic representations as used in visual modelling is derived from metaphorical reasoning (i.e. comprehension, problem solving and explanation) and their ability to communicate abstract information. These findings may also be relevant for visual languages. However, no research has been carried out to test this claim in a component-oriented development environment that makes use of both visual modelling and visual programming techniques.

The conclusion which can be drawn from this is that the use of graphical user interfaces which combine text and (diagrammatic) images are of great importance in order to improve usability. In addition, graphical user interfaces provide many advantages through "direct manipulation", see [SHNEIDERMAN-1983], [LEWIS-1991].

Efficient techniques to minimise user interactions (e.g. for selecting and connecting graphical elements) are now in use in many systems. All important graphic/modelling tools and CAD/CAE/CAM systems apply these findings.

In order to understand why one approach or technique is superior to another, the work of applied cognitive psychology and psychological research is relevant. One important contribution is what a metaphor contributes to comprehension, explanation and memory tasks. The surprising result is that explicit metaphors provide little benefit for cognitive tasks using diagrams as an external representation [BLACKWELL-1998]. This is a contrary statement to that usually expressed with HCI.

Different cognitive models have been developed to understand the visio-spatial working memory. However, the problem is that no one can really predict how effective tasks are carried out by users as many inherent problems exist, such as different cultures and different user types (e.g. visually oriented or not!). Thus, user evaluations have to reveal whether one approach is effective or not.

There is some evidence that concrete and diagrammatic representations lead to improved comprehension and memory [MAYER-1989]. Larkin and Simon [LARKIN-1987] discusses the advantages of images for maintaining information about the relationships among components, as compared with more linear textual presentation. Though some people are more visual (i.e. have greater visual aptitude) than are others Cunniff [CUNNIFF-1987] found that graphically presented programs are comprehensively better for nearly all subjects whatever their visual aptitude.
The benefit of metaphor in diagram use is largely restricted to mnemonic assistance, especially when the user constructs his own metaphor instead of the systematic and recommended HCI metaphor (e.g. desktop metaphor). Pictorial content in the diagram has improved problem solving far more than explicit metaphorical instructions. "Potential advantage arising from metaphor in diagrams is a mnemonic one" [BLACKWELL-1998].

Diagrams are considered to be an effective "cognitive artefact" [NORMAN-1991]. Furthermore, diagrams are best described by the contra-distinctions "text" and "pictures". The cognitive processing of text is closely related to auditory verbal comprehension whereas the construction and interpretation of pictures rely on some arbitrary "depictive" conventions. This means that "all texts are to some extent diagrammatic, and all pictures are to some extent diagrammatic." [BLACKWELL-1998, p. 9] The direct result of this is that all findings for diagrams are directly applicable to VP.

But as Blackwell mentioned, within cognitive psychology there is a debate on theories for metaphor interpretation (e.g. describing diagrams as metaphors rather than as structural or pictorial allegories). In addition, these have seldom been applied to diagrams or to HCI.

The value of diagrams has been demonstrated by various researchers (e.g. [BEVERIDGE-1987]). Diagrams are used during problem solving as external representations that supplement working memory and efficiently express problem constraints. In addition, graphs allow more rapid judgements than tables [WASHBURNE-1927]. But [J.MEYER-1997] showed that the these results have been unjustified.

Most aspects of diagrammatic notations for programming, in particular research on ergonomic factors, have been published in the context of psychology of programming. However, as stated in [BLACKWELL-2000], "there is no specialist forum for describing the psychological factors associated with visual programming, or general use of diagrams in software engineering". This makes it hard to decide upon the quality of a specific VPL.
Thus, an important point remains: which of the previous findings are relevant to the research of (visual) programming languages? Green developed a framework called "Cognitive Dimensions" [GREEN-2000]. This framework constitutes a small vocabulary of terms (e.g. abstraction gradient, consistency, secondary notation, viscosity etc.) describing the cognitively-relevant aspects of structure of an information artefact and may be used to compare the relative advantages of different programming languages.

From the point of view of designers, there are important trade-off relationships between the cognitive dimensions, therefore the position of an artefact in "cognitive dimension space" cannot be adjusted arbitrarily [BLACKWELL-2000]. The problem with this is that improving one dimension (e.g. viscosity) is likely to affect other dimensions. Thus, an "optimal" setting of these dimensions cannot be formulated and applied to the design of a new visual system. However, as Green and Petre mentioned, this framework is also useful outside the specialist world of HCI and may give designers a better chance of avoiding oversights by explicitly listing and naming the dimensions [GREEN-1996].

[GURKA-1996] reported that program visualisation assists students with learning about algorithms. They and others (e.g. [MULHOLLAND-1993], [BLACKWELL-2000]) also noted, however, that the value of program visualisation had not been investigated in much depth in experimental studies. However, the intuition is that visual languages may help in the task of program comprehension.

The representational features of diagrams induce subjects to adopt different problem solving strategies, which result in different task performances as many experiments show (e.g. [GREEN-1989b]). Studies in the field of visual programming show that flowcharts can outperform text for certain restricted cognitive tasks in either time or correctness (sometimes in both) by the ratio between 1.7 and 2.5 [SCANLAN-1989] but that, in the context of the entire programming process, flowcharts do not substantively outperform textual programming notation [CURTIS-1989]. Similar results have been observed by Vessey and Weber [VESSEY-1986] and by Cunniff and Tayler [CUNNIFF-1987].

The experimental study reported in [CHATTRATICHART-2000] compared how three most common directional representations (namely arrow, line, juxtaposition) affect users' performance. With respect to response time there is statistical evidence that the arrow has the fastest response time. Factor 1.51 between "arrow" and "juxtaposition" — factor 1.44 between "arrow" and "line". With respect to accuracy the
experiments carried out revealed the main effect that juxtaposition is the most error-prone. Furthermore, participants indicated in questionnaires an overwhelming preference for the arrow over the line and the juxtaposition.

Though it is claimed that the use of arrows is the best choice, it is important to notice that the experiments used a "maze" which represented a route map with possible routes connecting $n$ starting points with $m$ destination points. Thus, this study didn't include a real visual programming language. The question arises therefore whether these results are transferable to VP. One argument is that for a programming task the factor may be completely different, and perhaps irrelevant!

There are only a few studies that focus upon real programming aspects. One well studied concept is that of recursion. A case study carried out by Good [GOOD-2000] evaluates different types of graphical representations. The study suggests that differences in understanding are associated with differences in the representation, and that even when representations are "informationally equivalent" [LARKIN-1987], differences in presentation may have a detrimental effect on the student's understanding of recursion.

Difficulty with recursion is thought to stem from several sources: recursion is not a concept which is often encountered in everyday life, thus making it difficult to find satisfactory analogies to use in explanations [ANDERSON-1988]. Anderson also claims that novices may reach impasses due to an inability to regard recursion as anything other than a control construct, rather than, for example, reflecting on its effect on input data, and the relationship between input and output data. Mentally executing a recursive call can require large memory resources which humans may not possess [ER-1984], particularly if the call is non-tail recursive. Finally, novices have difficulties with the idea of passive control passing [KURLAND-1983]: they mistakenly think that once the base case has been reached, processing stops, rather than realising that control is passed back to a suspended procedure, whose execution carries on where it left off. This type of misunderstanding is most likely responsible for the widely reported "looping' model" of recursion, in which recursion is not differentiated from iteration [KURLAND-1983], [BHUIYAN-1991], [KAHNEY-1989].

Novices may benefit from using representations which attempt to respond to these difficulties. The representations could, for example, highlight and make explicit various aspects of recursion, such as data flow or passive control passing. They could
also act as a memory aid by allowing novices to "offload" intermediate values during the arduous task of mental execution.

One can conclude from the previous description that the research priorities in the psychology of programming community lie on the use of diagrams in programming. Further, many research questions on VPLs are still unanswered. Especially, since the empirical studies concentrate in nearly all cases on low-level and partial aspects of programming processes they cannot provide a clear statement to the usefulness of new visual systems. Therefore, it is essential that further research should concentrate on real tasks carried out with the help of a visual environment and language. One good example is the Visual Programming Challenge [VL-1997] which clearly showed that some visual approaches are very generic and powerful. Unfortunately, the task was not related to real-world problems. Thus, it is still doubtful whether visual programming is superior to text-based programming. No empirical study could be found in the literature that evaluates visual object-oriented programming languages in particular. Therefore, this research may contribute important findings to this type of programming.
3.3 Survey 2: Visual Object-Oriented Programming Languages

3.3.1 Overview

There has been little research on visual object-oriented programming. One reason for this may be that the VL community is more interested with formal definitions and theoretical foundations of VPLs. Another reason could be that the effort of implementing visual object-oriented programming systems is very high and therefore left to the industry. There are not many commercial products available which include a visual programming language at all. The following table lists the most important commercial systems.

<table>
<thead>
<tr>
<th>Language / Product</th>
<th>Company</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HP-VEE</td>
<td>HP</td>
<td>Visual creation of computer-based measurement and automation software.</td>
</tr>
<tr>
<td>Prograph</td>
<td>Pictorius (<a href="http://www.pictorius.com/">http://www.pictorius.com/</a>)</td>
<td>Visual Creation of complete object-oriented applications.</td>
</tr>
<tr>
<td>Visual Age for C++</td>
<td>IBM Corp.</td>
<td>Creation of (GUI-based) C++ applications.</td>
</tr>
</tbody>
</table>

Table 10. Commercial products including a VPL.

Interestingly, there are two domains where VP is used in commercial environments. One is the creation of measurement and automation software (see HPVEE, LabView) and the other are domain-independent visual object-oriented systems (see Prograph, Visual Age for C++). In the subsequent sections the academic and commercial visual object-oriented programming approaches, found in the literature, are described in more detail.

3.3.2 Vista

Vista (Visual Software Technique Approach) [SCHIFFER-1998], [BURNETT-1995] is a visual programming environment designed for software engineers. It is based on a visual multi-paradigm language which is firmly linked to the development system. It
supports a mix of textual and graphical notation. Using Vista, it is possible to write event-driven and data-transforming Smalltalk applications.

A program in Vista consists of a hierarchy of building blocks called processors. Programming takes place by specifying processors and connecting them into a network. The interaction and control flow is triggered through tokens which are sent via the I/O ports of the processors. There are several kinds of processors, such as data processors or signal processors. Vista includes many useful features such as abstract processor classes, prototypes, and self-modification [SCHIFFER-1998]. The figure below shows two sample processors: a signal processor and a data processor.

![Figure 10. A visual timer construct and a list processor [BURNETT-1995, p 210].](image)

In summary, Vista contains a very powerful programming model and a high degree of visual expressiveness, such that it outperforms many other VP systems. Though Vista makes use of visual representations it is not a complete visual programming environment as the user still has to understand and write Smalltalk code. Schiffer and Fröhlich [SCHIFFER-1998] argue that a discussion on how to incorporate software engineering principles has to take place. They are convinced that combining visual with object-oriented programming will yield significant improvements relevant for professional software production. One shortcoming of their approach is that VISTA only supports the visual construction of "semi-finished components". Therefore, they did not evaluate how a complete visual object-oriented programming language could be designed.
3.3.3 Ispel

One important research aspect lies in software development environments which support integrated visual and textual programming. Of interest is whether it is possible to specify a program using both techniques with full bi-directional consistency management. Included in this aim is the support for collaborative visual programming, version control and configuration management for visual programs.

One interesting approach is Ispel, a visual programming environment for object-oriented languages developed by Grundy et al. [GRUNDY-1991]. The main purpose of Ispel was to form the basis of a visual approach to programming with object-oriented languages, namely Kea and Eiffel. They build both an environment which supports these facilities and an environment generator/00 framework to facilitate the construction of such systems.

Ispel is based on the idea of diagrams which represent the class structure and the inter-relationships such as generalisation (inheritance) and aggregation (feature hierarchies). Furthermore, it uses the desktop metaphor (i.e. windows, views, menus, and dialogues). Class structure diagrams were regarded as useful in several areas of object-oriented programming, e.g. within Analysis and Design (to present the structure of the program), within Documentation and Browsing (to understand and maintain the program), within Implementation (to form the basis as well as to construct all or part of the executable program), and within Debugging (to describe the execution state of the object-oriented program).

The most interesting aspects of Ispel are the use of multiple diagrammatic and textual views simultaneously, and the emphasis on consistency management. Furthermore, multiple programs can be constructed simultaneously, and both the graphical and textual aspects of programs can be saved to files. As a diagramming tool, it provides the usual features (i.e. drawing boxes and lines etc.).

Not all programming is performed using visual or graphical elements. For instance, the implementation of feature bodies is coded in text. It was argued that the expression level aspects of Kea and Eiffel are better suited to textual construction. Hence, the textual representation of classes is integrated with the visual programming. In order to achieve consistency, Ispel is grounded on the architecture, shown in figure 11.
In the author's opinion, Ispel is a very good approach towards the integration of visual programming within OT. However, further work has to be done. Hence, Grundy et al. decided to develop a full implementation of Ispel using C++ with an X Windows graphical user interface. One shortcoming of this approach is that it is not a complete visual approach.

3.3.4 VIPR

VIPR (Visual Imperative PRogramming) [CITRIN-1995] is a completely visual object-oriented programming language based on the flowchart idea. It was inspired by Kahn's Pictorial Janus and tries to eliminate the known yo-yo problem [BUDD-1997] within textual object-oriented languages. VIPR programs are represented as nested circles together with textual elements. As programs are directly executed (i.e. programs conform to an "executable specification") , VIPR is a complete visual object oriented programming language.

VIPR's semantics derive entirely from graphical rules and provide a unifying framework for visualising static and dynamic program execution. Thus, the user of VIPR "does not need to understand two different (albeit related) sets of semantics, the semantics of the visual model and that of the underlying textual model" [CITRIN-1995].

The following figure shows how the Tcl code is translated into the equivalent visual specification. It contains a simple while loop. The program counts down from 5, sums the numbers as it counts down, and prints out the result.
set x 5; set Sum 0
while [expr $x > 0] {
    set Sum [expr $Sum + $x]
    incr x - 1
}
write $Sum

Figure 12. The definition of a main procedure with a while loop.

The most interesting concepts of VIPR are the object-oriented features. VIPR supports the definition of classes, inheritance, polymorphism, and dynamic dispatch. Figure 13 shows a sample class definition. As the figure shows, "the VIPR definition groups methods and instance variables by enclosing them with a dotted line" [CITRIN-1995]. Method calls between objects are specified by just pointing from one circle to another circle.

In the author's opinion, VIPR is an interesting approach in the shift to visual object-oriented environments. The idea to make an explicit representation of object-oriented aspects such as dynamic dispatch is certainly a step in the right direction. Control-flow problems that always exist in VPLs are also handled in an effective way [CITRIN-1993]. However, the visual representation of complex programs doesn't provide a clear and less complex representation. "There are also negative aspects of the current implementation of VIPR that detract from its overall appropriateness. Containment addresses some aspects of the scalability problem" [CITRIN-1998].
class point {
    int x, y;
public:
    void moveTo(int newx, int newy)
    { x = newx; y = newy; }
    int xDistance(point* p)
    { return p->x - x; }
};

3.3.5 Prograph

Prograph [STEINMAN-1995] is a commercially available visual data-flow programming language based on icons and is intended for a wide range of applications (e.g. graphics, word processing, scientific/business software, communication systems, and so on). Programming takes place by "wiring" graphical elements (icons) together. The execution of instructions is data-driven (i.e. instructions are processed when all input data which are necessary for the execution are available). In other words, the program is represented as visual operations interconnected by links through which data "flow". The integrated development environment consists of several components, such as a program editor, a code interpreter, a graphical debugger, and a user interface builder.

Figure 13. The definition of a class point.
If time-critical parts of an application have to be developed, low-level languages like C may be linked as external code. Further, Prograph has the ability to execute programs in a less order-dependent fashion which in principle would allow parallel execution. While other parts of the program are tested by the debugger, new program code may be added which is a very flexible mechanism. When a program is fully tested, the code compiler produces a fast stand-alone version of the application.

The authors claim that Prograph may be used for rapid program development (i.e. prototype and finished software) for a wide range of programming tasks and that its development environment is easier and more efficient than traditional programming environments based on textual languages. It reduces the amount of code that needs to be written and minimises the possibilities of errors in program code. Furthermore, program designer do not need to worry too much about "picky little" (syntactical) "details" [STEINMAN-1995]. Within the Prograph environment, there is no need for the translation step. Operations and elements are presented as pictures or icons with the effect of ease of use and no reduction of execution speed in the completed program.

There are two different programming styles/philosophies: the traditional structural procedural programming mode, and the object-oriented programming style.

An application (or project) is subdivided into several sections (or modules). Each section consists of three components, namely classes (for the OO-code), universal methods (for the procedural code), and persistence (for the global date). Sections are intended to be reusable. For instance, the application builder classes framework provides a set of ready-to-use classes and code modules (such as application, menu, document, data, window, standard menu, etc.) respectively.

A special code library for relational databases eases the development of database applications. Further, OO U/I and application framework code modules are reusable. One major aim of Prograph is to reduce the amount of lines of code a programmer has to type in. This is achieved by specifying references or interrelationships between classes. Built-in data types such as lists ease the specification of special processing.

Since Prograph seems to be a powerful and complete visual system it was judged to be evaluated by a usability trial (see chapter 6). An introduction to Prograph may be found in appendix Q (see evaluation 1).
3.3.6 Visual Age for C++

One relatively new method for visual programming is used by the Visual Age product family (e.g. Visual Age for C++) of IBM (see also appendix M). It is based on the Construction-from-Parts-Paradigm. A part represents an object class plus the corresponding interface. The part interface includes attributes (i.e. member data), actions (i.e. services or operations), and events.

Basically, there are visual and non-visual parts. In the first category are, for example, edit fields or scroll lists needed for the construction of user interfaces. Non-visual parts are used for the program logic such as calculations. A user is able to define new visual and non-visual parts. Through connecting available parts, actions are carried out. The connections of the different parts are standardised. The following kind of connections are possible:

<table>
<thead>
<tr>
<th>Connection type</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute-to-attribute</td>
<td>Connection of two data values which need to be synchronised.</td>
</tr>
<tr>
<td>event-to-attribute</td>
<td>Data values which need to be changed due to an event.</td>
</tr>
<tr>
<td>event-to-action</td>
<td>Actions which are to be started when a certain event occurs.</td>
</tr>
<tr>
<td>attribute-to-action</td>
<td>Actions which are to be started when the event ID of a data value receives a signal.</td>
</tr>
<tr>
<td>event-to-member function</td>
<td>Execution of functions after a certain event has occurred.</td>
</tr>
<tr>
<td>attribute-to-member function</td>
<td>Execution of functions when the event of a data value receives a signal.</td>
</tr>
<tr>
<td>custom logic</td>
<td>Any code parts which may be triggered by events or event IDs.</td>
</tr>
<tr>
<td>parameter</td>
<td>A parameter for an action or function.</td>
</tr>
</tbody>
</table>

Table 11. The different connection types within VisualAge for C++.

The advantage of the part technology is that major components of an application may be built through the reuse of parts. Thus less programming is needed and it is not necessary to build applications from scratch. The central development component of Visual Age for C++ is called the "Visual Builder". The development environment under MS-Windows 95/97/NT consists of several tools. For simple applications, only the visual builder and a command-line session (for the purpose of compilation) are needed. Additional tools (such as the composition editor for connecting the parts and the parts interface editor for the definition of attributes, actions and events) are started directly from the visual builder tool. It is important to note that Visual Age for C++ doesn't allow one to visually code the behaviour of the parts.
4 Overview of Different Approaches
Since the start of the PhD work different ideas have been identified for how VP may be integrated into OO technologies to design and implement both a VOOPL and a VOODE. In this chapter these ideas are described and this therefore serves as a record of the different approaches identified by this research. The following table gives a short overview:

<table>
<thead>
<tr>
<th>Approach</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1&lt;sup&gt;st&lt;/sup&gt; approach</td>
<td>This approach addresses the design of a VPL called VPL-C that is integrated in a visual environment called Visual Meta Builder tool. The generation of an application should be possible by the means of different types of processors. The Visual Meta Builder (VMB) was developed to ease the task in designing a new visual programming language. Object-oriented extensions led to VPL-C++.</td>
</tr>
<tr>
<td>2&lt;sup&gt;nd&lt;/sup&gt; approach</td>
<td>The first approach was expanded to show its use for dialog-oriented (i.e. user centric) applications. This included UML as the core modelling notation and aimed at the visual construction of complete applications. The support of dialog-oriented applications was made possible by the introduction of new diagram types. Specific parts of an application are predefined (e.g. command handlers) and may be hidden from the user. The overall application structure is mainly based on statecharts or state slots which could be individually defined by the programmer.</td>
</tr>
<tr>
<td>3&lt;sup&gt;rd&lt;/sup&gt; approach</td>
<td>Here a framework-based VOODE (i.e. parts of an application are predefined) was investigated which defines slots that may further be refined by visual code. This approach could also include design patterns [GAMMA-1995] or framework patterns [BUSCHMANN-1996]. VOOPL would not cover all aspects of a complete visual language but only provide code elements which are necessary to refine or extend the selected slot.</td>
</tr>
<tr>
<td>4&lt;sup&gt;th&lt;/sup&gt; approach</td>
<td>The fourth approach concentrates on the design and implementation of component-oriented systems. UML is used to design a CORBA component. The implementation is then carried out by a visual editor. The underlying techniques are based on the findings of the first approach. This includes both the construction of the editor and the VOOPL.</td>
</tr>
</tbody>
</table>

Table 12. Overview of different approaches identified by this research.
4.1 Approach 1: Visual Meta Builder

4.1.1 Background and Motivation

The initial hypothesis of the first approach was that if graphical user interfaces incorporating different levels of visualisation have proven to be highly beneficial for human-computer interaction then the same should certainly be true of programming tasks. Furthermore, it was highly predictable that visual programming systems would play a dominant role in the near future, as they offer the ability to build systems to those without a rigorous Software Engineering background. This was borne out by the creation of new commercial development systems such as the Visual Age product family based on visual programming techniques (here the parts paradigm) [NILSSON-1997] and the attractive OO paradigm.

The existing approaches and visual development environments (e.g. DiaGen [MINAS-1995], SPARGEN [GOLIN-1993], Visual Age for C++ [NILSSON-1997], Visua [TRIPLES-2000], Prograph [PICTORIUS-2000], LabView [NI-2000]) suffer from several shortcomings. Most tools do not support a flexible layout of a target visual editor and extension mechanisms. Furthermore, they provide a limited range of fixed interaction capabilities. New ideas are needed to improve this situation. When the author started the research the first goal was the identification of existing VOOPL approaches and to identify their shortcomings. It soon became clear that there was little ongoing research in the field of VOOP.

Therefore, it was decided to start the creation of a new visual object-oriented programming language. However, the design and implementation of a new visual programming language is a difficult and time-consuming task. In addition, the approach should be sufficiently flexible to allow different ideas to be researched. Another problem was that a static definition of a VPL is not sufficient as the usability of a VPL is closely related to the environment in which it is used. No existing tool seemed to be of much help. Therefore, a new tool was needed that supported the initial research goals.

Within the first approach, the author created a new meta tool which allowed the creation and usage of a visual programming language where "attributed icons" could be used as programming elements. This approach differs from existing techniques (see for instance Vampire [MCINTYRE-1995] and VLCC [COSTAGLIOLA-1995]) as it is based
on an easy-to-use description language, each lexeme of which is represented as an individual node on the workplace and is described by a special code file. It was important to investigate how visual elements/techniques could be integrated into a concrete visual environment which supported the programming task. Thus, the emphasis lay primarily on the visualisation of program structures and the possible definition of visual code elements. It was of great importance to keep the underlying semantics separate from the visual representation as this allowed the ongoing research to remain as open as possible.

4.1.2 Basic Idea

When designing a new visual language, a formal basis and an underlying visual metaphor is needed. The formal basis could, for instance, rely on a mathematical theory or formalism or on a higher level building block. In this instance it is not necessary to invent a whole new (visual) language from scratch but to take a proven textual language and extend it so that it is well embedded within a visual environment.

A suitable metaphor applied to the construction of an application could be for instance “the principle of substitution” (i.e. using specialisation), “the key-keyhole idea” (i.e. using software components which may be dragged into a pre-defined slot), or “the building blocks idea” (i.e. using composition of black-box components). So, when designing a meta tool, it is of great importance to use a generic underlying principle. Possible metaphors are shown in the figures 14 through 16.

**Figure 14.** The substitution metaphor.
A key representing a software component may be dragged into a keyhole (i.e. a pre-defined slot within a framework).

Figure 15. The key-keyhole metaphor.

Like building a house, an application is constructed by adding possible blocks on top of each other.

Figure 16. The building blocks metaphor.

In the first approach, the VPL that was created was based upon the processor metaphor which was mainly inspired by Vista [see 3.3.2] and focused on a complete visual programming language. The programs written by users are visually represented by 3-dimensional trees. Generally speaking and over-simplifying, a processor specifies an "action" which will be performed on an "object" after it was triggered by an incoming event. Further, the VPL was oriented towards ready-to-use components. The result of this was a language called VPL-C that was then regarded as an "open meta language". The author means by that, that both the syntax and the semantics are defined in such a way that different kinds of language classes may be supported (at least in theory). Furthermore, this resulted in a separation between the syntactical representation of the language elements and their semantic interpretation.

This approach may be regarded as very generic with respect to language design and development environment. One further possibility to extend the system, for instance, would have been that the language design could further be supported by a processor building toolkit. The general idea is shown in the next figure.
4.1.3 Language Design

In this paragraph, the author will outline some design considerations. This is important in order to keep track of the decisions made. After it was decided to make use of the metaphor based on the processor idea, the first task was to invent a VPL that eased the coding of procedural programs. As a starting point, the language C was chosen and an attempt was made to translate it into a VPL called VPL-C. The following list outlines some basic design rationales:

- A VPL-C program is represented by a 3-dimensional graph of icons.
- An icon is represented by a processor bitmap.
- An icon + attributes specifies a lexical unit which is translated into a C function call (including the corresponding parameter list).
- The semantic of the action (i.e. C function) is implicitly defined.
- The scope of an attribute is determined through the position within the graph.
- Attributes linked to an icon could have different modes: editable, visible but not changeable, or invisible.
- The flow of nodes represents the sequence in which the actions (and their corresponding C functions) are performed.

Figure 17. Architecture of the visual environment.
- Attribute values may be shared (dynamically at run-time) between different processor elements.

The following figure illustrates the general idea.

![Diagram showing processor icon and attribute sharing]

**VPL-C**

```
lexeme
function call : fn(<parameter list [attr]>)
```

**Figure 18.** A processor icon represents a function call.

The first idea can further be extended towards a VPL that supports object-oriented aspects. The author considered a language called VPL-C++ which would be based on C++ language aspects.

The following list shows the design rationale of this language.

- A VPL-C++ program is represented by a 3-dimensional graph of icons.
- A node represents an object instance with a unique identifier.
- The activation of a node is triggered by an incoming token (:= event + message) and represents one internal method call.
- The system allows the definition of user classes (including attributes and methods), and in addition association and inheritance relationships.
- There are pre-defined system object classes/instances (and methods) which allow the instantiation of user-defined objects.
- A notification framework and a central object manager form the basis of the visual system which have to be kept transparent to the user.
- VOODE/VPL-C++ is a complete visual and object-oriented system.
The following figure illustrates the general idea.

```
message + method + attributes + message

object instance

message call: <object> -> <method>(<parameter list [attr]>)
```

**Figure 19.** An icon represents an object's method call.

One possible extension could have been the support of and integration into OOA/D. This would have been possible by introducing specific block and modelling processors.

The following figure gives sketches of processors needed for VPL-C++.

```
new object

delete object

class def

relationship

inheritance

send message

activate service
```

**Figure 20.** VPL-C++: class/object handling plus event handling.
4.1.4 Introduction to the Approach

In the approach presented here the processor idea is used wherein, simply, a processor specifies an "action" which will be performed after it has been triggered by an incoming event. Thus, a processor contains an imperative character. Further, processors are oriented towards ready-to-use, or reusable components. A program is called a processor program and consists of many inter-connected processors placed on different layers of a graph.

In general, there are several ways in which a visual language is executed or transformed in order to run an application. Firstly, visual symbols may be directly interpreted and executed (-> visual interpreter). Secondly, a visual program may be directly compiled into an executable image (-> image animation, "visual" compiling, direct computing). Thirdly, a visual program may be transformed into an intermediate format which, in turn, is analysed and executed (-> indirect or language independent computing). Lastly, a technique could be applied so that code of a (semantically equal and known) textual language program is generated and "traditional" tools may be used to compile, link and run the application (-> language transformation).

As a graph is considered for the visual representation of a program it would be possible, for instance, to make use of "triple graph grammars" [SCHUERR-1994], which are used to specify "rather complex graph-to-graph translations as languages of graph triples". In this case, a processor graph and a graph based on the syntax and semantics of the underlying textual language together with their graph interrelationship would be defined. An incremental graph parser would then be needed for the execution of context-sensitive productions.

Another technique is supported by the tool shown in figure 21. The processor nodes are defined by individual code files and transformed into their textual representation by scanning the graph and using their node description. An external transformer \( \tau \) is then necessary to generate the equivalent textual code (e.g. C++ code). Using this technique, the tool may be used as a general-purpose visual language system and may then be regarded as an alternative to Vampire. However, it should be noted that the tool presented here supports programming-in-the-small rather than programming-in-the-large. To enable programming-in-the-large several features such as the creation of packages, the use of libraries and more enhanced navigation techniques would be required.
One key aspect in designing a meta tool is that of domain-independence. To achieve this, a special node description language (here called a *node modelling language*) was developed. Both the syntax and the semantics of a processor are defined in a way such that different language types may be supported. This point is discussed in the next section, in which the theoretical underlying principles of the formal language are described.

Figure 21. The general architecture of the transformation process.
4.1.5 The Underlying Theory

Before continuing further, there will be a more formalised description of the applied techniques and principles. In general, when designing a new textual or visual language many different aspects such as orthogonality, efficiency and program notation have to be considered. First there will be a description of the visual representation of a processor program, viz. by a graph. Then the underlying node modelling language will be presented. It is shown that any context-free language may be used for translation into a visual programming language.

4.1.5.1 Program Representation by a Graph

As mentioned earlier, a processor program is represented by a 3-dimensional graph.

A graph $G$ in its basic form is a set of nodes $N$ which are connected by a set of edges. As edges are used to describe relations $R$ between nodes, a graph is defined as:

\begin{equation}
G = \{N, R\}
\end{equation}

Graphs are used in virtually all branches of computer science. Thus, the author considered it to be a good starting point for the definition of a visual language. However, certain restrictions on the relation $R$ are necessary to yield special classes of graphs supported by the Visual Meta Builder. Many different extensions of graphs are described in the literature, e.g. hypergraphs or higraphs in [HAREL-1988].

Several issues are important for the definition of the graph. To ease the users' interactions, data and parameters which are being processed may be presented in different visual ways and are directly linked to the nodes or edges on different layers $l \in [1, \text{max}]$. To make the coding flexible and to allow dynamic construction of new processors, a node modelling language has been specified. Like the hierarchy tool in VisaVis [POSWIG-1996], an instance is needed to keep track of the various defined processors and to observe the construction of well-defined processor programs.

---

4 Orthogonality means "that there should be not more than one way of expressing any action on the language" [MARCOTTY-1986].
A processor program $V$ is defined as a 3-tuple:

\[(2) \quad V = (N, R, \alpha)\]

where $N$ is a finite set of node elements \{\(n_0, \ldots, n_m\)\} called nodes or processor nodes, and $R$ the set of (attributed) relations \{\(r_0, \ldots, r_n\)\} between any two nodes, defined as:

\[(3) \quad R \subseteq N \times N\]

The function $\alpha$ assigns to a processor node $n_i$ a set of processor attributes $\alpha(n_i)$.

As a processor program is cycle free, the possible graphs are restricted. Given the function $r$ which connects a node $n_i \in N$ to a node $n_j \in N$, $r$ is defined as $r : N \rightarrow N$. If $r^0(n_i) = \{n_i\}$ and $r^1(n_i) = \{n_i\}$ then $r^{i+1} = r(r^i(n))$. $r^i$ may be interpreted as the reachability of a node within $i$ steps and is called relational depth. It follows that:

\[(4) \quad r^+ (n) = \bigcup_{i \in \{1, 2, \ldots\}} r^i (n)\]

- with $r$ restricted so that $n \notin r^i(n)$

The topmost node of the graph is the root node. Only one root node may be specified within a processor program $V$. This node (denoted as $n_{\text{Root}}$), which may only be placed once on a workplace, specifies important project parameters (e.g. project name and type). Given function $\Omega$ returning the count of an existent processor type, and function $\lambda$ returning the layer attribute value of a node, the graph definition is restricted accordingly:
(5) \( G = \{N, R, \alpha\} \) so that \( n_{\text{Root}} \in N \),
\[
\Omega(n_{\text{Root}}) = 1, \quad \text{and} \quad \lambda(n_{\text{Root}}) = 0
\]

- with 0 denoting the topmost layer

Given the above description of a graph, the question arises of how a lexical unit (called a lexeme) may be represented as a node within the graph. A processor represents an action executed at a given time. Thus, a general transformation could be that of a function \( f \). The advantage of using a function is that the intended action may be changed by using different function attributes, which can be called specialisation.

A textual language which is intended to be visualised is traditionally described by a set of terminal symbols \( t \), a set of non-terminal symbols \( u \), a set of production rules \( p \), and a start symbol \( s \). A language is then described in BNF notation by the following (simplified) equations [cf. Wirth-1996, p 7]:

\[
\begin{align*}
\text{syntax} & := \text{production syntax} | \emptyset. \\
\text{production} & := \text{identifier} "=" \text{expression} ".". \\
\text{expression} & := \text{term} | \text{expression} "\mid" \text{term}. \\
\text{term} & := \text{factor} | \text{term factor}. \\
\text{factor} & := \text{identifier}.
\end{align*}
\]

In the first approach, both terminals and non-terminals may be described by means of processor nodes. In its simplest form, a node may represent one or more terminal symbols or one non-terminal symbol from one equation. To allow the direct transformation of textual languages written in (E)BNF into equivalent visual languages, non-terminals may be used as so-called block nodes. Block nodes may be refined by other (terminal and/or non-terminal) nodes placed on another layer of the graph. In this way, the transformation is not strictly fixed, which gives the language designer the freedom to group the nodes according to given needs. Next, a simple example is presented to highlight the basic idea.
Given the following set of equations $P$

\begin{align*}
(p_1) \text{AnyTextualLanguage} &::= \text{NonTermA NonTermB}.
(p_2) \text{NonTermA} &::= \text{termA1 termA2 termA3}.
(p_3) \text{NonTermB} &::= \text{termB1 NonTermC termB2}.
(p_4) \text{NonTermC} &::= \text{termC}.
\end{align*}

The equivalent equations $P'$ of the visual language could look like this (version 1):

\begin{align*}
(p_1') \text{AnyVisualLanguage} &::= \text{Root-Node}.
(p_2') \text{Root-Node} &::= \text{A-Node B-Node}.
(p_3') \text{A-Node} &::= \text{a-node}.
(p_4') \text{B-Node} &::= \text{b-node}.
(p_5') \text{C-Node} &::= \text{c-node}.
(p_6') \text{a-node} &::= \text{FA(attr_a1 attr_a2 attr_a3)}.
(p_7') \text{b-node} &::= \text{FB(attr_b1) C-Node FC(attr_b2)}.
(p_8') \text{c-node} &::= \text{FC(attr_c)}.
\end{align*}

The right side in equation $p_7'$ is semantically equivalent to "FB(attr_b1 attr_b2) C-Node" as the order of attributes within a node is irrelevant. Another transformation could look like this (version 2):

\begin{align*}
(p_1') \text{AnyVisualLanguage} &= \text{Root-Node}.
(p_2') \text{Root-Node} &= \text{A-Node B-Node}.
(p_3') \text{A-Node} &= \text{a-node}.
(p_4') \text{B-Node} &= \text{b-node}.
(p_5') \text{a-node} &= \text{FA(attr_a1 attr_a2 attr_a3)}.
(p_6') \text{b-node} &= \text{FB(attr_b1 attr_b2 FC(attr_c))}.
\end{align*}
In version 1 of the example, a well-formed sentence of the visual language is "FA(attr_a1 attr_a2 attr_a3) FB(attr_b1 attr_b2) FC(attr_c)". In version 2 it is "FA(attr_a1 attr_a2 attr_a3) FB(attr_b1 attr_b2 FC(attr_c))". Both sentences are semantically equivalent. In the latter case, the resulting b-node is more complex, with the benefit that the number of nodes is limited. The relationship between the nodes expresses the neighbourhood relation. Thus, when connecting two nodes, the semantic link represents a "followed by" relation.

Using the method as described, a visual language may be designed which consists of ready-to-use components. If a valid sentence of the language is totally constructed by composition, the visual language may be characterised as a visual language with black-box nodes. If a well-formed sentence may not be totally constructed by composition, but uses specialisation, it is a visual language with white-box nodes.

4.1.5.2 Implementation and Representation of a Graph Node

After defining the graph and the linkage to textual languages in a more formal way, it is necessary to describe how a node and the relationships between the nodes may be implemented and, more interestingly, visually represented. There are certainly many different ideas on design and implementation. However, defining and specifying a "nice looking" graph is one thing, implementing it is another. As Brad Myers states in [MYERS-1990] the task of implementing a visual programming language is much harder than implementing textual languages. This section describes the layout implemented in a working prototype system.

As described already, a node is represented by a processor symbol and defined by a code file. In general, a node may be regarded as an abstract object definition of data and/or code. Another view would be that it is analogous to a non-terminal or terminal in a formal language definition as described in the previous section. By "applying" a processor, a production rule is carried out and the terminals, linked to the processor, are used to define the processor template code. In any case, a processor node is meant to represent a part of a well-formed language construct with its inherited syntax and semantics. The reader should, however, be aware that the actual meaning of the language construct is less important than how it may be parameterised and placed at the right location of the application framework within the transformation process.
In order to follow an open concept, the *node modelling language* contains many different options. Table 13 gives a sketch of the most important entries defined for each lexeme. It clearly shows the separation of the semantics from its visual representation (see also appendix C).

Further features may be implemented in addition to the node modelling language. For example, to allow blocking, a group of processors may be set into a processor folder (similar to a folder editor). This is a logical grouping not specified by the underlying syntax.

In general, it should be possible for an experienced user to change the pictorial symbol of the processor. This is achieved by introducing the *bitmap* attribute which contains a reference to a bitmap file.

As for other development tools, a syntax checker is required. This is achieved by a component called *graph checker* which evaluates the nodes at run-time. The representation of a visible processor should indicate whether the processor is fully specified (by all *must* parameters) and syntactically correct. This could be implemented by an alternative processor node (see *bitmap_err*) or by other visual means (e.g. use of colour). Thus, a special error element may be defined. Further, visual feedback is needed to inform the user that the program may be generated and executed. Appendices D through G contain further details for the interested reader. In the next sections, further ideas, which may lead to more complete VOOPEs, are described.
<table>
<thead>
<tr>
<th>lexeme</th>
<th>Domain restrictions/constraints.</th>
</tr>
</thead>
<tbody>
<tr>
<td>title</td>
<td>Title of an object node.</td>
</tr>
<tr>
<td>bitmap</td>
<td>Visual representation of a node (bitmap file).</td>
</tr>
<tr>
<td>bitmap_err</td>
<td>Visual representation when syntax error (bitmap file).</td>
</tr>
<tr>
<td>pin_descr</td>
<td>General slot definition of a processor element.</td>
</tr>
<tr>
<td>help</td>
<td>Reference to help file.</td>
</tr>
<tr>
<td>lexeme</td>
<td>Lexeme, syntactical name.</td>
</tr>
<tr>
<td>group</td>
<td>Group to which lexeme belongs.</td>
</tr>
<tr>
<td>next_lexeme</td>
<td>Possible successor nodes. These entries refer to traditional adjacency lists.</td>
</tr>
<tr>
<td>layer</td>
<td>Layer information - with number in [0, max_num]. Using the layer attribute it is possible to define a root processor or to specify other restrictions. Layer 0 is the topmost layer.</td>
</tr>
<tr>
<td>max_count</td>
<td>Specification of how often an element may be placed on the workplace.</td>
</tr>
<tr>
<td>error_msg</td>
<td>The message which should appear if an error is detected.</td>
</tr>
<tr>
<td>block</td>
<td>Restriction whether element may be &quot;sub-levelled&quot;.</td>
</tr>
<tr>
<td>type</td>
<td>Type of an element. Intended for future use if other types of nodes (e.g. composite nodes) are necessary.</td>
</tr>
<tr>
<td>count</td>
<td>Number of slots in total.</td>
</tr>
<tr>
<td>slot&lt;i&gt;</td>
<td>Specification of slot attributes analysed at run-time.</td>
</tr>
<tr>
<td>slot_direction</td>
<td>(&quot;in&quot;</td>
</tr>
<tr>
<td>slot_type</td>
<td>(&quot;event&quot;</td>
</tr>
<tr>
<td>spec_option</td>
<td>(&quot;must&quot;, &quot;optional&quot;)</td>
</tr>
<tr>
<td>connection&lt;i&gt;</td>
<td>Specification of possible (typed) connections.</td>
</tr>
<tr>
<td>connection</td>
<td>(type1 &quot;-&gt;&quot; type2</td>
</tr>
</tbody>
</table>

Table 13. The basic node modelling language of VMB.
4.2 Approach 2: Extending UML for Dialog Processing

4.2.1 Background and Motivation

The second approach focuses on the definition of dialog-oriented applications. The idea is to create an alternative way of specifying the dynamic behaviour of these kinds of application. Current object-oriented tools such as MS Visual C++, JBuilder or VisualCafé are very efficient in designing a GUI, however the coding of the event-driven control of the different screens is less obvious. In addition, UML is rarely used in this process. This approach tries to investigate a way which makes use of both visual modelling and visual programming techniques. The intended VPL should be extended to include dialog processors with pre- and post-dialog elements. In addition, the control and data flow within one dialog form should also be supported by processors. Finally, the definition of help texts (linked to either the entire dialog form or to a control element) should be possible. The overall idea has been verified by a small example.

4.2.2 Basic Idea

The original processor idea as outlined in section 4.1 was extended to contain so-called parameter windows and further control elements (e.g. for condition and order). First an application processor is used to represent the entry point of an application. Compared to VIPR more advanced features are considered (e.g. application arguments, the application icon and a logging file may be specified). This block processor is then refined by declaration, dialog and event processors. The following two figures show the general application structure.
Figure 22. Sample application processor.
In order to support dialog processing it is necessary to define specific “control element” processors. These elements represent elements on the screen as conventionally used by GUI tools. To maintain consistency with the definition of application code, a similar processor notation was used. The general idea is related to state transition diagrams. However, the use of more powerful nodes is an attempt to

Figure 23. Overall structure of an application.
reduce the number of states and transitions. The VOOPL editor is intended to be used to define the pre- and post-processing rules. Furthermore, the dialog field has to be assigned to dialog classes. The following figures 24 through 26 depict the general idea.

**Figure 24.** Definition of the control/data flow.
Figure 25. Example of the definition of a code block (sub-processors).

Figure 26. General architecture of form specifications.
4.2.3 Additional UML diagrams

Instead of inventing a new OO modelling language, the way in which UML supports dialog-oriented applications was investigated. A standard way to use UML is to model window and control elements by using classes and defining specific stereotypes. A class diagram representing a dialog may contain a “window” class and use an “aggregates” association for the control elements. The dynamics of user interactions may be shown by sequence diagrams. No other specific elements exist to further specify a dialog. Thus, a shortcoming of UML is the lack of direct support for dialogs.

In the second approach, the author thought about the creation of new UML diagram types to include the visual modelling of dialogs. The following figure 27 shows one new diagram type which is called an “actor-form” diagram. Using this diagram type it is possible to define a form and to specify which user (i.e. role of actor) may access which form.

![Actor-Form diagram](image)

**Figure 27.** Actor-Form diagram.

One benefit of using actor-form diagrams is that complex user concepts may be expressed. For instance, the actor named “user” may access form x, y and z whereas the actor named “admin” may only access forms “y” and “a”. In addition, as a specific symbol is used, properties such as the dialog type or the GUI editor to be used may be explicitly defined and linked to the symbol.
By double-clicking on the form element the corresponding form can be opened and edited. This is shown in the next figure.

![Diagram]

**Figure 28.** Designing the form using control element processors.

In the requirements phase, use cases are intensively used to capture the domain specific aspects. It is important to link the forms to the corresponding use cases. For this purpose, an additional diagram type called the "form-use case" diagram is introduced (see figure 29). Again, by double-clicking on the form element the corresponding form can be opened and edited. By introducing this new diagram type it is possible to keep track of the use cases that contain dialog elements. Of course, it would be possible to combine actor-form diagrams with form-use case diagrams into one single diagram. But this is more a matter of preference.

![Diagram]

**Figure 29.** Form-Use case diagram.
Use cases are then refined by class diagrams which contain the domain objects. In addition, the forms may be associated within activity diagrams as provided by the business process link from Ensemble Systems (see appendix L). This helps to find the relevant domain classes. At this point, forms with control elements and associated classes are defined. Since a control element within a form has to be linked to a domain object in order to process its content the corresponding class definition has to be linked with the dialog description file. Furthermore, the panel buttons and menu items have to be linked to appropriate callback functions.

The required mapping is illustrated in the next figure.

The definition of the classes and their methods takes place within a UML tool. The implementation of the class methods could have been done with a specialised visual programming editor. The following figure 31 gives a sketch of how to code the add method within a fictitious Calculation object.

4.2.4 Dynamic Modelling

One problem identified in the first approach was the lack of efficient navigation techniques and the means to help the developer understand the overall application structure. Therefore, an adequate visual representation of dialog-oriented applications has been searched for. One idea for the dynamic modelling of the application was to make use of pre-defined VOODE classes. In this way, the visual system would contain
a set of architecture or framework classes, for instance, based on MFC or IlogViews classes.

Within the second approach, two different possibilities have been identified to model the dynamics of an application. One way is to make use of pre-defined application structures whilst another is to take statecharts. The two possible approaches which would be supported by the visual system are outlined in the following sections.

4.2.4.1 *Pre-defined Application Structures*

The first possibility for the dynamic modelling of the application takes into account that the general application structure is known and only high level elements have to be presented to the user. The construction of the specific application is then guided by the visual editor provided. The following figure shows the initial idea which allows the specification of panels and associated data which have to be kept consistent with the UML model.
The concrete translation into a real framework is highly dependent on the framework and tool chosen. As IlogViews [ILOG-1997] has been judged to be an effective (i.e. high-levelled and abstract) basic architecture two general application types have been identified which are presented in the following two figures.

**Figure 32.** Visual representation of a dialog-oriented application (static view).
The first figure shows the relationships of a single panel application based on IlvDialog (i.e. special gadget container with top window/view).

![Diagram of the application relationships]

Figure 33. Single panel application based on IlvDialog.
The second figure shows the relationships of a single panel application based on IlvApplication and IlvGadgetContainer (i.e. gadget container with top window/view). This architecture may be easily expanded to include more than one panel.

Figure 34. Single panel application based on IlvApplication.
It should be noted that the exact mapping between the visual description in the editor and the IlogViews architecture was not developed any further. A better idea because it is less IlogViews specific seemed to be based on statecharts and is outlined in the following sections.

4.2.4.2 Statecharts For Structuring an Application

UML provides state transition diagrams to capture the states of a single object and to show the events and messages that cause a transition from one state to another. They are created only for classes with significant dynamic behaviour and are not intended to be used for the definition of an application structure or the specification of different application states. The author tried to develop a method that allows the use of statecharts as a framework for visual programming. On the one hand statecharts are in themselves visual whilst on the other they may be used as a generic tool. Different statechart diagrams were created and verified by an MFC-based C++ application.

Though there are a number of different statecharts described in the literature (e.g. [HAREL-1988]) the notation used here is similar to that of UML. A state is represented by a rectangle and contains actions within the state. There is no need for special states such as a start or stop state as predefined state names define this already. UML makes use of a transition specification that consists of the event name, an optional condition and the actions to be carried out. In UML the keyword 'do' precedes activities associated with a state that normally persist whilst an object is in that state. The keywords 'entry' and 'exit' refer to entry and exit events for a state and precede actions that are triggered by entry or exit events.

The following examples (see figures 35 through 38) are intended to clarify the notation which could have been used by the VOODE system. Figure 35 shows the main states of the application. Each box represents a code part of the underlying application framework. It contains two parts: the upper part provides the logical name to the state, the lower part includes a list of action prefaced by the keyword "do". The states correspond directly to classes within the framework. Actions represent methods of theses classes. The coding would be done by the use of a visual processor-based editor as depicted in figure 41. The arrows used show the transition from one state to the other. The named links define the identifiers to be used for the mapping of events and states. This means, for instance, that the state "Application help" should handle either "ID_Help" and "ID_Help(CCalculatorDlg)". The supplement "/WinHelp" forces the
use of the external WinHelp.exe application. One can conclude from this explanation that the statecharts provide an abstract visual view of the (changeable resp. configurable) application parts.

Figure 23 shows a refinement of state "Start of application". In particular, the statechart contains the state "Construction – active object", which contains two actions in order to define the behaviour when the application is instantiated and in addition when the application object is set active. Figure 24 is a refined statechart of the state "Dialog". It contains a "block state" named "Dialog processing" which is responsible for the message execution (see figure 25).

*Statechart “Application”*

![Statechart “Application”](image)

*Figure 35. A sample statechart “Application”.*
**Statechart “CWinApp”**

- **Start of application**
  - ID_HELP[Message Map]
  - INSTANTIATION[Default]

- **Construction - active object**
  - do: CCalculatorApp::CCalculatorApp
  - do: CCalculatorApp::InitInstance

- **End of application**
  - do: CCalculatorApp::CCalculatorApp

*Figure 36. A sample statechart “CWinApp”.*

**Statechart “CCalculatorDlg”**

- **Start of application**
  - do: CCalculatorApp::InitInstance

- **Construction - active object**
  - do: CCalculatorDlg::OnInitDialog
  - do: CCalculatorDlg::InitInstance

- **Dialog processing**

*Figure 37. A sample statechart “CCalculatorDlg”.*
Statechart “Dialog Processing”

![Statechart Diagram]

Figure 38. A sample statechart “Dialog Processing”.

Within VOODE, different pre-defined “state frameworks” should be selectable. These could be based on MFC or any other appropriate application framework. Therefore, it would not be possible to add further main states. However, a state may be further refined by sub-states. Therefore, as pre-defined slots exist, these diagrams are called “slot diagrams” within the VOODE. The activities within a slot-chart are coded through the VOOPPL editor. In order to refine the sub-states new domain classes (including attributes and methods) may be defined to extend the existing global object model. After all class methods and callback functions have been defined, the slot-charts could be refined.

When a new “VOODE single dialogue framework” is selected, the principle structure of the slot-charts is provided to the user. Each predefined class is linked to at least one special category. A state represents a code part within the application. There are no transitions modelled because this would not be helpful for understanding the application’s controlflow. Furthermore, the concept would be less flexible because, for instance, different slot types (e.g. code, event, etc.) may not be defined.
A possible integration within the visual environment is illustrated in figures 39 and 40.

**Figure 39.** VOODE framework and sample "slot charts".

**Figure 40.** Defining the action "load calculation object".
The following figure gives an impression of the definition of the \textit{AddButtonCB()} function.

\textbf{Method definition: VOODE\_Container::AddButtonCB()}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{method-definition}
\caption{Defining the method \texttt{VOODE\_Container::AddButtonCB()}.}
\end{figure}
Furthermore, the mapping from slots to VPL code is straightforward as the reader may see from the following figure. The slot diagram file is intended to describe the general framework and the included slots.

![Slot diagrams]

Category0="main"
Category1="VOODE_Applic"
Category2="VOODE_Cont"

[VOODE_Cont]
S0=s1.sf
S1=ss.sf

Slot Diag. File: VOODE.sdf

---

**Figure 42.** Description files needed to maintain the definition links.

There are two general basic techniques to be used for framework construction. The first one is based on inheritance. The idea here is to override existing class methods (i.e. implementation methods). This is directly supported by all important OO languages. The following figure gives a C++ example.

```c++
class VOODE_Display : public IlvDisplay
{
    VOODE_Display();
    virtual VOODE_Display();
    ...
}

VOODE_Display::VOODE_Display() : IlvDisplay(PROGNAME, PROGNAME)
{
...
}
```

**Figure 43.** Using inheritance with a base class.
Another technique is based on delegation. The basic idea here is to encapsulate the service functions and to provide a new set of service functions. The constructor is used to create a link to the service object. The new service functions may make use of the object's interface.

The following figure gives a C++ example.

```
class VOODE_Dispaly
{
    VOODE_Dispaly() {_display = new IlvDisplay(PROGNAME, PROGNAME);} 
    virtual ~VOODE_Dispaly() {delete _display;}
    ...
    IlvDisplay * _display;
}
```

**Figure 44.** Using delegation to use a base class.

The advantage of using framework classes are as follows:

- Existing and well documented classes are used.
- New classes may be defined (e.g. based on existing framework classes).
- The support and use of design patterns is enabled.

In summary, it can be concluded that the framework approach is a very powerful approach.
4.2.5 Development Environment and VOODE models

The general idea of the second approach is to use known and effective OO techniques and extend them so that the functional modelling is done by a visual object-oriented programming language. The visual language can be quite simple and the various models may be translated directly into C++ code (or any other appropriate language). The generated code may be used directly to build the application. Any development environment (such as MS Developer Studio) supporting C++ development may be used as the development environment. The following figure depicts the different components of the proposed system.

![Diagram showing components of the visual object-oriented development system]

**Figure 45.** The complete visual object-oriented development system.
Several description files (e.g. flat description files) may be used to represent the various features. The following figure shows the general layout and the separation of functionality from internal representation.

![Diagram showing VOODE components]

**Figure 46.** Necessary VOODE components.

The big advantage of the second approach is that the application is completely defined with visual elements. Therefore, textual coding is no longer necessary and the visual system is a complete visual object-oriented development environment. As this system will have knowledge of the underlying software architecture (e.g. the VOODE classes) the application developer could be supported better than is currently the case with existing CASE tools.

The architecture of an application is critical to the reduction of complexity and thus the reduction of errors. To help create a stable basis the proposed visual system makes use of an architecture model which may be based on an effective framework (e.g. MFC) and/or a class library (e.g. IIlogViews). Therefore, it is predictable that such a system would be an improvement. As the framework approach seems to be very appealing the author investigated the possibility of refining this idea.
4.3 Approach 3: Framework-based Development

4.3.1 Background and Motivation

The third approach focuses on visual framework-based development. The overall idea is as described in the second approach the usage of a pre-defined object-oriented application framework and a VOOP. The parts of the framework which may be refined by the user are called slots. A slot defines a code part which is directly linked to the framework code. Slots which may be logically organised in categories have a meaning and define the actions which are carried out in this state of the application. This concept is more generic and simpler in use than state transition diagrams. Furthermore, there is a direct visualisation of a framework and the overall application structure.

Within this approach, there are two basic places where visual programming may replace string or textual coding. First of all, the classes may be defined using a special VOOP. In particular, the service functions may be completely coded using these techniques. In addition, the actions which are defined by "slot-charts" may also be visually coded. The lexemes of the new VOOP would be based on icons as in the previously described approaches.

4.3.2 Basic Idea

The basic idea of this approach is the definition of a platform-independent visual representation of an underlying application framework. The visual representation should not be fixed and may include images or other types of diagrams which are conventionally used in real-word projects in order to show the key system/application components. The work of this approach focuses on visual units called slots which are linked to these high-level views.

Figure 47 shows how a simple standard framework, together with predefined slots, could be visually represented. The use of slots may be thought of as being a step towards components, or component-based software parts. If supported by VOODE, a complete slot may be dragged & dropped from a library slot (see figure 47). If interface
methods are not compatible, then adapter code may be generated. This could also be supported by visual means.

**Figure 47.** The layout of a standard framework.

The slots correspond directly to classes and their methods. The definition of a class method would be carried out using a VOOPL editor. The following two figures show two early sketches.

**Figure 48.** Defining the method Calculation::Add().
One key issue of this approach is to define how visual code can be linked to the slots as well as defining the ways in which the slots can be linked to the predefined framework. An existing framework has to be described by a framework definition file. Here the "public parts" (i.e. public classes and methods) have to be specified. Then these parts have to be mapped to slot representations. These in turn may be refined and specified by sub-slot definitions. The complete slot definitions together with the VOOPL configuration files would then be used for the visual editor. Figure 50 illustrates the general idea.

The pre-requisite for this approach is a generic framework definition language that allows the description of an application framework and the defined slots and which may be refined by visual/textual code. This should take into account the fact that different slot types may exist. The following figure 51 shows an example for a function slot type. This corresponds directly to a VOODE class. Another type would be the main slot. This slot is a predefined code part which represents the main entry point of any application (see figure 52).
Figure 50. How visual code may be linked to a framework slot.
Apoication ýhuctwe
AvIlb, VOODE VOODE
Application Container
1 -90
Slotcategory JVOODE Container
Construction
Destruction
Category Slot Action new delete
+ F1: main
+ F2: VOODE_Application
+ F3: VOODE_Container
+ S1: Construction
S2: Destruction
A1: IvPrint
A2: new_display
A3: new_container
A4: load and show panel

Action [S1, A3]: load calculation object F3_S1_A3.vpl

Figure 51. Sample function slot.
Figure 52. Sample main slot.
Within this approach, the author has improved the processor paradigm. This has been done using a bottom-up approach i.e. looking at concrete code examples and then representing the code in the equivalent visual code. The following figure shows the code for the application constructor method. It shows the use of tags in order to locate the place where the visual code is to be included. The tags should be automatically generated by the system.

```c
// Constructor
//@VOODE:VOODE_Applic
VOODE_Application:VOODE_Application()
{
  ifdef NDEBUG_ALL
  l1vPrint("Construct VOODE_Application");
  #endif
  //VOODE_end:A0
  //VOODE:A1
  VOODE_Display *display = new VOODE_Display();
  //VOODE_end:A1
  //VOODE:A2
  VOODE_Container *container = new VOODE_Container(this, display, 10, 10, 200, 400);
  //VOODE_end:A2
  //VOODE:A3
  container->show();
  //VOODE_end:A3
} /* VOODE_Application::VOODE_Application */
//@VOODE_end:VOODE_Application
```

**Figure 53.** Sample textual code.

The visual equivalent is shown in the next figure.

**Figure 54.** Corresponding visual code.
One general improvement over the previous approaches is the idea that predefined textual code is not visible. This would significantly reduce the number of visual elements required. Furthermore, pre-defined parameters and object instances would be selectable via the so-called "data linkage section" (see figure 55). The following figure shows an example.

```c
#ifndef _DEBUG_ALL
#define _DEBUG_ALL

VOODE_Application:: VOODE_Application()
{
    #ifdef _DEBUG_ALL
    tlvPrint("Construct VOODE_Application");
    #endif
    VOODE_Display * _display = new VOODE_Display();
    VOODE_Container * _container = new VOODE_Container(this, _display, 10, 10, 200, 400);
    // @VOODE:VOODE_Appl
    _container->show();
    } /* VOODE_Application::VOODE_Application */
```

**Figure 55.** Representing pre-defined code parts and objects.

### 4.3.2.1 Design Aspects of the VOOPL-1 Editor

For the proposed extensions to be successful, it is very important to achieve a smooth integration of the VOOPL editor into an existing CASE environment. This means that the dependencies and relationships between the classes and components have to be kept consistent. Thus, the structure which is used in the editor has to be kept consistent with the current design model. The visual editor may be called in many different ways, for instance, from the class diagrams or sequence diagrams.
Figure 56 shows the different possible links. Depending on this, the selectable elements may be restricted.

![Diagram of different links and their relationships to the VOOPL-1 editor]

**Figure 56.** Navigational links to the VOOPL-1 editor.

Figure 57 gives a sketch of the VOOPL editor which could have been implemented. It consists mainly of two sub-canvases. The upper part is the navigation area where the user may select a component type and a method of one of the different assigned classes. The lower part contains the links to other reachable objects, and provides the means to write visual code. The selected class method is coded on the workplace in the middle of the lower canvas.

![Sketch of VOOPL editor with class and method details]

**Figure 57.** Sketch of the VOOPL-1 editor.

The third approach was considered to be very useful. Therefore, a very simple dynamic prototype was developed. Appendix K shows some sketches of a possible user interface. However, there are also some problems. First of all, the invention of a
generic framework description language seems to be very difficult. Therefore, the author searched for a different approach. As the slots represent application parts or components of an application, the author wanted to keep this idea. After reading more about UML and CORBA the idea to try a combination of VP, UML and CORBA was generated. The result of this final approach is described in 4.4 and 5.
4.4 Approach 4: Component-based Development

4.4.1 Background and Motivation

As modern applications become more sophisticated, their development increases in complexity and the time required for completion. 'Component Software' [SZYPERSKI-1998] is an important and emerging area in the software field that facilitates a more rapid production of robust programs. It describes the aim of application/system developers to create software systems that are built from individual components. Components are nontrivial software parts which conform to and provide a set of interfaces. There are several component technologies of current importance. In the fourth and final approach, the author focuses on the design of fine and large grained CORBA objects or components [OMG-2000]. In this context, a component is a reusable software unit with an implementation and a set of separate interfaces which a client may use in order to request the services provided. Of importance is to investigate how visual modelling can be combined with visual programming for component implementation. This approach has been inspired by the previous approaches and tries to include some of their ideas.

4.4.2 What Can Be Improved?

As described in section 2.3.2, the task of designing and implementing CORBA components is fairly difficult and time consuming. To apply CORBA technology requires knowledgeable application developers and architects. In addition, since several tools and techniques have to be used, the learning curve of inexperienced developers is quite steep. The general question that arises is: "how can this process be facilitated?".

First of all, an application developer has to use several special tools (e.g. CASE tool, IDE, ORB product) and various languages (e.g. UML, IDL, C++, Java). This requires a broad skill set. Furthermore, there are inherent mapping problems (e.g. between UML and IDL, or between IDL and C++) which complicate the design and code generation. Thus, it is fairly difficult to decide what should be modelled and how. So, can visual systems and languages help to avoid such problems? Answering this question is the focus of the fourth approach.
None of the available commercial tools for CORBA component development enable the usage of a VPL (see section 2.3.2). This is somewhat surprising since visual programming (VP) can greatly enhance the strengths of proven techniques (such as object-oriented programming and modelling). This is especially true when the design model is very precisely defined so that the generated and refined code can be closely tied to the design model and round-trip engineering techniques may be used. The key idea of this baseline model is that "the system is the model" and "the design appears to be executable".

One problem of many visual systems is the lack of support for a wider problem domain, which is particularly necessary when (in an enterprise computing environment) CORBA components have to be developed. In addition, most visual systems are not seamlessly integrated in a modelling environment. As far as the author is aware, there is currently no specific visual system that incorporates a VPL for CORBA development. Furthermore, today's CASE tools support most parts of the currently defined UML [UML-2000] and provide many code generation facilities for all major textual languages such as C++, Java, and Visual Basic.

4.4.3 Basic Idea

The first approach makes use of a tool called Visual Meta Builder (VMB) which allows a language inventor to define a visual programming language by transforming any context-free language into the underlying node modelling language supported by the VMB tool. The idea of the newly proposed system called VOODE/VOOPL-1 (Visual Object-Oriented Development Environment/Visual Object-Oriented Processor Language version 1) is to develop the Visual Meta Builder concept further and to link the visual programming language to a design model that was developed with an existing CASE tool such as Rational Rose [RATIONAL-2000].

The intended VOODE/VOOPL-1 prototype should provide a means to create CORBA-based server components using mainly visual techniques and should prove the overall concept. Furthermore, it should integrate with UML/Rational Rose and other necessary tools. Therefore, a combination of Visual Modelling with Visual Programming will be achieved. The overall aim is to demonstrate the basic concepts of a visual system that reduces the complexity and therefore decreases the time needed to create a CORBA server component. Generally, the time needed to learn C++/Java and
CORBA for novices is several months. The new system should decrease this learning period to a few weeks in order to be judged successful. The overall system should hide the underlying technology (i.e. CORBA) as much as possible so that the user may concentrate on writing business logic.

4.5 Critical Discussion of the Different Approaches

After the introduction of the four approaches the advantages and disadvantages of the different concepts are discussed.

The first approach addresses the problem of defining a visual programming language. The VMB tool based on a specific node modelling language enables the creation of simple visual processor programs. One shortcoming of this approach is that the visual environment is not intended to be a real programming environment and thus, the interaction techniques are very basic. This concept may be used to directly compare a visual programming language with a string language as the transformation process from a textual to a visual language is fairly flexible and powerful. One drawback is that the overall structure of the application is hardly visible and more powerful navigation techniques would be needed. Furthermore, the visual feedback is not sufficient. This work may therefore be seen as the ground-work for the subsequent approaches.

The second approach expands the initial idea towards the construction of complete dialog-oriented applications. One advantage of this approach is that it makes use of UML to model the architecture of the application. However, as UML is judged to be insufficiently powerful, new diagram types are introduced. Thus, it is a specific solution. Though the second approach is very generic statecharts are not thought to be the ideal model. One severe drawback is, that for large applications the overall structure of the application gets lost, due to the number of different charts (cf. “scaling-up problem” [BURNETT-1995]).

Therefore, another visual representation is needed. The third approach includes the idea of application frameworks and incorporates an alternative model to statecharts (called slot diagrams). A slot defines a code part which is directly linked to a special place within a given framework and has a meaning. It defines the actions which are carried out in this state of the application. In addition, slots may be grouped into logical categories. Unlike statecharts as used in the second approach, no transitions and link
names are necessary. Slot charts are thus more generic and simpler to use than state transition diagrams. Furthermore, there is a direct visualisation of a framework and the overall application structure. The visual code linked to a slot may be based on the same processor idea. One problem of this approach is the definition of a generic meta framework language. The investigation of this would however lead to another research field.

The final approach concentrates on component-oriented development. It reduces and focuses the problem of the third approach towards components i.e. software units with clear interface methods. The advantage is that the internal specification of a component is hidden from the client component. Thus, it doesn't matter whether a visual programming language is used for its implementation. This approach also makes use of UML as in the second approach. Standard extension mechanisms of UML (e.g. use of stereotypes and export control attributes) may be used. The final approach makes use of different degrees of abstractions to represent the intended computation. The UML model provides the "big picture" whereas the visual implementation model concentrates on low-level details.

In conclusion, the final approach has the highest potential with respect to the research goals since it naturally combines visual modelling with visual programming. It also follows the trend in the software industry towards component-based development and extends existing concepts and techniques. As a result, the author has focused his research on the last approach. The findings are described in more detail in chapters 5 and 6.

4.6 Justification For the Approach to Combine Visual Modelling With Visual Programming

It is important to justify why combining OO concepts with VP techniques will yield a better (i.e. more productive and easier-to-use) system. This is done by a short characterisation of the two basic paradigms and a description of some obvious links between them.

Within the object-oriented methodology, objects are well-defined system parts with well-defined interfaces. Behaviour is caused by receiving/sending messages from/to other objects. In the area of visual programming environments, graphical representations and structures are also well-defined objects, and appear in the form of
icons or pictorial images. In most cases, interaction is caused by associated links which could also be understood as applying an object's method. Furthermore, visual programming considers data abstraction and encapsulation alike. Icons may hide complexity and make relationships or dependencies more explicit. Direct manipulation of icons has proven to be very efficient and is inherently linked to a visual programming system.

Therefore, it seems to be possible to combine the strengths of both techniques. This is even more true for component-based software development. The Unified Modelling Language (UML) is a language used to specify, visualise and document object-oriented systems and now plays an important and accepted role in the IT market (UML is the *de facto* standard). As Terry Quatrani [QUATRANI-1998] states "UML is an attempt to standardise the artefacts of analysis and design: semantic models, syntactic notation, and diagrams". *Visual modelling* is a way of thinking about problems using diagrams which model real-world objects. It helps understand the problem domain and supports the iterative and incremental development life cycle of object-oriented systems. However, it should be noted that UML is designed to be used only for analysis and design. Thus, it is independent from implementation details, such as the underlying programming language (e.g. C++, Java, Smalltalk) and the hardware platform (e.g. PC, workstation, host) etc. To extend visual modelling towards visual programming seems "natural" since it avoids a methodological gap. In chapter 5 the concepts and the prototype system of the final approach are described in more detail. The general conclusion formed during the development of the different approaches is that it is difficult to predict whether an approach is really useful. Therefore usability studies are always necessary. In chapter 6 the results of usability trials for the final approach are discussed further.
5 VOODE/VOOPL-1 for CORBA

5.1 Objectives

In this chapter, the final approach is described in more detail. Due to space limitations only the main points are described. The objectives of this approach are to investigate to what extent it is possible to combine visual programming with visual modelling and to study the tradeoffs for a VOOPL. This approach has been evaluated by usability trials which are discussed in chapter 6.

5.2 Introduction and Motivation

Component-based development (CBD) is an emerging area in computer science. One general problem of CBD is the visualisation of components, their behaviour and their inter-dependencies. In order to support an integrated way of designing, implementing and maintaining components, one could demand a complete visual system that makes use of both visual modelling and visual programming techniques. However, no research has taken place to investigate how a visual programming language (VPL) could be combined with a visual modelling language (like UML) and be integrated with an existing CASE tool. The final approach uses a system called VOODE/VOOPL-1 (Visual Object-Oriented Development Environment/Visual Object-Oriented Processor Language version 1) which tries to combine visual modelling with visual programming for CORBA component development. The following sections present the current status of the intended system and describe the overall underlying concepts. The emphasis lies primarily in the visualisation of structures and design issues rather than on implementation details. The aim of the prototype which was to show the benefits of the proposed system. Due to the amount of work which would be necessary to implement all intended features the prototype system only demonstrates some core parts which are sufficient for the intended evaluations (see chapter 6).
5.3 Design Aspects of VOODE/VOOPL-1

When designing a visual system the designer has to answer several questions. One important question is how the visual interaction model should look and how the various aspects of the intended system could be mapped to the corresponding textual interaction model. This is necessary since no computer system is able to directly execute visual code. In the case of VOODE/VOOPL-1 the basic visual metaphor is that of an iconic system and a visual grammar based on the processor idea (see section 4.1). The design was guided by the rule that UML should be used to model the overall component architecture and visual programming should be used for component implementation. This is guided by Tufte's findings that good graphics (and this clearly includes diagrammatic/visual languages) should have both macro reading (i.e. global, or summary, information available at a glance) and a micro reading (i.e. local, or detailed reading, information available on closer reading) [TUFTE-1990].

As described in section 2.3.2 the difficulty with CORBA development is often two-fold: understanding the concept and learning how to apply it. This concept, therefore, reduces complexity by supporting the basic concept of having components with service methods (i.e. interfaces) described by the means of UML and hiding complex CORBA mapping mechanisms through a transparent interface/method relationship. This clear separation of the different roles of UML and VPL will make the concepts more transparent to the user. In addition, this approach follows to some extent the intention of generative techniques for Component-Based Software Engineering (CBSE), viz. that custom components can be produced from elements that can be considerably simpler than executable components themselves, thus keeping the complexity of an evolving component system under control [NOD-2000].

Within the fourth approach, the visual grammar corresponds directly to the underlying textual (or string) grammar. This means that the iconic sentences which may be coded within VOODE/VOOPL-1 are translated to the textual representation of the target language (e.g. C++). This strategy is derived from studies carried out by Baroth and Hartsough [BAROTH-1995]. They found productivity benefits in the development of test and measurement systems when using LabView and VEE due to the clear relationship between the visual syntax used and the users' experiences on circuit diagrams.
Furthermore, the icons make use of different colours. Tufte [TUFTE-1983] indicated that the effective use of colour can significantly enhance the information-carrying capacity of representations, although the poor use of colour can be distracting. In addition, small bitmaps are used for most processors to enforce the visual comprehension.

Also, compared to the first approach, several extensions to the visual editor have been necessary. It is quite important for a developer to get visual feedback on the correctness of the visual code since the visual code is not directly interpreted but transformed into the textual equivalent. This is known as the “liveness” degree [BURNETT-1995, p 11] and may include different concepts.

The following figure depicts the general idea of the different interaction models and the artefacts (i.e. UML model and icon-based implementation model) within VOODE/VOOPL-1.

![Diagram](image)

**Figure 58.** Mapping the Visual (Interaction) Model to the Textual (Interaction) Model.
5.3.1 Processor Metaphor and VOOPPL-1 syntax

The syntax of VOOPPL-1 is based on the same metaphor and node modelling language used within the first approach. A processor icon represents a more or less complex C++ statement whereas the connection of possible processor icons are controlled by slot (i.e. link point) descriptions. A processor may have different slot types (e.g. events, data) and contains attributes which specify the arguments of the action to be carried out. The following figure 59 shows a sample processor with two event and two data slots and the corresponding C++ statement (see appendix H for further sketches).

![Figure 59. A sample "Add" processor with three attributes.](image)

A method (or operation) within VOODE/VOOPPL-1 (modelled in Rational Rose) consists of many inter-connected processors placed on different layers of a graph. The processor nodes are defined by individual code files and transformed into their textual representation by scanning the graph and using their node description.

The different code files have been developed such that the various methods of a C++ class may be completely described through VOOPPL-1 processors and be used by the VOOPPL-1 editor. The following table describes informally the basic set of VOOPPL-1 processors ('$' denotes a processor attribute, actionList a network of connected processors) and shows that the mapping is straightforward.
It is important to note that processor attributes are, in most cases, available by drag&drop techniques from context-sensitive toolbars. Like the parameter helper in JBuilder [JBUILDER-2000], VOODE/VOOPL-1 could also make use of this kind of technology. Another interesting aspect is that identical processor attributes may be shared between several processors by connecting them to the corresponding data slots (see figure 60).

5.3.2 Design aspects of the VOOPL-1 editor and Its Integration In Rational Rose

The VOOPL-1 editor was designed to allow inexperienced or expert users to write processor code for a specified CORBA component. To achieve this, the user interface has been kept simple and reflects the idea of a processor and its associated attributes. The editor consists mainly of two sub-canvases.
Figure 60. Design of the Visual Editor.

A: menu  
B: VOOP-1 shortcuts  
C: navigation and context tree  
D: system variables, method parameters  
E: object association processors  
F: workplace for processor code  
G: language processors  
H: attribute processors

The left part (tree element) is the navigation area where the user may select the different sections of the processor tree. The right part contains the workplace and the available processors which are organised in processor categories. The processor icon may be dragged & dropped on the workplace in order to define the behaviour of a method. The lower part of the panel contains the attributes associated with a selected processor and the status line which gives important feedback to the developer.

Transforming or wrapping textual code in a visual manner is not sufficient to build a successful visual system. In order to support the developer more effectively, the visual
system should make use of pre-defined contexts that incorporate previous knowledge of programming issues.

Different contexts are defined within VOODE/VOOPL-1. A context or sub-context is closely related to a task which a developer has to carry out. One example is the context "CORBA Server" with its sub-contexts "OnStart", "OnServerInit", and "OnExit" for the definition of the server mainline. The following table provides an overview of the predefined contexts.

<table>
<thead>
<tr>
<th>Context::Sub-context</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORBA IDL::IDL File</td>
<td>Used to construct the CORBA module with its interfaces and definitions.</td>
</tr>
<tr>
<td>CORBA Server::OnStart</td>
<td>Definition of the initial behaviour when a CORBA server is started.</td>
</tr>
<tr>
<td>CORBA Server::OnServerInit</td>
<td>Creation of the CORBA objects including exception handling.</td>
</tr>
<tr>
<td>CORBA Server::OnExit</td>
<td>Definition of the behaviour upon server timeout or after a system exception.</td>
</tr>
<tr>
<td>CORBA Module::CORBA_IMPL</td>
<td>Implementation of the interface methods.</td>
</tr>
</tbody>
</table>

Table 15. Predefined implementation contexts.

The selectable processor types of an available language category are defined for each context. This ensures that only valid processors may be selected. For the visual system to be successful, it was thought to be very important to achieve the integration of the VOOPL-1 editor into an existing CASE environment. Since Rational Rose is a widely used CASE tool and provides both a script language and a COM interface it was considered to be open enough for the research goal.

5.3.3 Visual Feedback

An important aspect of visual programming systems is the way the user gets feedback about the correctness of the visual code.

Within VOODE the so-called "traffic light" metaphor is used. This means that a processor program containing processors which have been placed on the workplace could be in three different states. To provide the user with appropriate visual feedback, the state of a processor and its related context gets checked after each relevant alteration. For each
processor type three different bitmaps are defined. The visual representations and their meanings are shown for each context in following tables 16, 17.

<table>
<thead>
<tr>
<th>Tree view</th>
<th>Meaning</th>
<th>Result (post-condition)</th>
</tr>
</thead>
<tbody>
<tr>
<td>? red</td>
<td>Specification incomplete (i.e. links are missing or must declarations are incomplete).</td>
<td>No legal VOOPL-1 syntax/semantic.</td>
</tr>
<tr>
<td>amber</td>
<td>Syntax/configuration ok, but at least one processor is not connected or has missing links.</td>
<td>No legal VOOPL-1 syntax/semantic.</td>
</tr>
<tr>
<td>green</td>
<td>All processors are connected and syntactically correct.</td>
<td>Code generation is possible for this context.</td>
</tr>
</tbody>
</table>

**Table 16. Visual feedback coming from the tree view.**

<table>
<thead>
<tr>
<th>Processor on workplace</th>
<th>Meaning</th>
<th>Result (post-condition)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDL Module</td>
<td>red</td>
<td>Processor specification is incomplete (i.e. must attributes have not been assigned a value).</td>
</tr>
<tr>
<td>amber</td>
<td>All processors attributes have been specified and are syntactically correct. However, connections are missing.</td>
<td>Code generation is possible for this processor.</td>
</tr>
<tr>
<td>green</td>
<td>All processors attributes have been specified and are syntactically correct.</td>
<td>Code generation is possible for this processor.</td>
</tr>
</tbody>
</table>

**Table 17. Visual feedback coming from the workplace.**

---

5 Due to b/w printing, the usage of different colours is hardly discernible.
5.3.4 The VOODE/VOOPL-1 architecture / code generation

The VOODE/VOOPL-1 system contains different system components. The underlying architecture is shown in figure 61. The starting point is Rational Rose which is used to create the business component model. The definitions of the component model are transformed into a description file (.comp file) which functions as the input for the VOOPL-1 editor (see appendix I).

![Figure 61. The VOODE/VOOPL-1 architecture.](image)

The VOOPL-1 editor creates .voopl files for each context. These files contain the processor code and are referenced in the corresponding .comp file. The translator makes use of the model information stored in the .comp and the .voopl files to create the target code and a makefile.

The IDL files are generated from the processors of type "IDL". The relevant definitions are stored in the processor properties. The IDL compiler of Orbix is used to
create the CORBA skeleton code. Finally, the `nmake` utility from MS-VC++ is taken to create the executables. The translation step is shown in figure 62.

![Diagram of VOOPL-1 editor (IDL context)](image)

**Figure 62.** The translation step from processor code to IDL.

### 5.3.5 UML - VOOPL-1 Mapping

The mapping of UML elements to VOODE/VOOPL-1 representations proved to be straightforward. The following table describes the corresponding elements.

<table>
<thead>
<tr>
<th>UML element</th>
<th>VOOPL-1 representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>VOOPL-1 package in navigation tree</td>
</tr>
<tr>
<td>Class association</td>
<td>Object Processor in association toolbar</td>
</tr>
<tr>
<td>Class attribute</td>
<td>VOOPL-1 attribute in navigation tree</td>
</tr>
<tr>
<td>Class constructor</td>
<td>VOOPL-1 method OnConstruction</td>
</tr>
<tr>
<td>Class destructor</td>
<td>VOOPL-1 method OnDestruction</td>
</tr>
<tr>
<td>Class method</td>
<td>VOOPL-1 method in navigation tree</td>
</tr>
<tr>
<td>UML Data Types</td>
<td>VOOPL-1 data types</td>
</tr>
<tr>
<td>Inheritance relationship</td>
<td>Used to distinguish interface methods from class methods.</td>
</tr>
<tr>
<td>Method parameter</td>
<td>Processor parameter in system processors / parameter toolbar</td>
</tr>
</tbody>
</table>

**Table 18.** Overview of UML/VOOPL-1 mapping.
5.3.6 Working with VOODE/VOOPL-1

In this section, a small working example is provided. Screen images of the current prototype system are presented in order to present an initial overview. Due to space limitations, the explanations are kept to a minimum.

5.3.6.1 How to model a CORBA component?

The first step in designing a CORBA object or component is to start Rational Rose and to select the pre-defined CORBA component framework model as a starting point (see figure 63).

![Figure 63. The pre-defined CORBA component framework within Rose.](image)

This model contains all the necessary definitions and packages for a typical CORBA component which will be implemented later using the VOODE/VOOPL-1 editor. A CORBA component of the framework (see figure 64) consists of one interface class (with stereotype <<Interface>>), one implementation class (with stereotype <<Implementation>>) and one helper class (with stereotype <<HelperClass>>). The
CORBA server class is specified by using the stereotype <<CORBAServer>> and contains an instantiation relationship to the implementation class. The next step is to rename the existing class names and to add and refine the methods and attributes (not described here as there are no specifics). When the CORBA component is specified, the VOOPL-1 editor is started to create the IDL file, and to implement the methods as well as the behaviour of the CORBA server mainline.

![The CORBA component within Rose.](image)

**Figure 64.** The CORBA component within Rose.

### 5.3.6.2 The VOOPL-1 editor

The added entry "VOODE/ VOOPL-1 editor" under the Tools menu in Rational Rose (see next figure 65) may be used for component implementation. After the selection, a Rose script is started which creates a component description file (with extension .comp) for the editor. This file contains all the model information plus the links to the VOOPL-1 implementation files.

The editor provides a main menu together with the pre-defined contexts and component definitions (see figure 66). The navigation part on the left reflects the structure
of the underlying meta model and the specified component. To enter visual processor code, the user may select from the following global contexts: **IDL file, CORBA server mainline or CORBA component implementation.**
5.3.6.3 IDL file generation

The editor provides a more convenient way of modelling IDL, than from within UML/Rose which proved insufficient to requirements. It avoids the usage of specific stereotypes and reflects more the structure of IDL files (i.e. modules, interfaces and operations).

The IDL definition is achieved through processors. After selecting the IDL file context only the selectable processors for this context are visible. The available processors may be selected and dragged & dropped on the workplace. The processors may be connected via the event/data slots. Figure 67 shows a sample IDL definition.

![Processor-based visual IDL definition.](image)

5.3.6.4 Implementation of CORBA server mainline

Next, the server mainline may be coded. Here, the CORBA objects are instantiated. To help the developer implement the server, three sub-contexts are defined. Within the sub-context OnStart, the behaviour of the server at start-up time may be visually coded. Typically, a start-up message may be created here. The OnServerInit context is used to initialise the server object. Finally, the OnExit context is used to react when a timeout has
occurred, or in the event of a system error. A typical example of a server mainline program written in C++ is given in figure 68.

How may this kind of code be mapped to processors? This is fairly simple. The textual code is analysed and once a textual pattern has been detected the corresponding processor is defined. If additional attributes are to be used then special data processors (of type "System Variable") are introduced. The general mapping idea is shown in figure 69, and a concrete sample is given in figure 70. The reason why the processor code looks easier than the textual code is that the VOOPL-1 editor has knowledge about the context, and reduces the necessary definitions and the cognitive workload for a developer.

```c++
#include <iostream>
#include <stdlib.h>

int main()
{
    // create a grid object - using the implementation class grid_i
    grid_i myGrid(100,100);

    OnStart
        cout << "server started" << endl;
        try {
            // tell Orbix that we have completed the server's initialisation:
            CORBA::Impl_is_ready("grid_i");
            catch (CORBA::SystemException &sysEx) {
                cerr << "Unexpected system exception" << endl;
                exit(1);
            }
            catch (...) {
                // an error occurred calling impl_is_ready() - output the error.
                cout << "Unexpected exception" << endl;
                exit(1);
            }
        }

    OnServerInit
        cout << "<server started" << endl;
        try {
            // tell Orbix that we have completed the server's initialisation:
            CORBA::Impl_is_ready("grid_i");
            catch (CORBA::SystemException &sysEx) {
                cerr << "Unexpected system exception" << endl;
                exit(1);
            }
            catch (...) {
                // an error occurred calling impl_is_ready() - output the error.
                cout << "Unexpected exception" << endl;
                exit(1);
            }
        }

    OnExit
        // Impl_is_ready() returns only when Orbix times-out
        // an idle server (or an error occurs).
        cout << "server exiting" << endl;

    automatically generated!
return 0;
}
```

Figure 68. Typical server mainline code (C++).
Figure 69. Mapping typical server mainline code to processors.

Figure 70. Visual definition of CORBA Server mainline: OnServerInit.
5.3.6.5 Implementation of CORBA methods

Finally, the CORBA implementation methods are visually coded. There are a number of selectable VOOPL-1 processors such as Assign processor and Return processor which enable the user to define the behaviour of the method. A sample set operation is depicted in figure 71.

![Visual definition of a CORBA implementation method](image)

Figure 71. Visual definition of a CORBA implementation method.

5.3.6.6 Code Generation

After the visual coding has been completed by the developer, the relevant textual code is generated (see figure 72). This includes the resulting IDL file and the code specification files. A translator is then used to create the target C++ code (cf. figure 61). This build process is hidden from the user. After successful code generation the user is presented with a message box. The next step would be to create a client application in order to test the created CORBA component. However, this is not yet supported by the system.
5.3.7 Advantages of the Final Approach

The final approach focuses on the definition of a complete visual object-oriented programming system. The advantage of this approach is that complex CORBA concepts (e.g. the IDL mapping) are avoided (see section 2.3.2.2). Standard UML elements are used to outline the component. Furthermore, processor icons are used to implement the behaviour of the interface methods. The proposed language has many benefits. First of all, the underlying metaphor is very simple. All coding is supported by drag & drop mechanisms based on a “select-connect” paradigm. The user doesn't have to think about memory leaks or system limitations and most actions are easy to parameterise. A very useful feature is that object references are selectable according to the underlying UML model. This helps user to see which object is within the scope of a method. Though the author sees many benefits of the final approach, the usefulness of such a system has been evaluated by usability trials (see next chapter).
6 User Evaluations

6.1 Objectives for User Evaluations

Though the final approach of this research appears to be very attractive it is difficult to judge whether users would like the visual environment and what improvements have been achieved by this. Petre [PETRE-1992] argues that designers should not expect that the users would be using their visualisation tool in exactly the context they had planned. (Therefore, the tool should be as flexible as possible.) Lattu [LATTU-2000] noticed the effect that the use of animation seemed to motivate students. Thus, does a complete visual environment for program development also motivate students to learn (visual) programming? Lattu states that creating visualisation is not a trivial task as a successful visualisation depends on cultural aspects (i.e. personal preferences, course habits and wider society-level symbol languages). As a result: usability studies are always necessary because nobody can predict the performance of users or prove the claim that one system is better than another.

Before usability studies for this research can be carried out several important questions have to be answered. The central question is whether VL research is an independent research area and whether results of other fields are applicable to it? Other questions are "what kind of usability studies are applicable and what can be measured" and "how should usability trials be carried out".

Results of existing studies show that their application to other techniques/representations is fairly difficult and unpredictable. Previous evaluations in the field of VL concentrated in most cases on low-level aspects (see section 3.2). Important questions that have been answered have been whether metaphors increase usability, or, whether arrows are better than undirected lines. Not many studies have concentrated on object-orientation and visualisation problems. Most studies dealing with Software Visualisation focus on the efficiency of the visualised program vs. traditional methods in teaching [LATTU-2000]. In addition, most evaluations within the VL field are not directly task-oriented towards programming. Studies have often been undertaken without a
concrete visual (programming) environment. Thus, there is still the need for further investigations and usability evaluations.

Related research fields, as mentioned in section 3.2, are HCI and cognitive psychology. Many evaluation studies in these areas have also concentrated on programming language design and its usability as well as cognitive approaches to software comprehension.

Thomas Green has proven that there are severe usability problems within text-based structured programming [GREEN-2000]. He showed, for instance, that two equally-well-structured notations caused large usability differences which could not obviously be explained by the principles of structured programming. Green introduced the "match-mismatch" law which considers the match or mismatch between the surface comprehensibility of the programming language and the task the programmer is trying to achieve: "every notation or information structure highlights some information at the cost of obscuring other information". Green has also shown through complementary studies that the "match-mismatch" law applies to graphical programming languages just as it does to textual ones, and that this applies to professionals, novices, and intermediates.

The conclusions that can be drawn from the previous statements are that task-oriented usability studies are needed for this research. The relevance and acceptance of visual programming for component-oriented development has to be proven. Since questions within VP about usability are rarely straightforward to answer, the types of usability measurements are important. There are two main approaches: the analytical methods and the empirical methods.

Known specific analysis techniques [DIX-1998] are for instance GOMS (e.g. keystroke-level analysis, CPM-GOMS), the Cognitive Walkthrough (expert appraisal based approach), and Heuristic Evaluation (i.e. operations are not fully predictable therefore usability inspection is needed).

Empirical methods are based on prototyping and specific evaluation techniques. One method is to run informal user tests or to conduct a series of field tests. The tests should be carried out under controlled conditions.
Derived from Nielsen's *Heuristic Evaluation* [NIELSEN-1994], the following questions could be raised: Does the system make use of a simple and natural language? Does the system speak the users' language? Is the memory load minimised? Is the user interface consistent? Is feedback provided? Are clearly marked exits provided? Are good error messages presented? Does the system prevent errors? Are shortcuts provided? And finally, does the user need documentation?

However, to answer these questions would only prove the quality of the visual environment and not really show the degree of usability and acceptance.

Another possibility would be to address D.A. Norman's statement: "The better the system image reflects the designer's intention, the closer the user model will be to that which the designer intended." (see next figure)

![Diagram of D.A. Norman's three models](image)

**Figure 73.** D.A. Norman's three models.

But how can the "closeness" be measured? How can it be shown that a visual system is better than a text-based system?
A first answer is: "by usability properties" such as task performance (time in secs), user satisfaction (high, middle, low) and user motivation (high, middle, low), task completion (number of tasks correctly completed), user errors (classification of error types). This consideration led to the following (null) hypothesis for the planned evaluations.

Well-designed visual systems may outperform text-based systems iff:

(1) the task completion time is (in average) shorter
(2) the code and the program structure are more "transparent" to the user
(3) the code is equal or more "compact" (i.e. "efficient")
(4) the final outcome is equal to that using the text-based system (i.e. the same result is achieved)

Question (1) addresses the creation of new applications and components. If a visual system enables users to write code in a shorter time then this is a clear advantage. Question (2) addresses the understandability of visual code. Do users know what the legal syntax is? Can other users easily understand the code? Question (3) is related to question (1) and (2). If the visual code is more "compact" or "expressive" (i.e. in terms of operations carried out) compared to the corresponding textual code, then the visual system is more efficient. Finally, question (4) includes the prerequisite that the "outcome" is semantically equal for both paradigms.

In order to test the hypothesis for component-oriented development two different environments have to be selected. The first one is VOODE/VOOPL-1 the other one is MS-Visual Studio for C++. In order to compare these environments the following questions can be considered for both approaches:

(1) How easy is it to understand existing code?
(2) How difficult is it to change an existing method and/or to write new code?
(3) What are the typical errors made when defining a new method?
(4) How motivated are users to use the approach?
The evaluation should be carried out using an observational approach. This type of usability study is useful when only a few subjects do the trials and the studies work on a more abstract level. In addition, the reasons why problems exist are more clear to the evaluator. Since the observational approach is less formal, questions may be asked during the evaluation session.

A general problem of usability trials (working on a higher level like this evaluation) is to find the subjects with the required skills. Students have only basic programming skills in textual languages and less experience in visual modelling, visual programming and CORBA development. Thus, the author had to design usability trials which can help judge the usability of the final approach. Two different trials have been carried out for this research. The first one concentrated on a comparison of textual and visual coding independent of component development. The second one addressed the development of CORBA objects using VOODE/VOOPL-1 and MS-Visual Studio. In the following section the user trials and the results are described in more detail.

6.2 User Trial 1

6.2.1 Design of the Evaluation

The objective of this evaluation is to compare the efficiency of different approaches with respect to "readability" and "writability" of a visual program. Of special interest is to find out whether a dataflow metaphor is easier to understand within visual programming than a control-flow (imperative) metaphor. As a candidate for visual dataflow programming the commercially available tool Prograph was chosen. VMB-C was used to represent the control-flow paradigm.

According to Oberlander [OBERLANDER-1999] dataflow VPLs are judged to offer particular advantages to novice programmers, but the evidence for this is in some doubt. Beaumont states in [BEAUMONT-1998, VL98 p 244] that the difficulties associated with visual control flow are well known to visual language designers. Therefore many dataflow approaches and chart-based techniques have been developed. However, Beaumont adds that chart-based techniques have proven to often fail miserably when employed to
represent code which does not adhere strictly to the use of standard selective and iterative control constructs. Corritore and Wiedenback found that novice mental representations were primarily procedural in nature \[\text{CORRITORE-1991}\]. In contrast, Adelson reported that novices form representations based on how a program functions (i.e. concrete presentations), as opposed to the more abstract representations formed by experts \[\text{ADELSON-1984}\].

The evaluation was carried out at DMU with 30 final year CS students (N.B. actually, there were 33 students but 3 students didn't return the evaluation sheets). The material used for this evaluation can be found in appendix Q. The evaluation has been structured as follows. First, a questionnaire was given to the students to get a skill profile of the students. Then there were some questions about the readability of programs. Students were asked to read a few simple programs and to write down what they thought the programs did. Prograph and Fortran code was chosen because both languages were unknown to the students.

Next, some training in Prograph and VMB-C was provided. This was necessary to explain some basic concepts for both approaches. After the training period students were asked to form three groups. After a short break the evaluation concentrated on the writability of programs. Students were asked to write a program for the calculation of fibonacci numbers greater than two. Depending on the group this has to be done using the Prograph, C/C++/Java or VMB-C notation. Each group had to use two different languages. The following table gives an overview for the design of user trial 1.

<table>
<thead>
<tr>
<th>User Trial 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) Questionnaire (2 min.) including tests on &quot;readability&quot; of Prograph/Fortran programs (max. 13 min.)</td>
</tr>
<tr>
<td>2) Prograph Training (25 min.)</td>
</tr>
<tr>
<td>3) VMB-C training (25 min.) break (15 min.) building groups A, B, C</td>
</tr>
<tr>
<td>4a) Tests on &quot;writability&quot; (three groups, max. 20 min.) group A - Prograph program group B - C/C++/Java program group C - VMB-C program</td>
</tr>
<tr>
<td>4b) Tests on &quot;writability&quot; (three groups, max. 20 min.) group C - Prograph program group A - C/C++/Java program group B - VMB-C program</td>
</tr>
</tbody>
</table>

Table 19. Design of user trials 1.
6.2.2 Students’ Skill Profile

Students were asked which programming languages they are familiar with. In addition, they should rate their degree of knowledge. The following table provides an overview of the languages mentioned and their degree of knowledge.

<table>
<thead>
<tr>
<th>Background Knowledge</th>
<th>High</th>
<th>Medium</th>
<th>Low</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>5</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>C++</td>
<td>2</td>
<td>10</td>
<td>7</td>
</tr>
<tr>
<td>Java</td>
<td>3</td>
<td>20</td>
<td>6</td>
</tr>
<tr>
<td>JavaScript</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Visual Basic</td>
<td>3</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>Visual Basic for</td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>Application/Script</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pascal</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Assembler ASM</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Delphi</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Python</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Natural</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COBOL</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Perl</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>PHP</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Smalltalk</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Macromedia</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DHTML</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 20. Questionnaire.

If clustered into categories, it becomes apparent that most students know Java, followed by C/C++ and Visual Basic (see figure below).
Background Knowledge

- Other: 20%
- C: 18%
- Visual Basic: 15%
- C++: 19%
- Java: 28%

**Figure 74.** Background knowledge.

Within the questionnaire, students were asked whether they had any knowledge of a graphical language (i.e. a programming language that makes use of icons or diagrammatic elements rather than textual code elements). Though the term *graphical language* was explained, most students mentioned textual languages. The following table shows the given answers.

<table>
<thead>
<tr>
<th>Graphical Languages mentioned by students</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delphi 5 Interactive Development Environment (IDE)</td>
</tr>
<tr>
<td>Visual Basic</td>
</tr>
<tr>
<td>JavaBeans (JBuilder)</td>
</tr>
<tr>
<td>VBA (Access) – QBE?</td>
</tr>
<tr>
<td>Macromedia</td>
</tr>
<tr>
<td>IconAuthor</td>
</tr>
<tr>
<td>VC++ (icons and images)</td>
</tr>
<tr>
<td>Java</td>
</tr>
<tr>
<td>Lingo</td>
</tr>
</tbody>
</table>

**Table 21.** Background knowledge of subjects.
6.2.3 Results

In this section the results for both parts of evaluation 1 are reported. This is done by listing the observations and problems associated with the tasks and then discussing their implications and conclusions.

6.2.3.1 Readability of Programs

Several problems have been stated by the subjects (see also appendix 0). Though the first Prograph program represents a very simple program for adding the numbers 2 and 3 and showing the result, a few students couldn't understand the semantics at all. Many students tried to understand the abstraction of the program. Examples are: "This program shows a kind of syntax tree in a graphical way", "This is a diagram of different processes to give an error result. It uses different symbols to demonstrate different joining methods", "It's showing a view of a model in which numbers are added up and displayed by using show() for output. Looks some kind of basic CASE tool". Thus, many students believed that the Prograph program is a visual representation of a concept rather than a visual executable specification.

The second Prograph program includes a list processing feature of Prograph. This is achieved by a small alteration of the icon's adornment. Most students didn't understand the meaning of this program. First of all, the expression "(2 4)" was not interpreted to represent a list of two numbers. Consequently, the adornment "(...)" was not understood. Examples of students' statements are: "Adding numbers, that is '24' and '3' using graphical notation", "Wrong list result '24+3' is '27'", "The sum of 24 is 27", "Same as previous, with different numbers and doesn't tell you about what number is being added".

The FORTRAN program also provided many problems to the students. Statements have been, for example: "Execute loop until something", "It does some maths", "Small algorithm with a loop that keeps returning itself?". What students easily identified has been that the Fortran program contains a loop. However, the meaning of the calculation within the loop was not easy to recognise. This is not surprising because the students are trained to recognise loops in other text-based languages.
After analysing the results, it became apparent that only a few students have been able to understand the programs. This is shown in the next table.

<table>
<thead>
<tr>
<th>Result of Task 1</th>
<th>Prograph 1</th>
<th>Prograph 2</th>
<th>FORTRAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct answer, or very close</td>
<td>21</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>“No idea” or completely wrong</td>
<td>9</td>
<td>23</td>
<td>28</td>
</tr>
</tbody>
</table>

Table 22. Results for Task 1.

While most students understood the meaning of the first Prograph program quite well the second Program showed the opposite result. This was caused by a small but important adornment in the visual code. A conclusion which can be drawn from this is that a small change (e.g. different slot decoration) in the graphical notation of a visual program may lead to the false results. This is known as the “fragility” of a visual language.

Furthermore, Oberlander [OBERLANDER-1999] reports that the difficulty with data flow stems partly from the large demand on memory resources necessary to understand the meaning of the visual expressions. He states that novice difficulty with dataflow is not a function of the paradigm itself, but of the way in which it is implemented. Therefore, an indication of this evaluation is that Prograph does not implement the dataflow paradigm very well and that it should improve its visual representations and concepts.

The students were asked to provide feedback about the problems they encountered whilst conducting the tasks. They stated many problems. Examples of Prograph problems stated include: “Shapes mean nothing – no visual metaphor used”, “Not sure about the meaning of some connectors at ends of lines. Also, not sure about handling of strings. Looks interesting, though!”.  

Before the training students were asked which approach they would prefer. For complex tasks only a few students stated that they would like to use Prograph. However, for simple examples many students did like the visual approach. Only 2 students mentioned that Fortran is easier to understand.
6.2.3.2 Writability of Programs

After the training of Prograph and VMB-C, students were asked to carry out a programming task. Students had the opportunity to work in groups of two. The problem was to create a program that calculates the Fibonacci number [BUEHLER-1987] for a given value. Each group was required to use two different approaches.

There were a number of ways to solve the problem. When using C/C++/Java an array of temporary variables could be used to store the results after each iteration. The solution in Prograph could rely on a recursive or iterative algorithm. The VMB-C was expected to be similar to the C/C++/Java solution.

The results show that students were more able to solve the problem when using a text-based approach (see appendix 0). An unexpected result was that there was no significant difference in performance between Prograph and VMB-C. Thus, this is an indication that the “new” visual coding dominated rather than the underlying visual paradigms. The following table shows how close students came to a solution when using one approach.

<table>
<thead>
<tr>
<th>Fibonacci numbers</th>
<th>C/C++/Java</th>
<th>Prograph</th>
<th>VMB-C</th>
</tr>
</thead>
<tbody>
<tr>
<td>solved/nearty solved</td>
<td>8</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>partly solved</td>
<td>6</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>not solved</td>
<td>6</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>

Table 23. Results on Task 2 after training.

This task clearly shows that students did perform much better when using C/C++/Java though students mentioned various difficulties. The performance between Prograph and VMB-C looked very similar at first glance. However, a few students directly mentioned that they would find Prograph easier than using VMB-C. A reason for this could be that VMB-C code was very difficult to draw on paper and the training didn’t provide sufficient information. In the following some statements of the participants are listed: “I feel that VMB-C and Prograph told [sic] promising prospects”. However, many programmers would still prefer to code rather than use the visual tools. Also to persuade programmers to use this product, you will have to provide extensive training such as on-line tutorials and
an in-depth help guide", "Text languages should be used for specific mathematical problems -> use graphical languages for user interaction", "Notation, layout and syntax were unfamiliar, although the algorithm was relatively easy to solve". These statements show that the students' belief is that visual programming is less adequate for programming than text-based languages.

After the end of the evaluations the students were asked about their motivation to use one language or the other. There was a clear vote to use a text-based system as the following table shows.

<table>
<thead>
<tr>
<th>Results after Task 2</th>
<th>C/C++/Java</th>
<th>Prograph</th>
<th>VMB-C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Which program languages would you like to use?</td>
<td>27</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 24. Results on motivation to use a language after Task 2.

One conclusion which can be drawn from this evaluation is that students are used to writing programs with textual languages and thus prefer to use them. The time to learn a new visual language like Prograph and VMB-C is prohibitive. If a language like VMB-C just attempts to make C statements visual then the performance breaks down. As a result: the claim that a VPL is "better" than a text-based language could not be proven for this evaluation. Furthermore, there is no evidence that visual dataflow programming outperforms visual control-flow programming.

One can speculate that this negative result is mainly due to education and training reasons. Most students tried to remember the textual syntax and then to solve the problem. Whether a VPL is easier to learn than a text-based language is however still doubtful because the underlying programming concepts are equally complex and are not dependent upon whether a textual or visual language is used. Another possible explanation is that the user trials have been carried out without using a visual environment. It could be argued that VP would perform better when embedded and used in a visual environment. Thus, the visibility of structures/relationships/concepts within a complete visual environment may be of more importance in order to increase the efficiency of program development.
Another general problem is that students had certain beliefs like "mathematical problems will never be solved in a graphical language!", or "maybe these both graphical languages are too old fashioned", and "we should prepare to use OO-Tools like Rational Rose. They build code from UML diagrams." Thus, the motivation is very low for using a VPL. Citrin [CITRIN-1995] suggests that students should be taught strategies to utilise the visualisation tools effectively. And Boroni [BORONI-1996] stated that software visualisation (SV) accelerated learning for students. In addition, Merrill [MERRIL-1993] reported that the novices would have needed syntax-related support. Again, according to some of the findings the development of the SV tools is not a straightforward issue. Petre, Blackwell and Green [PETRE-1992] have claimed that there is an inherent contradiction in the SV. It makes programming more concrete, but on the other hand using symbols has the habit of making things even more abstract. The result of this evaluation indicates that when judging a VL approach there is the need to show the effectiveness within a visual programming environment. This was the objective of another investigation reported in the next section.

6.3 User Trial 2

6.3.1 Background Information

The objective of the second usability evaluation is to investigate whether the visual environment VOODE/VOOPL-1 helps users in the design and implementation of CORBA objects, and what the problem areas are. It is expected that this approach would be attractive to both intermediate users and professionals. One problem of this evaluation was to find suitable subjects with sufficient knowledge of OOP and CORBA who were able to carry out the trials. This also influenced the design of this usability evaluation. Finally, this experiment was carried out with two research students from DMU and three professionals from Dresdner Bank AG (Germany). The material used for this evaluation as well as the observations and problems which occurred during the trials can be found in appendices P and Q.
6.3.2 User's Skill Profile

The subjects of the trials were asked about their OOP, UML and CORBA knowledge. The following table summarises their experience.

<table>
<thead>
<tr>
<th>Participants</th>
<th>Knowledge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Research Student 1 (RS1)</td>
<td>Java, C, XDesigner, basic CORBA theory only</td>
</tr>
<tr>
<td>Research Student 2 (RS2)</td>
<td>C++, Java, Delphi, basic UML, very basic understanding of CORBA</td>
</tr>
<tr>
<td>CORBA/JAVA Professional 1 (P1)</td>
<td>C++, Java, UML, Java Beans concepts, CORBA (Visibroker)</td>
</tr>
<tr>
<td>CORBA/JAVA Professional 2 (P2)</td>
<td>C++, Java, UML, CORBA (Visibroker, Orbix)</td>
</tr>
<tr>
<td>CORBA/JAVA Professional 3 (P3)</td>
<td>C++, Java, UML, CORBA (Visibroker, Orbix)</td>
</tr>
</tbody>
</table>

Table 25. Background knowledge of subjects.

All subjects had good or excellent OOP knowledge. The research students possessed only basic UML and CORBA knowledge. The professionals had excellent knowledge on all topics and have used these technologies in various projects for several years.

6.3.3 Design of the Evaluation

The evaluation was structured as follows. Firstly, a short overview of the objective of the trial was presented to the subjects. Then some basic training was given to the research students so that they would be able to remember basic concepts and techniques. No specific training was given to the professionals and within four minutes, the problem was stated and both environments had been shortly introduced.

Next, several tasks had to be carried out using the MS-VC++ and VOODE/VOOPL-1 environments. After each task the subjects were asked to state whether they think that they have solved the task and to write down any experiences and problems encountered whilst conducting the trials.
The following table provides an overview of the different steps involved in this evaluation.

<table>
<thead>
<tr>
<th>User Trial 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a) Introduction to evaluation (2 min.)</td>
</tr>
<tr>
<td>b) Training on MS-VC++, VOODE/VOOPL-1</td>
</tr>
<tr>
<td>(ca. 60 min. for research students, ca. 4 min. overview for professionals)</td>
</tr>
<tr>
<td>c) Tasks using MS-VC++</td>
</tr>
<tr>
<td>1a) changing an existing method, print statement</td>
</tr>
<tr>
<td>1b) if statement</td>
</tr>
<tr>
<td>feedback</td>
</tr>
<tr>
<td>2a) definition of a new method</td>
</tr>
<tr>
<td>2b) access to array elements</td>
</tr>
<tr>
<td>feedback</td>
</tr>
<tr>
<td>3) definition of CORBA server mainline</td>
</tr>
<tr>
<td>feedback</td>
</tr>
<tr>
<td>d) Tasks using VOODE/VOOPL-1</td>
</tr>
<tr>
<td>1a) changing an existing method, print statement</td>
</tr>
<tr>
<td>1b) if statement</td>
</tr>
<tr>
<td>feedback</td>
</tr>
<tr>
<td>2a) definition of a new method</td>
</tr>
<tr>
<td>2b) access to array elements</td>
</tr>
<tr>
<td>feedback</td>
</tr>
<tr>
<td>3) definition of CORBA server mainline</td>
</tr>
<tr>
<td>feedback</td>
</tr>
</tbody>
</table>

Table 26. Design of user trials 2.

The tasks concentrated on several every-day problems encountered when developing CORBA objects. The aim of the subjects was the implementation of a "grid" component. The grid consists of two dimensions called width and height and is internally represented by a two-dimensional array of long integer values (see next figure). The component has two interface methods called set and get used to store and retrieve values.
The first task was to change an existing method. The \textit{set} method should be extended so that a trace message is printed out on the screen as soon as the method is called. This represents the definition of a simple \textit{print} command. The second part of task one was to change the constructor method. It should only be possible to create arrays with a minimum size. This task requires the use of an if statement and a simple "or" condition.

The focus of task two was to evaluate how difficult it is to add a new interface method to an existing component. The first part required the definition of a new method \textit{print} and the addition of a trace message as the initial method behaviour. The second part was to write the code that displays the first five elements of the array. In order to solve this problem, the user had to think about how to access array elements in an iterative way.

Finally, task three required the definition of a standard CORBA server that instantiates the grid component. In the following sections the results are reported for each environment.
6.3.4 Design A: MS Visual Studio

Task 1 was designed to be solved in a very simple and straightforward way. Though all subjects were fluent in OO programming several problems became apparent (see appendix P). Main statements have been: "It is easier to write code when previous knowledge may be used", "Unsure about C++ syntax", "didn't know that a constructor doesn't return a value", "different naming conventions for implementation files between Orbix and Visibroker".

All subjects had some problems with the C++ syntax. In particular, the professionals mentioned that they were unsure about the correct syntax and that they would like to compile the source code. However, no participant made use of the online help.

After each of the tasks the participants were asked to state how well they thought that they had solved the tasks. Interestingly, the research students believed that they solved the task better than the author thought they had. In contrast, the professionals stated that they performed worse than judged by the author (see table below – 1 means that the task has been solved excellently – 6 means very poor, not solved).

<table>
<thead>
<tr>
<th>Participants</th>
<th>Participant / Evaluator</th>
</tr>
</thead>
<tbody>
<tr>
<td>RS 1</td>
<td>1/4</td>
</tr>
<tr>
<td>RS 2</td>
<td>3/3</td>
</tr>
<tr>
<td>P1</td>
<td>2/1</td>
</tr>
<tr>
<td>P2</td>
<td>4/2</td>
</tr>
<tr>
<td>P3</td>
<td>2/1</td>
</tr>
</tbody>
</table>

Table 27. Judgement on Task 1 completion.

The second task was expected to be more difficult to solve. The strategies used by the subjects were very different. The research students had more problems than the professionals. All subjects looked at the existing code and tried to write the new code by copying existing code fragments and then changing them. Here are some problems encountered by the participants: "It is nearly impossible to know where CORBA code comes from or MS context/relationship", "Several iterations to find correct signature: finally
copied and pasted code”, “Lots of file content navigation to see variable definitions and to look for items to ‘cut + paste’”.

Again, the research students thought that they solved the task better than judged by the evaluator. How the tasks were completed may be seen in the following table.

<table>
<thead>
<tr>
<th>Participants</th>
<th>Participant / Evaluator</th>
</tr>
</thead>
<tbody>
<tr>
<td>RS 1</td>
<td>1 / 4</td>
</tr>
<tr>
<td>RS 2</td>
<td>3 / 4</td>
</tr>
<tr>
<td>P1</td>
<td>2 / 1</td>
</tr>
<tr>
<td>P2</td>
<td>4 / 2</td>
</tr>
<tr>
<td>P3</td>
<td>2 / 1</td>
</tr>
</tbody>
</table>

Table 28. Judgement on Task 2 completion.

Task 3 required very good CORBA skills. Both research students and one professional failed completely. They didn’t remember what to do and had no clue what kind of statements are required. Examples of statements are: “Needed the Orbix documentation/header files”, “No clue how to parameterise data”, “Problem: remembering syntax and C++/CORBA”. The following table shows the judgement on task completion.

<table>
<thead>
<tr>
<th>Participants</th>
<th>Participant / Evaluator</th>
</tr>
</thead>
<tbody>
<tr>
<td>RS 1</td>
<td>6 / 6</td>
</tr>
<tr>
<td>RS 2</td>
<td>6 / 6</td>
</tr>
<tr>
<td>P1</td>
<td>6 / 6</td>
</tr>
<tr>
<td>P2</td>
<td>3 / 2</td>
</tr>
<tr>
<td>P3</td>
<td>5 / 5</td>
</tr>
</tbody>
</table>

Table 29. Judgement on Task 3 completion.

6.3.5 Design B: VOODE/VOOPL-1 for CORBA

The second part contained the same tasks but now required the use of VOODE/VOOPL-1. It was expected by the author that the subjects would perform equally well or better compared to the MS-VC++ environment. Though some basic training was carried out with the research students they mentioned that they were still unfamiliar with the visual environment (including the visual language) and therefore encountered many problems.
The professionals had less problems and quite easily recognised the concept behind the visual approach. However, they also had some problems writing the visual code. In the following, some statements are presented: “Having to decide upon layout for icon placement — no template to follow”, “problems in seeing the relationship between input parameters and icons interface”, “problems in finding where to start”, “problems in finding print processor”.

Interestingly, the research students and professionals performed about the same. Most problems derived from visual representations or interaction techniques the subjects have been unfamiliar with. The research students were more optimistic about the extent to which they thought they had solved the task (see next table).

<table>
<thead>
<tr>
<th>Participants</th>
<th>Participant / Evaluator</th>
</tr>
</thead>
<tbody>
<tr>
<td>RS 1</td>
<td>2 / 3</td>
</tr>
<tr>
<td>RS 2</td>
<td>4 / 5</td>
</tr>
<tr>
<td>P1</td>
<td>2 / 1</td>
</tr>
<tr>
<td>P2</td>
<td>3 / 2</td>
</tr>
<tr>
<td>P3</td>
<td>2 / 2</td>
</tr>
</tbody>
</table>

Table 30. Judgement on Task 1 completion.

The second task was expected to be easier to solve when using the visual environment. The definition of a new method is straightforward. First, the Interface method has to be defined in the UML model then VOODE/VOOPL-1 has to be started. However, no participant tried to change the UML model rather they directly added the new method to the IDL definition section. This means that it would have been better if the visual coding were more tightly integrated with Rational Rose. All subjects had major problems with this task. Next, the different problems are mentioned: “trying to think how to write a solution I know in code into visual objects”, “problems in understanding where to put the processors in the loop”, “problems about constructing the processors to get a ‘green light’ ”, “Still unfamiliar with the tool”. From the given statements it can be concluded that the training was not sufficient to entirely understand the visual environment. However, all participants were able to work with this environment after a short period.
The quality of the solution was quite bad as the following table shows.

<table>
<thead>
<tr>
<th>Participants</th>
<th>Participant / Evaluator</th>
</tr>
</thead>
<tbody>
<tr>
<td>RS 1</td>
<td>4 / 4</td>
</tr>
<tr>
<td>RS 2</td>
<td>6 / 4</td>
</tr>
<tr>
<td>P1</td>
<td>4 / 5</td>
</tr>
<tr>
<td>P2</td>
<td>6 / 5</td>
</tr>
<tr>
<td>P3</td>
<td>2 / 3</td>
</tr>
</tbody>
</table>

Table 31. Judgement on Task 2 completion.

This was not an expected result. One can speculate that this is mainly due to the unfamiliarity of the subjects. All users stated, that the approach is interesting and could provide many benefits to novice users. The professionals stated that they would prefer a text-based system as used in previous years.

To solve task 3 in the visual environment required less CORBA skills than in the MS-VC++ environment. All subjects found it very useful that the visual environment limited the number of selectable processors. Though a design heuristic has been applied in this context the participants were uncertain about the underlying concept. Clearly, more training and explanations were needed. Compared to the usage of MS-VC++ the results of VOODE/VOOPL-1 were judged better. The following table shows the results.

<table>
<thead>
<tr>
<th>Participants</th>
<th>Participant / Evaluator</th>
</tr>
</thead>
<tbody>
<tr>
<td>RS 1</td>
<td>3 / 4</td>
</tr>
<tr>
<td>RS 2</td>
<td>4 / 2</td>
</tr>
<tr>
<td>P1</td>
<td>3 / 3</td>
</tr>
<tr>
<td>P2</td>
<td>5 / 3</td>
</tr>
<tr>
<td>P3</td>
<td>2 / 1</td>
</tr>
</tbody>
</table>

Table 32. Judgement on Task 3 completion.

6.4 Implications Derived from Experiments

After carrying out the trials all research students and professionals stated that it was a fair evaluation. However, they were unsure about their motivation to use the VOODE/VOOPL-1 approach within a project. One conclusion which may be drawn from this is that in general the motivation of expert users is low when it comes to using a visual approach as they are more comfortable with what they are already familiar with.
It is worth noting that the results provided little evidence for substantial effects of individual differences between subjects. All subjects had some problems with the visual representations and interaction techniques due to their unfamiliarity. Furthermore, differences in task performance are due to differing knowledge of CORBA and OO.

Though the participants saw some advantages in the visual approach they believed that it would be a more suitable system for novice and intermediate developers. A general problem is that none of the subjects have used a visual programming language before. Thus, more training would have been necessary to increase their knowledge. A severe drawback is that there is no standard definition of visual icons. This situation is comparable to the introduction of window-based systems in the 80s when no standard icons for “file open”, “print” etc. existed. Now all users are used to the standardised representation of a GUI and would not like to use a command-line system.

One hypothesis of the author was that the processor code is more efficient because the time needed to correctly select and configure a processor is less than the time needed to write the equivalent and correct textual code. This may be achieved by the use of heuristics and design patterns. One example is the definition of the CORBA server mainline. The approach taken is very efficient in this respect. This could be verified and proven by the usability trials.

However, the efficient visual representation of control logic and mathematical expressions is still a problem. Experienced users are much faster at entering the code in a textual mode. A conclusion which can be drawn from this is that a complete visual system is more suitable to a novice or intermediate user who needs more direct help.

The user trials showed that a visual environment should allow the use of both visual and textual programming. This would require the definition of a hybrid system that also includes multiple views techniques so that visual and textual code may be edited in parallel. This is also the current trend in commercial tools (see for instance HTML, XML tools and JSP development tools such as DreamWeaver, or LabView that provides special windows for entering expressions as text). In addition, a visual fold editor which allows the structuring of the visual and textual code may be beneficial.
With respect to user interaction the evaluations reveal the need for better code look-up facilities. When users work with a textual approach they look at other code fragments to derive the code needed. Drag and drop facilities are also central to an efficient visual environment.

A final implication of the evaluation is that a tighter integration of UML and the VPL is beneficial. This would certainly lead to a higher acceptance and reduce the tool gap (N.B. Appendix J shows some screen dumps which highlight this idea). The evaluations have shown that there are some differences in the performance of the different survey groups. Therefore, it would be interesting and necessary to increase the number of subjects in order to elaborate the key differences. For instance, the understanding of the general problem-solving process that is underlying the combined usage of UML and VPL elements would be of great importance. In this evaluation, it became apparent that the research students weren't aware of the potential that UML may offer. There may be different reasons for this. e.g. the different viewpoints that may be associated with diagrams, the dichotomies used (i.e. the differences between type and Instance, specification and realisation, or static and dynamic), different levels of abstractions (e.g. system level, class level, method level), and the various extension mechanisms. Therefore, it may be concluded that UML is a complex notation that is not appropriate for inexperienced users and it would be interesting to further investigate the tighter integration of visual programming elements to form a coherent and complete visual system.
7 Conclusions

7.1 Vision of Research

This thesis set out a vision for improving OOP by combining Visual Modelling with Visual Programming. After conducting the research and carrying out the evaluations the question arises as to whether the vision is still valid.

Research on visual programming languages started more than three decades ago and has led to many academic and some commercial visual programming systems (e.g., see [GLINERT-1990], [ICHIKAWA-1990], [SHU-1986], Prograph [STEINMAN-1995], [PICTORIUS-2000], HP-VEE [HESEL-1998], Visual Age for C++ [NILSSON-1997], VISUAL [TRIPLES-2000]). These systems are based on different formalisms, paradigms, interaction models and techniques. As Kim Marriott stated in [CHANG-1999] the trend in visual language research is currently concentrating on application specific languages and on identifying what criteria makes one visual language formalism more suitable than another. This work also reflects this trend. In the next sections, the findings and possible future directions are outlined.

7.2 Conclusions on VMB/VMB-C

The VMB-C approach has been used mainly to identify the criteria needed to build a successful visual system. This work included several steps such as the definition of a new VPL and the development of a prototype system.

Chapter 4 describes in some depth the concept for the translation of a known textual language into an equivalent visual language. The transition of context-free languages (described by EBNF notation) into a visual programming notation via graph techniques is demonstrated. To explore the new concept, a simple language called VMB-C was implemented. It was shown that this approach is straightforward and could lead to a usable system.

One important achievement for this part of the research is that, using this technique, new visual languages may be easily explored (i.e. tested and evaluated) before
they are implemented in a new system. It is therefore an alternative approach compared to existing techniques and grammars described in the literature (e.g. graph grammars). What is important in this new approach is the separation of the syntax/semantics from their visual representation by using the described code file for each lexeme. The tool may not only be useful for visual programming but also for visual configuration systems. Thus, the Visual Meta Builder represents a multi-purpose environment.

The main benefits of the node modelling language for defining VPLs are the description of lexemes within a text file (i.e. code file), the easy extension respectively refinement of new language features, the possibility to change the visual representation of a lexeme by replacing the underlying bitmaps, and the independence of a given paradigm (i.e. controlflow vs. dataflow) – in summary: the open concept.

Some aspects of this approach were tested by user evaluations. At a first glance this approach seems unappealing to both students and professionals alike. This is not surprising as nearly all CS students are not trained for visual programming and related paradigms (e.g. data-flow programming). For instance, the DMU curriculum for CS students covers only OOP (main language is Java) and other text-based approaches and not a single visual approach. Finally, the evaluation has shown that the claim of VL researchers that visual programming is better could not be proven. The author concludes that the discussion of whether a VPL is better then a text-based language is the “wrong” discussion. The effectiveness of a visual approach should only be measured within a concrete visual (programming) environment.

7.3 Conclusions on VOODE/VOOPL-1

The main objective of the second part of this research was to investigate the issues of a complete visual system that supports the design and implementation of CORBA objects [MOWBRAY-1997a/b], [OMG-2000] and to evaluate its applicability. This research has created several ideas and basic concepts concerning the combination of visual modelling with visual programming. VOODE/VOOPL-1 contains aspects of all important ideas outlined in chapter 5 and is judged to be potentially the best candidate.
The VOODE/VOOPL-1 prototype provides a way for *visual programming in the large*, and wants to "leave the visual language ghetto" [MUENCH-1999]. The work is partly influenced by the Vista system [BURNETT-1995] and the general trend in component development. This approach makes use of the processor idea as introduced in the VMB approach. VOODE/VOOPL-1 aims to provide a complete visual system (i.e. hiding the underlying textual language as much as possible) in contrast to Vista which is integrated in a Smalltalk environment. The new contribution of the work is that visual modelling can be easily combined with visual object-oriented programming. It was not an attempt to replace or extend UML and invent a new more powerful language but to rely on the strength of UML whilst linking it to a new VPL.

The prototype system is based on the earlier VMB tool and was extended in order to discover more about the tradeoffs involved in combining visual modelling and visual programming for CORBA component development. The approach is not limited to CORBA but may also be suitable for other component technologies such as EJB. The intended system can be called a complete visual system as it does not force an application developer to use any textual programming code. It includes visual modelling and visual programming. Furthermore, it emphasises the need for a tool that supports the component-oriented software development.

The concepts of VOODE/VOOPL-1 and their usefulness were verified by usability evaluations. The results indicate that the claim that complete visual systems outperform visual environments with textual programming even in a domain-specific/general-purpose problem domain such as CORBA enterprise component development is difficult to prove. The results of the evaluation indicate that professional programmers do not like visual programming (and not even generator approaches used in current commercial tools) whilst novice and intermediate programmers need to be trained in the new paradigm before being able to use the visual system effectively.

Nevertheless, in the context of VOODE/VOOPL-1 the training required may be less than in a text-based approach since the system avoids the gaps between graphical environments and string-based programming, and the different mapping problems (i.e. from UML to IDL and C++). To prove this statement would require another evaluation based on more students and further comparison studies which is beyond the scope of this
research. However, the evaluation clearly indicated that novice and intermediate users would benefit from this approach.

7.4 Major Contributions of this Thesis to the VL Community

After conducting this research it is important to mention what the outcome of this research is. The author started to investigate VOOPLs from the work and experiences of the VL community. Therefore the contributions are very important for this research field.

This work shows the definition of a new approach for defining VPLs based on a node modelling language. Different mapping concepts and abstractions into visual environments are described. Much research in the VL field concentrates on formal languages and how compilers could be made more efficient. However, this work should be extended more towards visual environments and interaction techniques. In addition, the results of this research have shown that the claim that VPLs are better than textual/string programming languages could not be supported in all cases. A VPL "lives" in a visual environment. Thus, direct manipulation and the visualisation of structures and relationships are of fundamental importance.

The combination of visual modelling with visual programming to form a complete visual object-oriented system is straightforward and very attractive. All subjects of the trials mentioned that they did like the system. Professionals judged it to be a good approach for novices and intermediate programmers. Since UML is not meant to be a visual programming language this is a good chance for the VL community to investigate the combination of visual modelling and visual programming. In addition, component development is extremely important and VLs could make a major contribution to this field. This would also influence the way of the development of component-based development, especially when the separation of component modelling, component implementation and component assembly is considered.

Another result of this research is that visual programming and its integration in a visual environment makes hidden dependencies more visible. Of great importance is the direct manipulation of visual elements. Control logic and mathematical equations are easier to implement in a text-based approach. Alternatively, a user may choose which
representation is more suitable and use it. This clearly requires a hybrid system that combines the strengths of VP with the benefits of text-based programming. In order to achieve this, another meta level would be needed that describes the behaviour of the code and which would present the synchronised interaction in multiple views.

7.5 Recommendations for Further Research

Future directions for the Visual Meta Builder could be many-fold. One possibility is that the node modelling language could be extended so that event-driven visual object-oriented programming languages may be easily designed and tested. Another idea is that the specification of the environment could be made more generic. This means that different visual environments could easily be built, configured and tested. Then it would be directly comparable to approaches such as VLCC [COSTAGLIOLA-1995] and Vampire [MCINTYRE-1995]. A useful extension would also be to investigate how textual code may be automatically generated based on the lexeme's description. This would make the system even more generic and useful resulting in a more efficient development of complex visual systems.

In addition, the formalism used within VOODE/VOOPL-1 are well understood and less important than the way it attempts to visualise design and implementation issues for CORBA component development. There could be many future directions for this approach. For instance, some extensions, like an improved syntax checker and additional processor types, could be implemented in another release. Another possibility would be to explore more advanced user interactions and to investigate how visual programming and textual programming could be combined and integrated in an hybrid object-oriented approach. In addition, a tighter integration into a CASE tool and the UML models should be investigated. A complete set of processors should be identified, implemented and tested. This research activity should also include efficient design and code patterns.

Of special interest is also the generalisation of visual component development. Therefore the shift from the CORBA technology to other component implementations such as EJB and COM/DCOM it should be investigated. The different and common features
have to be identified and adaptations to the visual representations have to be created. This may also address the way in which components are modelled with UML.

Further investigations for this VPL approach should also concentrate on usability trials. A crucial question is to what extent it is possible to develop a complete visual system that is acceptable by (all) professional users. This investigation should include the transition of more complex programming abstractions (i.e. architectural and design patterns for component development) into the visual programming environment and their evaluations for acceptance and usability/effectiveness. The trials should be set up with enough subjects coming from industry projects. In addition, a specific course for undergraduate students should be implemented in order to gain more insights into the various problem areas of VPLs. The evaluations should contain low-level investigations such as efficient interaction techniques (e.g. for selecting the intended visual symbol) and other visual feedback mechanisms (e.g. for bad syntax and style).

A specific problem for all visual systems is the "connect problem". This means that it is not clear in all circumstances which connections are possible and allowed (e.g. attribute-attribute connection, or 1:n/1:n connections). This issue could lead to a mixed approach: the general programming concept may follow the controlflow paradigm whereas the attribute connections are in accordance to the dataflow concepts. The implications of such a mixed approach should be evaluated.

Finally, concrete programming tasks (e.g. accessing other component services or addressing persistence issues) should be defined to prove the usability of the system. These investigations should concentrate on server specifics such as timeout and exception handling, message printing, instantiation of CORBA objects. The many CORBA implementation issues such as the definition of new helper classes, the specification of component behaviour, the finding and corrections of errors and the completion of a component implementation should also be taken into account. Ideally, all user interactions necessary for textual languages within a graphical environment should be converted and their usability tested in the complete visual environment. All trials should be conducted such that the "readability" and "writeability" of the approach is evaluated. Furthermore, to gain some statistical significance, the number of subjects for each trial should be at least 15 and should include both professional and novice users.
References


programming languages, IEEE Computer, pp 57 to 69, 08/1983.

[SHU-1986] N. C. Shu, Visual Programming Languages: A Perspective and a Dimensional
Analysis, pp 11 to 34, Plenum, New York, 1986.


[SMITH-1977] D. C. Smith, PYGMALION: A computer program to model and stimulate

[SNOBOL-2000] Introduction to SNOBOL Programming Language,

[STEINMAN-1995] Scott B Steinman et al., Visual Programming with Prograph CPX,
Manning, 04/1995.


communication system, AFIPS Conference Proceedings, Spring Joint Conference, pp
2 to 19, 1963.

[SZYPERSKI-1998] Clemens Szyperski, Component Software, Beyond Object-Oriented


facilitate novices' understanding in a new domain: Forward and backward reasoning
in programming, Proceedings of the Thirteenth Annual Conference of the Cognitive


[TSOFT-2000] TogetherSoft, TogetherJ,


[UML-1999] UML, Unified Modeling Language, Version 1.0, 1.1, 1.2,
[UML-2000] Object Management Group, Inc., OMG homepage for UML,  


[VL-2000] Proceedings of 16th IEEE Symposium on Visual Languages, Seattle,  

[WASHBURNE-1927] J. N. Washburne, An experimental study of various graphical,  
tabular and textual methods of presenting quantitive material, Journal of Educational  
Psychology, 18, pp 361 to 476, 1927.

[WEBSTER-1995] Bruce F. Webster, Pitfalls of Object-Oriented Development, M&T  

[WIRTH-1996] N. Wirth, Compiler construction, Addison Wesley Longman Limited,  
1996.


[ZHANG-1998] Da-Qian Zhang et al., VisPro: A Visual Language Generation Toolset,  
Proceedings of 14th IEEE Symposium on Visual Languages, Nova Scotia, Canada,  
1998.