Michael Kölling

The design of an object-oriented environment and language for teaching

Dissertation submitted in fulfilment of
the requirements for the degree of

Doctor of Philosophy

at the Basser Department of Computer
Science, University of Sydney

1999

To my parents,
Klaus and Brigitte Kölling

Abstract

While object-orientation has been widely accepted as an important programming paradigm, teaching object-orientation remains difficult. Experience reports suggest that some problems can be avoided by teaching object-orientation as the first language in an introductory course. However, other problems remain, in particular languages and environments are regularly criticised as unsuitable and overly complex.

This thesis discusses aspects of object-oriented languages and environments that affect their suitability for first year teaching. General requirements for languages and environments are identified and used as a means to judge the suitability of the most popular systems in use today. The discussion shows that none of the currently available systems is suitable for object-oriented first year teaching.

A new integrated software development environment and language called *Blue* is presented. Blue overcomes the problems identified and opens new possibilities for teaching and learning object-oriented programming. The Blue language and environment are highly integrated and specifically designed for teaching. This focus allows the provision of tools and techniques not available in other systems. The Blue language is small and clean and the environment supports visualisation and sophisticated interaction facilities.

In this thesis, both the language and the environment are described in detail. Design decisions and alternatives are discussed and compared to other systems on the market.

Acknowledgments

Many people had contact with and made contributions to the Blue project since it started four years ago. Although Blue started as my PhD project, it could not have grown to its current state without the input of these people. Blue has become much more than just my project: it has been used by many students and teachers, and this could never have happened without generous support by others.

The most important input came from Professor John Rosenberg, my supervisor, who was part of all design decisions throughout the whole lifetime of this project.

Oscar Bignucolo implemented the Blue compiler, a substantial part of the Blue system. In doing so, he produced one of the most reliable parts of the system while often having to respond to my changing requests for features, which he handled in impressively short times. I thoroughly enjoyed working with him.

Axel Schmolitzky designed the Blue GUI library, which was then implemented mainly by Michael Cahill and Ute Toberer. Michael Cahill also created the Blue CD which was handed out to students and greatly helped with the acceptance of the system.

Jeff Kingston helped design the Blue collection library – his knowledge and experience was very welcome. He also implemented this library.

Stefanie Fetzer implemented part of the Blue abstract machine and also wrote the first version of the Abstract Machine Manual.

Jeff Kingston, Alan Fekete, Tony Greening, Judy Kay and Nicole Lesley produced a textbook for a computer science introductory course using Blue.

Heather McCarthy worked on an improved design for a library browser for Blue.

I also have to thank the Basser Department of Computer Science at the University of Sydney as a whole for taking the brave step of deciding to use the Blue system for teaching their first year course and the students at the department for patiently putting up with the system's initial shortcomings. Many students also contributed through their never ending energy in finding and reporting bugs.

I am further indebted to some people for personal support and encouragement that have laid the foundation for this work, sometimes many years ago, and without whom I would not have been able to finish (or even start) this thesis.

Firstly, I am forever grateful to my parents who convinced me of the value of a good education and enabled me to study.

I have to thank my good friends Detlef Rudolph, who first introduced me to computers and awoke my interest in these machines, and Axel Schmolitzky, who helped a great deal in maintaining the fun of it while we were studying. Axel also gave valuable comments on a first draft of this thesis. Damien Watkins read large parts of this work, and gave helpful feedback.

I am grateful to Professor J. L. Keedy who has opened many doors for me that have turned out to be important in my life, one of which was the one to Australia.

Very special thanks go to Professor John Rosenberg who, as my supervisor, not only provided the most excellent help and supervision I could wish for, but also was of great importance to me privately when I arrived in this then foreign country. His support enabled me to find my way and build a new life in Australia.

Finally, I thank my wife Leah for all her love and understanding during these years. She put up with my long work hours and at the same time managed to cope with a pregnancy and the birth of our daughter Sophie, who is now seven months old.

mk, 9 September1998

Table of contents

1 INTRODUCTION 1

2 REQUIREMENTS FOR A TEACHING SYSTEM 4

2.1 ADVANTAGES OF OBJECT-ORIENTATION FOR FIRST YEAR TEACHING 4
2.2 REQUIREMENTS FOR THE LANGUAGE 5
2.2.1 Language requirements overview 6
2.2.2 Language requirements in detail 7
2.2.3 How simple should a language be? 14
2.3 REQUIREMENTS FOR THE ENVIRONMENT 15
2.3.1 Environment requirements overview 16
2.3.2 Environment requirements in detail 16
2.4 SUMMARY 21

3 LANGUAGES 23

3.1 C++ 24
3.2 SMALLTALK 28
3.3 EIFFEL 32
3.4 JAVA 35
3.5 OTHER LANGUAGES 39
3.6 SUMMARY 41

4 ENVIRONMENTS 42

4.1 BACKGROUND 42
4.2 ENVIRONMENTS FOR OBJECT-ORIENTATION 44
4.2.1 Object support 45
4.2.2 Visualisation support 47
4.3 ENVIRONMENTS FOR TEACHING 49
4.4 SOME ENVIRONMENT EXAMPLES 50
4.5 SUMMARY 54

5 THE BLUE SYSTEM - AN OVERVIEW 55

5.1 GETTING STARTED 56
5.2 EDITING AND COMPILING 57
5.3 CREATING OBJECTS 59

5.4 INTERFACES 61
5.5 USING LIBRARY CLASSES 62
5.6 DEBUGGING 62
5.7 SUMMARY 63

6 THE BLUE LANGUAGE 64

6.1 INTRODUCTION 64
6.2 THE OBJECT MODEL 66
6.2.1 What is an object? 66
6.2.2 Storage of objects 70
6.2.3 Object creation 71
6.2.4 Manifest classes 71
6.3 THE TYPE SYSTEM 74
6.3.1 Type safety 74
6.3.2 Types vs. classes 75
6.3.3 Predefined types 76
6.3.4 Type conformance 77
6.4 CLASSES 77
6.4.1 Constructor classes 77
6.4.2 Enumeration classes 78
6.4.3 Encapsulation 79
6.5 ROUTINES AND PARAMETERS 80
6.5.1 Structure 80
6.5.2 Parameters and result variables 83
6.5.3 Multi-assignments 84
6.6 DESIGN BY CONTRACT 85
6.6.1 Pre and post conditions 85
6.6.2 Class invariants 86
6.7 COMMENTS 87
6.7.1 Compulsory comments 88
6.7.2 Interface comments vs. implementation comments 89
6.8 VARIABLES 90
6.8.1 Undefined variables 90
6.8.2 Nil 91
6.9 STATEMENTS 93
6.9.1 Assignment and assignment attempt 93
6.9.2 Procedure call 96
6.9.3 Return from routine 96
6.9.4 Assert statement 97
6.10 ALIASES 97
6.11 CONTROL STRUCTURES 99
6.11.1 Conditional 99
6.11.2 Selection 101
6.11.3 Iteration 103
6.12 EXPRESSIONS 104
6.12.1 Equality 104

6.12.2 Set membership 109
6.12.3 Object creation 109
6.13 INHERITANCE 113
6.13.1 Inheritance - the Swiss army knife 114
6.13.2 Inheritance for subtyping 114
6.13.3 Inheritance for code reuse 115
6.13.4 Problems with inheritance 116
6.13.5 Inheritance in Blue 116
6.13.6 Inheritance and creation 117
6.13.7 Access protection 119
6.14 GENERICITY 120
6.14.1 Unconstrained genericity 120
6.14.2 Operations in generic classes 121
6.14.3 Constrained genericity 122
6.14.4 Genericity and conformance 122
6.15 CONCEPTS NOT INCLUDED IN BLUE 123
6.15.1 Multiple constructors 124
6.15.2 Function overloading 124
6.15.3 User defined infix operators 125
6.15.4 Explicit blocks 126
6.15.5 Routine parameters 127
6.15.6 Immediate objects 127
6.15.7 Multiple inheritance 128
6.15.8 Iterators 129
6.16 SUMMARY 130

7 THE BLUE ENVIRONMENT 131

7.1 INTRODUCTION 131
7.2 KEEPING IT EASY – THE USER INTERFACE 133
7.3 THE PROJECT 134
7.3.1 Working with structure 134
7.3.2 Design notation 137
7.3.3 System abstraction 138
7.4 EDITING 138
7.4.1 Class views: interface and implementation 138
7.4.2 Graphical vs. textual editing 141
7.4.3 Alternatives 142
7.5 COMPILING 144
7.5.1 Invoking the compiler 144
7.5.2 Display of error messages 145
7.6 INTERACTING WITH OBJECTS 146
7.6.1 Calling interface routines 146
7.6.2 Linguistic Reflection 150
7.6.3 Composition 152
7.6.4 Inspection of objects 152
7.6.5 A word on testing 154
7.6.6 Execution without I/O 156
7.6.7 Pedagogical benefits 156

7.7 RUNTIME SUPPORT 157
7.7.1 Error detection 157
7.7.2 Instruction counting 159
7.8 DEBUGGING 160
7.8.1 Integration 160
7.8.2 Functionality 160
7.9 BROWSING CLASS LIBRARIES 162
7.9.1 Browsing 164
7.9.2 Searching 165
7.9.3 Integration 165
7.9.4 Documentation 165
7.9.5 Additional functionality 166
7.9.6 The libraries 167
7.10 GROUP SUPPORT 167
7.11 SUMMARY 170

8 IMPLEMENTATION 172

8.1 IMPLEMENTATION ENVIRONMENT 172
8.2 SOFTWARE ARCHITECTURE 173
8.3 THE COMPILER 175
8.4 THE ABSTRACT MACHINE 176

9 EXPERIENCE 177

9.1 RUNTIME ENVIRONMENT 177
9.2 FEEDBACK 177
9.3 STABILITY 178
9.4 PROBLEMS 178

10 FUTURE WORK 181

10.1 COMPLETION AND ENHANCEMENTS 181
10.1.1 Completion 181
10.1.2 Enhancements of functionality 182
10.1.3 Enhancements of implementation 184
10.2 EVALUATION 185
10.3 SUPPORT MATERIALS 185
10.4 PORTING TO OTHER PLATFORMS 186
10.5 FOLLOW-ON PROJECTS 186
10.6 SUMMARY 187

11 CONCLUSION 188

APPENDIX A: RELATED DOCUMENTS **190**

APPENDIX B: DOWNLOAD LOCATIONS **192**

APPENDIX C: CD CONTENTS **193**

REFERENCES **194**

Table of figures

FIGURE 5.1: THE BLUE MAIN WINDOW 56
FIGURE 5.2: EDITOR WINDOWS SHOW THE SOURCE OF CLASSES........ 57
FIGURE 5.3: DISPLAY OF COMPILER ERROR MESSAGE IN AN EDITOR WINDOW........ 58
FIGURE 5.4: AN OBJECT ON THE OBJECT BENCH........ 59
FIGURE 5.5: OBJECT MENU ENABLES INTERACTIVE ROUTINE CALLS........ 60
FIGURE 5.6: INTERFACE OF A BLUE ROUTINE........ 60
FIGURE 5.7: THE LIBRARY BROWSER........ 61
FIGURE 5.8: SOURCE OF A CLASS WITH BREAKPOINT........ 62
FIGURE 6.1: AN EXAMPLE OF A BLUE CLASS........ 65
FIGURE 6.2: ASSIGNMENT OF OBJECTS 72
FIGURE 6.3: STRUCTURE OF A CLASS........ 78
FIGURE 6.4: STRUCTURE OF AN ENUMERATION CLASS 79
FIGURE 6.5: STRUCTURE OF A ROUTINE........ 81
FIGURE 6.6: STRUCTURE OF A ROUTINE IN INTERFACE VIEW........ 82
FIGURE 6.7: FORMAT OF A CLASS COMMENT........ 88
FIGURE 6.8: INTERFACE AND IMPLEMENTATION COMMENTS IN IMPLEMENTATION VIEW 89
FIGURE 6.9: ROUTINE IN INTERFACE VIEW 89
FIGURE 6.10: STRUCTURE OF A LOOP 103
FIGURE 6.11: OBJECTS AND EQUALITY........ 105
FIGURE 7.1: THE BLUE MAIN WINDOW 132
FIGURE 7.2: INHERITANCE AND USES ARROWS IN THE PROJECT DISPLAY 135
FIGURE 7.3: IMPLEMENTATION VIEW OF A CLASS 139
FIGURE 7.4: INTERFACE VIEW OF A CLASS........ 140
FIGURE 7.5: ICONS INDICATE WHETHER A CLASS HAS BEEN COMPILED 144
FIGURE 7.6: AN ERROR MESSAGE REPORTED BY THE COMPILER 145
FIGURE 7.7: INTERFACE OF CLASS “PERSON”........ 147
FIGURE 7.8: OBJECT CREATION DIALOGUE........ 148
FIGURE 7.9: AN OBJECT ON THE OBJECT BENCH........ 149
FIGURE 7.10: CALLING A ROUTINE ON AN OBJECT 149
FIGURE 7.11: ROUTINE CALL DIALOGUE FOR “CHANGENAMES”........ 150
FIGURE 7.12: RESULT DIALOGUE FOR FUNCTION “GETNAMES” 150
FIGURE 7.13: OBJECT INSPECTION DIALOGUE 152
FIGURE 7.14: INSPECTION OF “ADDRESS” OBJECT........ 153
FIGURE 7.15: DISPLAY OF A RUNTIME ERROR 158
FIGURE 7.16: THE INSTRUCTION COUNTER........ 159
FIGURE 7.17: SETTING A BREAKPOINT........ 160
FIGURE 7.18: THE EXECUTION CONTROLS........ 161
FIGURE 7.19: DISPLAY OF CALL SEQUENCE AND VARIABLES 162
FIGURE 7.20: THE BLUE LIBRARY BROWSER........ 163
FIGURE 7.21: CLASS DISPLAY IN A GROUP PROJECT........ 168
FIGURE 8.1: APPLICATION STRUCTURE OVERVIEW 173

1 Introduction

Object-oriented programming has, in recent years, become the most influential programming paradigm. It is widely used in education and industry, and almost every university teaches object-orientation somewhere in its curriculum. The software community more or less agrees that teaching object-oriented programming is a good thing. It elegantly supports the concepts that we have been trying to teach for many years, such as well structured programming, modularisation and program design. It also supports techniques for approaching problems that have only more recently made their way into the curriculum: programming in teams, maintenance of large systems and software reuse. In short, object-oriented programming seems to be a good tool for teaching those programming methodologies that we consider important.

Teaching object-oriented programming, however, remains difficult.

Many reports of the experience of those attempting to teach object-oriented programming include a long list of problems with many different aspects of the systems used. (We will discuss these in more detail later.) Why is it difficult? Or, to be more precise, why does the teaching of object-oriented programming seem to be more difficult than the teaching of structured programming?

Before we attempt an answer to this question, we should look at one other aspect of this problem: *when* should object-oriented programming be taught?

For a long time, object-oriented programming was considered an advanced subject that was taught late in the curriculum. This is slowly changing: more and more universities have started to teach object-orientation in their first programming course. The main reason for doing this is the often cited problem of the *paradigm shift*. Learning to program in an object-oriented style seems to be very difficult after being used to a procedural style. Anecdotal evidence (e.g. in [Stroustrup 1994]) indicates that it takes the average programmer 6 to 18 months to switch his or her mind set from a procedural to an object-oriented view of the world. Experience on the other hand also shows, that students do not seem to have any difficulty understanding

object-oriented principles if they encounter them first. It is the switch that is difficult, not object-orientation.

For teaching programming, the lesson is clear: if we want to teach object-orientation, we should do it first. The path to object-orientation through procedural programming is unnecessarily complicated. Students first learn one style of programming, then they have to “un-learn” the previously learned, before we show them how to do it “right”.

Unfortunately, many text books use procedural programming as a pathway to object concepts. One of the main influences in this is the language C++ [Stroustrup 1991], which has evolved as the most widespread object-oriented language over the past years. Its popularity in industry (which is best explained by historic developments) has led to it being often used for teaching as well. C++ is a hybrid language that supports procedural (“C style”) programming and object-oriented programming. It was developed as an extension to C. This has led to a misunderstanding: many people view object-orientation as just another language construct that can be taught after control structures, pointers and recursion. This is a serious mistake.

Object-orientation is an underlying paradigm that shapes our whole way of thinking about how to map a problem onto an algorithmic model. It determines in fundamental ways the structure of even simple programs. It cannot be “added on” to other language constructs; rather it replaces the fundamental structure of procedural programming.

Because of this, we firmly believe that object-oriented concepts should be taught from the very beginning. If it is seen as necessary that students can also program in a procedural style, then a procedural language can be introduced later. The paradigm shift backwards (from object-orientation to procedural) is much easier. Programmers can just think about it as writing the complete solution within one class (a solution an object-oriented programmer might not be very happy about, but easy enough to understand).

Having come to this conclusion, we can now ask the question about the difficulties in teaching object-orientation more precisely: Why is it difficult to teach object-oriented programming to first year students?

In our view, it is not object-orientation in principle that causes the problems, but the tools available to teach it. Programming languages used are too complex and programming environments – if they exist at all – are too confusing. Some systems used for teaching were really developed for professional software engineers, making it difficult for first year students to cope; others were not “developed” at all but grew out of historic coincidences.

In short: in our view the reason for all the trouble is that the wrong languages and environments are being used.

That, of course, immediately leads to the next question: What tools should we use? What should a good language and environment for teaching object-oriented programming look like?

This is the question that we try to answer in this thesis. Chapter 2 provides a detailed discussion of requirements for a teaching system for object-oriented programming. What characteristics should a language and an environment have to be useful for teaching? In chapters 3 and 4 we discuss existing systems and compare them with our requirements. We point out why we came to the conclusion that none of them is ideal for teaching. Chapters 5, 6 and 7 give a detailed description of the Blue system – our attempt to design a system that is better suited to the task. Readers who already agree with us on shortcomings of existing systems might like to start reading at chapter 5, which gives an overview over the Blue system, to determine whether this work is interesting to them.

Chapter 8 gives an overview of the implementation of the Blue system and chapter 9 reports some experiences with its use for teaching first year students. Chapters 10 and 11 describe some future projects and our conclusions.

2 Requirements for a Teaching System

In the previous chapter, we have already mentioned some arguments for teaching an object-oriented language in first year. It is not really the topic of this work to argue *why* the use of an object-oriented language in first year is advantageous – we take this as a premise and investigate *how* to best teach object-oriented programming in first year. We will, nevertheless, summarise here some arguments for the use of object-orientation for first year teaching to provide a basis for later discussion. The reasons for using an object-oriented language give us some aims on which to base the requirements for such a language.

This is followed by a discussion in detail of the requirements for a system for teaching object-oriented programming. These requirements fall into two categories: requirements of the language itself, and requirements of the programming environment. (These requirements have been published in a much shorter form in [Kölling 1995].)

2.1 Advantages of object-orientation for first year teaching

Several arguments strongly support the use of an object-oriented language in first year:

- Object-orientation encourages well structured programming, which is one of the most important lessons we try to convey to first year students.
- Re-using existing code can be taught in addition to the development of new code, leading to a more realistic perception of the tasks expected of a programmer.

- The opportunity for students to make use of ready-made objects in their applications opens a wide range of possibilities for real-world problems and interesting examples and exercises. Skublics et al. [Skublics 1991], for example, write: "With Smalltalk, it is easier to use examples which model the real world problems which students understand and enjoy". LaLonde and Pugh report the same experience [LaLonde 1990]. This is true for object-oriented programming in general, not only for Smalltalk in particular.
- Important software development concepts, such as evolution and reuse, can be introduced and experienced through object-oriented techniques at an early stage.
- Problems with the paradigm shift in moving between object-oriented and non-object-oriented environments seem to be reduced. It has been found that many students whose first programming language is a procedural language, such as Pascal, experience problems in adjusting to the object-oriented paradigm [Decker 1993, Ryba 1997, Skublics 1991, Temte 1991]. On the other hand, switching from an object-oriented language to a non-object-oriented one is not anticipated to cause as much difficulty (provided the syntax is not too different) [Decker 1993]. If introduced as a first programming style, object-orientation is quickly accepted as "natural" by beginners [Böhm 1997].
- Object-orientation currently is the most popular paradigm in industry. This provides a strong motivation for students, since they learn state-of-the-art technology that they can use in their later careers.

All of these arguments support the idea of using the object-oriented paradigm in teaching programming in the first year of a computer science course.

Unfortunately when one examines the object-oriented languages which are available, all reveal major deficiencies which make them inappropriate as a first year teaching language. We would contend that there is a major need for a new object-oriented programming language *specifically* designed for teaching. Such a language would serve a similar purpose to that of Pascal in the 1980s.

To support that claim, we will now identify the requirements of an object-oriented teaching language. These requirements then serve as a means of measurement to judge the suitability of various existing languages, and will later serve as a guideline to design our own system.

2.2 Requirements for the language

The requirements for object-oriented languages in general have often been discussed in the literature. Many arguments have been brought forward for and against specific language constructs; every possible feature has been argued for and against. The question as to whether or not multiple inheritance is necessary, for example, has sparked ongoing disputes, often carried out with religious fervour.

General discussions like those are often pointless and unproductive. Languages are not good or bad *per se*; they are good or bad *for a specific purpose*. A language that is bad in one context can be excellent in another (or vice versa). It is more a question of the right tool for the right job: it is pointless to argue whether a hammer is better than a screwdriver if you are not arguing in the specific context of what you want to do.

The following discussion of requirements for an object-oriented language is in the context of the use as a teaching language for beginners. The following criteria should be met by a language *to be useful as a teaching language* for first year students.

We first list the criteria in a brief form, and then go on to elaborate in more detail.

2.2.1 Language requirements overview

- *Clean concepts* – The language should support clean, simple and well-defined concepts. This applies especially to the type system, which will have a major influence on the structure of the language. The basic concepts of object-oriented programming, such as information hiding, inheritance, type parameterisation and dynamic dispatch, should be supported in a consistent and easily understandable manner.

- *Pure object-orientation* – The language should exhibit "pure" object-orientation in the sense that object-oriented constructs are not an additional option amongst other possible structures, but are the basic abstraction used in programming.

- *Safety* – It should avoid concepts that are likely to result in erroneous programs. In particular it should have a safe, statically checked (as far as possible) type system, no explicit pointers and no undetectable uninitialised variables.

- *High level* – The language should not include constructs that concern machine internals and have no semantic value. This includes, most importantly, dynamic storage allocation. As a result the system must provide automatic garbage collection.

- *Simple object/execution model* – It should have a well defined, easily understandable execution model.

- *Readable syntax* – The language should have an easily readable, consistent syntax. Consistency and understandability are enhanced by ensuring that the same syntax is used for semantically similar constructs and that different syntax is used for other constructs.

- *No redundancy* – The language itself should be small, clear and avoid redundancy in language constructs.

- *Small* – The language should be as small as possible (while including all necessary features). This is different from avoiding redundancy: not only

should there be only one way to do one thing, but concepts which we do not want to discuss should not be present in the language.

- *Easy transition* – The language should, as far as possible, ease the transition to other widely used languages, such as C++, Java [Arnold 1996] and Smalltalk [Goldberg 1989].

- *Correctness assurance* – It should provide support for correctness assurance, such as assertions and pre and post conditions.

- *Environment* – Finally, the language should have an easy-to-use development environment, including a debugger, so that the students can concentrate on learning programming concepts rather than the environment itself.

Note that some issues, such as efficiency, which are often considered extremely important for production programming languages, are of little significance for a teaching language; it is only required that the language be able to be supported in a teaching environment with reasonable response time. Similarly, it is not important that the language be flexible enough to develop real-world applications (e.g. by the inclusion of operations such as arbitrary bit manipulation) – it is not intended to be used for this purpose.

We now discuss the points listed above in more detail.

2.2.2 Language requirements in detail

1 *Clean concepts*

The concepts that we want to teach should be represented in the language in a clean, consistent and easy-to-understand way. In particular, they should be represented in the same way we want to talk about them when teaching. We should avoid presenting a model of a construct in lectures in one way and using a language that implements variations of the model (or does not implement the model at all). We should also not let the language dictate what we have to talk about in first year lectures.

When we, for instance, talk about the object model of programming, we want to present a consistent, easy-to-understand model as the basis of discussion. We can then talk about message passing and object methods. In that case we would not want a language that forces us at implementation time to insert definitions about which functions are or are not dynamically dispatched. This is not part of the conceptual model at this point, and the language should not force us down to this level. We also do not want to be forced to think about memory layout or other machine details. The implementation language should reflect the level of abstraction that we want to use for our conceptual models.

2 *Pure object-orientation*

The expression "pure object-orientation" is chosen to mean the opposite of "hybrid" languages – languages that support the object-oriented but also non-object-oriented paradigms. (C++, for example, is a hybrid language.)

Many people have argued for hybrid languages, in particular for C++, on the basis that the hybrid character of the language might ease the entry to object-oriented programming for students with prior programming experience [Biddle 1994, Stroustrup 1994]. If a student knows C already, a hybrid language like C++ might provide an easy entry path. Object-orientation could be gradually added to an existing body of knowledge, making understanding of the problems easier by flattening the learning curve.

While this argument sounds plausible at first, it is fundamentally flawed.

It is true that many students enter universities with prior programming experience, and it would be foolish to ignore this fact, or not to try to exploit its benefits. On the other hand, it is often the case that students come into the institution with a self-taught "cowboy" style of programming that in no way resembles the good programming practices which we try to convey. In teaching, we have to make sure that those students change their habits over time. Changing one's habits, however, is harder than learning new ones. The problem is described in detail by a group of computing experts in [Pancake 1995].

The danger with hybrid languages is that students with prior programming experience in a procedural language are not encouraged to change their style. On the contrary – they can write programs for a long time, believing them to be object-oriented, while missing all of the important concepts. Experience shows that beginning students often believe their work to be finished as soon as the compiler accepts their program. Equally, as soon as a C++ compiler accepts their code, they believe that they have written an object-oriented program. Of course, this might not be the case at all. The fact that a C++ compiler accepts non-object-oriented C programs as valid input, turns out to be a hindrance rather than a help in getting students to write object-oriented programs.

The requirement of a pure object-oriented language is a requirement that the language should force students to write all code in an object-oriented style. Classes should be the fundamental mechanism for structuring code and objects the basic runtime construct. No code should exist independent of a class.

3 *Safety*

The principle of safety is that errors that can easily be detected by the compiler or the runtime system should be detected. Furthermore, they should be detected early, and clear messages should be given about their cause. If error-prone constructs can be avoided altogether, they should be avoided.

While this statement sounds obvious, it is far from being the current state of affairs in some of today's most popular languages. An example of this principle is a good type system: the language should be strongly and, as far as possible, statically typed. Languages such as C++ and even Eiffel [Meyer 1992], which are not type safe, can cause fatal execution errors at runtime, which students (and often professionals as well!) find very hard to debug. Dynamically typed languages such as Smalltalk have a similar drawback: the point of detection of the error might be a long way away (in time and location) from the actual source of the problem. This makes understanding and eliminating program errors an unnecessarily difficult task.

Other examples of safety are checking of array bounds or checking for the use of uninitialised variables. Constructs known to be problematic should be avoided, where possible. Explicit pointers, for example, are known by every programming teacher as a source of major difficulties for students. Their use in a programming language can be avoided altogether – several programming languages do not require pointers to be explicitly dereferenced. Pointer arithmetic has no place in a first year programming course.

An argument often brought forward against this is efficiency. Pointer arithmetic saves time, checking array boundaries is expensive, and so on. The answer is: efficiency is *in our context* not of major interest. The issues of learning about efficiency and having an efficiently executing system are quite separate. "While students need to learn how to write efficient programs, efficiency is not itself a primary goal of their programs" [Tewari 1994, p321].

This is one of the key points of our argument: a teaching language does not have to meet all the criteria for an industry-strength production language. The requirements for teaching are different. It is possible to meet the teaching requirements better than it is currently done in existing languages if we are prepared to pay with other aspects, such as efficiency, that are not as essential for a teaching language.

4 *High level*

The programmer should not need to be concerned about machine internals. Tasks that can easily be carried out by the compiler or the runtime system should not be the responsibility of the programmer.

The most prominent example of a violation of this requirement is the explicit management of dynamic storage. Putting the responsibility for storage management into the hands of a programmer (especially if it is a beginning student) is unnecessary and leads to frustrating experiences. The problems with errors that are hard to find due to dangling pointers, double deallocation, or other forms of memory corruption are well known and form one of the hardest-to-debug groups of errors in programming in general. All of this can be avoided by using a system that provides automatic garbage collection.

The benefits of garbage collection are indeed so overwhelming that many modern languages use it even in a production environment. While in industrial contexts

there are arguments both ways (efficiency against productivity and correctness), for teaching languages the choice is clear. Explicit memory management poses problems for the student that distract from understanding the more fundamental principles.

5 *Simple object/execution model*

The model of execution should be simple and easy to understand. This includes the object model and/or the memory model. Many languages distinguish between, for example, two different kinds of memory layout for objects: objects on the stack and objects on the heap. Some languages require some objects to be explicitly allocated while others are allocated automatically. Some of these differences might be necessary internally. None of them, however, should be visible in the language itself.

6 *Readable syntax*

The syntax used should be easily readable and consistent. There are many reasons for this. First of all, a readable program is more likely to be correct. While readable syntax, of course, does not guarantee correctness, there are regular cases where errors are discovered in programs that are present only because a programmer did not understand another part of the code. If we assume that readability increases understandability, then the case for readability is clear. But does readability really increase understandability? The case might be different for beginners and experienced programmers, but we believe the answer is yes in both cases. Before we go further into this, let us state more clearly our view of the meaning of readability.

The most important aspect is that we favour keywords over symbols. Words are much more intuitive than symbols – carefully chosen keywords can reveal most of the semantics of many instructions. This makes the language both easier to learn and its programs easier to read. A good example of how the reliance on symbols can be detrimental to clarity is C++: in this language, great emphasis is put on having only a small number of keywords. The results are such bizarre constructs as the use of the word *static* at three different locations in the grammar with three unrelated meanings, or the use of the symbols “= 0” after a function header (suggesting an assignment!) to indicate deferred functions. None of this is clear to an uninitiated reader, and even programmers with some experience are often struggling to remind themselves correctly of the semantics of these constructs.

A readable language has many advantages for teachers and students. Teachers are often reluctant to learn a new language only to teach a new course. Programs of a well designed language can be immediately readable for anyone who is experienced in another language with similar concepts. Pascal, for example, is easily understandable for an experienced C programmer (but not the other way around). In this way, a readable language takes a burden off the teachers who have to learn the language themselves. For students it means that they can, after only a very short time, take educated guesses at the meanings of programs. Many people,

for example, when they get a textbook about a programming language, really read only the programming examples in that book. This should not be considered "wrong" – learning by example is a powerful way of learning that we all apply at some time. The easier the examples are to understand, the better it works. We should try to exploit and encourage this way of learning. Keywords also can be looked up in the index of a good textbook – an important point for a teaching language.

Another aspect of readability is consistency. The same syntax should be used for same semantics, different syntax for different semantics. The fact that in C, for example, a function definition and a function call can be syntactically identical serves no purpose other than to confuse programmers. On the other hand, the fact that C++ has references to variables as well as pointers to variables, which serve roughly the same purpose, but must be used syntactically differently, is hard to justify as well.

For both beginners and experienced programmers readability has clear advantages. It makes the learning of the language easier and helps to reduce the number of errors. Maybe a lesson should be taken here from the art of writing language as it has developed over many centuries. In the writings in human languages it has long been recognised that the fundamental aim for useful or aesthetically pleasing writing is in the effect on the reader, not the speed of the writer. This is true for all forms of writing, from technical manuals to poetry. The quality of the writing lies in the way that it is able to convey meaning to the reader – easy understandability for technical documents, emotional association for poems. In none of these cases would the writer dream of arguing that some of the words should be saved because the reader can still work out the meaning, and the writer has to type less. On the contrary: writing is done *for the reader*.

In an age in which we have recognised that programmers actually spend much more time reading programs text than writing them, and in which we agree that reading and understanding code may be more difficult than producing it, writing for the reader should become an accepted principle in computing as well. Gone is the time when programming can be considered as "writing for the machine". With improving compiler technology, producing a program understandable to a machine has become the easier part. Producing a program that is understandable to humans is the real challenge.

7 *No redundancy*

"No redundancy" means that for everything we want to do in the language, there should be one, and only one, way to do it.

Redundancy is often connected with flexibility and efficiency: having different constructs to achieve the same task is often useful to optimise code. These alternative constructs might differ in low level details such as performance, memory layout, etc. To write efficient code it may be essential to influence these low level details.

For beginners flexibility leads more often to confusion than to efficiency. Having three different mechanisms for the same thing, which differ in sometimes subtle detail, might pose problems for students when they cannot make a complete judgement as to the effects of their choice. An example is again taken from C++: several mechanisms exist to create an object. It can be created automatically, explicitly or with a "copy constructor". Some objects must be explicitly deleted, some must not. Some are in danger of being deleted twice, if the programmer is not very careful. The flexibility here is detrimental for first year teaching. Some language constructs that increase flexibility for experts increase confusion in beginners.

8 *Small*

The language should be as small as possible while including all important features that we want to discuss in the first year programming course. Use of larger languages like Eiffel or C++ usually requires the instructor to teach a subset of the language. We do not consider that a good solution. One obvious problem is that students use parts of the language that are not in the officially sanctioned subset (because they read examples from a textbook). The teacher then needs to explain why it should not be used, or accepts its use and is forced to deal with it.

With a small language students can reach the point at which they know all of the language constructs. This is psychologically very important. If students write their programs knowing that there are still several chapters to come in their text book introducing new, mysterious constructs, they always retain a degree of insecurity. With many problems they try to solve they cannot be sure whether they are really doing it the right way, whether there is not another construct just around the corner that would solve the problem in a much easier, more elegant way. We consider it important to get to a point where we can say to a student: "This is all. You have now seen all of the available constructs; from now on programming is not about learning new constructs any more, but about how to put them together." We believe that this allows a student to focus better on algorithms and design aspects, rather then spending too much time on specifics of many single constructs. Pascal and LISP are languages where this is possible, which is often a reason for their use in first year teaching.

Which particular language features should be part of a teaching language and which should not is open to debate and often not easy to decide. This question will come up again frequently over the following chapters, and we will present our opinion on this issue.

9 *Easy transition*

An essential aspect of any teaching language is that what is learnt by using it must be relevant to what is needed later.

The result of this argument is often that a language that is popular in industry (like C or C++ or, increasingly, Java) is used for first year teaching. This might not be the best choice.

Most institutions are not free of forces from the “real world” – we have to teach our students to program in a language that is relevant to industry. But it is a wrong conclusion that this means that we have to start teaching in C++. Most people would agree that, by the time students leave a university, they should be competent programmers in an industry-relevant language. This only means that students should end up with strong, say C++, skills, but not that they must start with it. On the contrary: We, as teachers, have to ensure that students learn *programming* as opposed to *a programming language*. We must make sure that we teach them the principles, and not just the details of one system that happens to be around at the time. Most people working with computers today, who left university more then five years ago, do not use the languages they learned at university. They use languages that became popular only after they finished their education (such as C++, Eiffel, Java, Visual Basic, etc.). It will be the same for our next students: most of the time they will not use the languages we teach them.

For us this means that we have to teach them the principles of programming. We might do this in any language we think appropriate. (See also [Knudsen 1988] for a good discussion of the difference between teaching programming principles and teaching a programming language.) On the other hand, they should not leave our universities without being able to use a current real-world language. This means that the language used for first year teaching should be relevant to current industrial languages. Concepts learned with the teaching language must be easily transferable to the next language.

If we think of designing a language especially for teaching first year students, then this means that we should not take a revolutionary approach. We should not try to do everything completely differently, or invent radically new concepts. We must ensure that we use language constructs that are relevant to other languages.

10 *Correctness assurance*

Many software engineering principles are now taught in first year programming courses. Among those are techniques such as programming by contract, which relies on the use of pre and post conditions, and correctness support through, for instance, class invariants.

The language used for teaching should support these as first class language constructs. While pre and post conditions, for instance, can be emulated in many languages (with assertions or comments) their existence as a separate construct much better reflects the importance that we now give them. Students take these techniques much more seriously if they are supported by the language rather than given in style guidelines.

11 *Environment*

The language must have a good graphical integrated program development environment to support it. The environment must hide details of the underlying operating system and allow students to focus on the programming task at hand.

One of the reasons, why teaching object-oriented programming has been difficult in beginners' courses is the complexity of the environment. Students often have no prior computing experience. To be able to use an object-oriented language, a student typically has to learn to use an editor, a compiler, and to execute a program. He/she has to be able to manage multiple source files, handle directory structures, manipulate (move or delete) files, etc. The effect of this is that we spend the first few weeks talking more about the environment than the language. Problems with existing environments fall into two categories: they are either non-existent (i.e. Unix or a DOS shell is used) or they are complex environments developed for professional software engineers. Both of these cause problems.

We would like an environment that, firstly, lets us concentrate on the programming language rather than the operating system, and secondly, supports the object-oriented paradigm.

The environment is, in fact, one of the most important points in this list of requirements. A suitable language can be unusable for teaching because of a lack of an environment. If we were to rate them in importance, the environment requirement alone would probably be as important as all the other points together.

Since the environment is so important, and since it is not at all clear what it means to have a "good" environment that "supports the object-oriented paradigm", we will discuss the requirements for the environment in much more detail below.

Some of the requirements listed above contradict each other. The inclusion of software engineering constructs such as pre and post conditions and class invariants, for instance, and the goal that the language should be small are in conflict. For a good teaching language it will be essential to strike a good balance between these conflicting goals. There will, of course, always be discussions about how small is too small, whether really *all* redundancy can or should be removed, and so on. For the design of a teaching language it is important to use these requirements as guidelines to design decisions and weigh the effects of different requirements against each other. They are not a straightforward recipe to create the perfect language, but they are a basis on which to judge the suitability of a particular language for first year teaching.

2.2.3 How simple should a language be?

All of the points above aim at making a language easier to use and avoiding unnecessary errors. A counter argument often brought against this is that students should encounter these difficulties so that they can learn to cope with them. A language, the

argument goes, should have explicit memory management, because students should learn to use it.

While we agree with the statement that students should learn about these things, we strongly disagree that it should be in the first year. The question is *when* students learn these things, and what they learn first.

In our view, students should first learn the important principles of programming with a clean language that supports those. They should then (after about one year) switch to another programming language where they can start learning all the details necessary for optimisation or low level programming. It is a question of what comes first. In our view, algorithms and correctness come before optimisation. We do not want to be forced to teach about obscure details before we introduce the principles. A small and simple language allows us to teach in the sequence we think best. It does not mean that all other aspects should not be encountered at all.

2.3 Requirements for the environment

Environments for teaching and the requirements for those environments are much less discussed in the literature than the languages themselves. It seems that, when academics argue for or against a particular teaching language, a lot of energy is spent on examining language features, and comparatively little on discovering benefits or drawbacks of the programming environment. The reason might be partly historic: for early languages there were no substantially different environments. If a department used a Unix machine, for example, it was clear that the environment would be the Unix shell.

This has changed considerably within the last decade. First of all, substantially different environments have become available. This development was to a large degree driven by the spread of integrated graphical environments on PCs. Today the same language can be used from a command line prompt or from within an integrated graphical environment, creating vastly different programming experiences.

Secondly, better environments have become necessary. Earlier introductory courses focused on the development of algorithms in procedural or functional languages. To do this, an editor and a compiler was all that was needed for the practical part of the work. Modern courses now use object-oriented languages and subject material taught includes testing, debugging and code reuse. This creates the need to deal with multiple source files and multiple program development tools from the very start. To give a beginning student a chance to cope with this increased complexity, better environment support is needed.

As object-oriented languages became more popular, developments in environment research and languages were brought together and integrated environments for object-oriented languages emerged.

In examining the requirements and the suitability of environments for our task, two aspects are of particular significance: support for object-orientation and suitability for teaching.

2.3.1 Environment requirements overview

We can group the requirements into seven categories:

1 *Ease of use* – The environment must be suitable for first year students. A student should not have to spend a lot of time learning how to use the environment. It should be intuitive and non-threatening.

2 *Integrated tools* – Important software development tools, such as the compiler, the debugger and a class browser, should be closely integrated into the environment.

3 *Object-support* – The environment should support classes and objects as its basis of construction. It should allow interaction with and manipulation of objects.

4 *Support for code reuse* – Code reuse has often been named as one of the primary motivations for object-oriented programming. To teach realistic and good programming practice, a system must enable both the use of library classes and the building of class libraries.

5 *Learning support* – Some techniques valuable to learning programming, such as visualisation, experimentation and interactive testing, should be supported.

6 *Group support* – Students working in teams on a common project must be possible.

7 *Availability* – The environment must be available at reasonable cost and must be executable on available infrastructure.

As above, we now discuss these requirements in more detail.

2.3.2 Environment requirements in detail

1 *Ease of use*

One of the most important factors in deciding about the suitability of a software development environment for teaching is ease of use. The environment must be easy enough to manage for inexperienced students to be useable for programming tasks after a very short time. This virtually implies that the environment must have a graphical user interface. The tasks that have to be managed by the environment internally are quite complex: file management, editing, compilation, management of compilation dependencies, debugging, browsing, testing, execution, etc. Even with this incomplete list it is clear that the available options are not trivial to

master. Relying on a textual interface in non-trivial systems has been shown to require a steeper learning curve, especially for inexperienced users.

Ease of use also means the hiding of unnecessary detail. Operating system details, for example, such as file management details, should be managed by the environment automatically, while letting the user work at a higher level of abstraction. When a user creates some classes, for instance, there is little interest in the number of files and in which subdirectories information about these classes are stored. In a good environment, it should be possible to operate for a long time without the need to be proficient in the use of the underlying operating system. This would avoid a common problem for first year courses: while the subject of the course is programming, the first two or three weeks are actually spent mainly on explaining operating system commands. This is necessary to get to the point where students know enough to use the computer system to type in, compile and execute their programs.

Most existing systems fail to meet our ease-of-use requirements. They typically suffer from one of two common problems: too little support or too much support. Both make them unsuitable for teaching first year students.

The first category is that of systems that rely on a loose collection of (at least partially) text based tools. They provide a command line interface (typically Unix or DOS) and little integration. An editor and a compiler are used as separate tools, which communicate through the user: the compiler uses line numbers to indicate locations of errors, and the user must locate the corresponding lines him/herself in the editor to correct the error. A rich set of file management commands is needed (listing, copying, deleting, etc.). Sometimes some of these tools provide graphical interfaces (an editor or a debugger). Nevertheless, the interface of the separate tools is typically different enough that the use of each additional tool is a major task that requires a significant amount of separate instruction and practice. The effect of this usually is that some tools, such as debuggers or class browsers are not used in the first year, because the time required to become familiar with them is not available.

The second category of available systems is that of very sophisticated integrated environments or CASE tools. These systems are often very powerful: integration between components can provide new functionality and a unified “look and feel”. The user interface is typically graphical.

The problem with this second category of system is that those that are available today are all developed for professional software engineers. They require experts to use them. Typically dozens, often hundreds, of functions are available to the user for many sophisticated tasks. For a professional software engineer it can be worthwhile to spend several months becoming an expert in the use of a powerful system, since it can pay off in increased productivity or quality of work over many years to come. For a teaching situation with beginners it is impossible. A system that is too powerful can be as unusable as a system that is too weak.

What is needed is a system that is simple enough to be useable for beginners after a very short time, yet powerful enough to provide many of the tasks of software development easily or automatically.

2 *Integrated Tools*

The requirement of tool integration is a direct result of the ease-of-use requirement. Above, we have already hinted at some advantages of an integrated system. The integration of tools can have many benefits:

- *A unified interface*. If the interface of all components appears in a unified way, the use of additional components is easier to learn.

- *Smaller interfaces*. Often, one tool can make use of an interface already provided by another tool. A debugger, for instance, can use the text display provided by an editor to mark the current execution position, or to let the user specify the location of a breakpoint. This results in fewer different interface components for the user to learn.

- *Increased productivity*. Integration can easily provide shortcuts for the edit – compile – execute cycle. If, for example, the compiler detects an error, it can open an editor window and highlight the corresponding location in the source code. The step of finding the line and column of the error can be automated. The same can be done for runtime errors if the execution of the program is integrated as well (e.g. via a virtual machine).

- *Better functionality*. Often, a tool can provide additional functionality by making use of information that was generated by another tool. An editor can, for example, provide language dependent functions (such as structure editing or pretty printing) by making use of the parser in the compiler (without writing a second parser, as is done in some systems). Even more interesting in our context is the automatic generation of class interface views in an editor by using symbol table information generated by the compiler. This goes far beyond the ability of a general purpose (non-integrated) editor, especially since it involves information from other files (such as inherited routines).

Systems that support the use of non-integrated tools have some advantages as well. The main one is the ability to use previously known tools for a common task, thus avoiding the need to learn to use a new tool. The most common example is the ability of an environment to let users use a text editor of their choice. Many experienced users have become very familiar with one editor and resent being forced to use another one.

A second advantage may be the availability of many general purpose tools for a system that is used as the basis for a non-integrated environment. Unix, for example, provides a large number of tools (e.g. for text formatting, counting words, printing, etc.) that might be used if a Unix shell is chosen as the development environment.

These advantages, however, do not outweigh the advantages of integrated systems. For beginners especially, the arguments for non-integrated systems are weak. First year students typically are not experts in the use of a particular text editor, and they do not know a relevant number of tools on a given platform. And even for more experienced users the disadvantages of an integrated system can be made less serious by providing a good environment: a text editor, for instance, that can be adapted to users' preferences with regard to key bindings, or a good set of tools which are easy to learn. The advantages of integration, on the other hand, benefit all kinds of users.

3 *Object support*

Many existing software development environments have evolved over time. They were originally developed for non-object-oriented languages and later adapted for object-orientation. This adaptation, however, usually fails to exploit the possibilities that come with object-orientation – they are, in their character, still structured (not object-oriented) environments.

The major shortcoming of these environments is the lack of support for objects. While the language is designed around classes and objects, the environment still manages files and produces a program (a single executable). In an object-oriented environment the user should be able to reason about, and interact with, classes and objects. The environment should represent the classes involved in a project and their relationships. It should also let the user create and interact with objects of any class. The result is the availability of important testing facilities: the programmer can create and inspect different objects, compare them, or pass objects as parameters to other objects.

The ability to create and test objects of any class without the need to write test drivers encourages incremental development. Classes can be tested separately from each other much more easily, and many debugging problems can be avoided.

Another aspect of object support is closely related to another requirement: learning support (discussed below). Direct interaction with objects can greatly improve understanding of object-oriented concepts.

4 *Support for code reuse*

The facilitation of reuse of existing code is one of the main motivations for object-oriented programming. In teaching, we must aim at giving our students a realistic impression of the task of programming. We must try to include as many real programming experiences into students exercises as possible, for students to understand the issues of software development and to form good habits.

Because of this, it is important to facilitate the reuse of existing code from the very beginning. The development environment must provide a class browser that lets a student find out about existing library classes. It should also have the ability to build new libraries of classes that the student has written. These can then be reused

later by him/herself or by other students. This is important to enable students to experience the need to write code for reuse.

5 *Learning Support*

The environment must support some techniques that are known to support learning of programming concepts. Among those are:

- *Visualisation*. The structures talked about when teaching object concepts should be, as far as possible, made visible on the screen. It is a common experience of teachers teaching object-oriented programming, that it is initially difficult for some students to think in terms of classes. The reason for this might be that all they ever see on the screen are lines of code. If students are expected to think and talk about classes, then the classes (and their relations) should be visually represented. The same can be said about objects at runtime: relationships between objects are better understood when made visible. We can even go further: if we expect our students to think about program design before coding, and if the tool we use to talk about program design is a graphical notation, then a student should be able to create this graphical notation in the environment as a starting point during the process of system development. We do not only require an environment to visualise existing code structures, but to *edit the visualisation* as part of system development.

- *Interaction / experimentation*. To provide a hands-on approach by enabling interaction with classes and objects can greatly increase the understanding of object-oriented concepts. The ability to experience the notion that, for example, many objects may be created from one class, and that those objects behave similarly, but have a separate identity and a different state, can greatly help in clarifying the relationship between classes and objects. Equally, the experience of stepping through code and seeing the effect of control structures and watching the value of variables change can contribute a great deal to the understanding of writing code.

It is interesting to note that the "learning support" aspects are not only useful in educational systems. Visualisation and interaction can greatly benefit professional software developers as well, since in non-trivial programming projects it is often necessary to "learn" something about the software artefact the programmer is manipulating. Visualisation of class structures is one of the most prominent features of many professional CASE tools.

6 *Group Support*

Another characteristic of programming in the real world is the need to work in teams. The success of object-oriented languages is partly due to their advantages in group projects. Ideally, we also want to teach our students about the techniques needed for teamwork. To do this, it is essential that the environment has some form of support for group work.

It should be possible for a group of students to simultaneously work on different parts of a system under development, and to have a controlled system integration process. Some form of group communication mechanism would also be beneficial.

7 *Availability*

Some of the requirements mentioned so far are implemented in existing CASE tools. There is, however, one serious problem with some of those tools: cost. Being developed for professional use with large companies as customers in mind, the pricing of some of those tools is outside the reach of many university departments. In addition to that, some of them require very sophisticated hardware (either very fast machines or large memory). In order to be useful as a university teaching language, a system must be available at reasonable cost, and it must be able to run on commonly available hardware. "Commonly available hardware" can mean different things: ideally, it would be available on different platforms. It should be available for Intel PCs, preferable under MS-Windows, since this is the most commonly used platform by students. If it is available for Unix to be run in a multi-user environment with terminals, then the resource requirements of the system (processor time and memory) must be such that a typical university machine can serve at least several dozen terminals concurrently. It is, of course, unclear what "a typical university machine" is, but we can at least say that a system that requires the latest, fastest system on the market will not be of use to many institutions.

None of the requirements named above is really new. Each of them has been implemented in some existing system. (Existing systems will be examined in chapter 4.) The combination of these requirements, however, is what is really important in our context. In particular the combination of the requirement of ease-of-use with that for fairly sophisticated technical support may at first glance seem contradictory. We are asking for a powerful system that appears simple to the user. The degree to which this combination is achieved will be the determining factor in assessing the suitability of systems for first year teaching.

2.4 Summary

We have, in this chapter, discussed the characteristics that an object-oriented teaching language and environment should have. We have seen that the requirements for a teaching system are different from those of a professional production system, and that those of an object-oriented system are different from a procedural one.

As mentioned earlier, we believe that no existing system meets the requirements. No existing system is, therefore, ideal for teaching. Furthermore, we believe that no existing system is *suitable* for teaching, since each of them fails to meet the requirements in substantial ways.

In the next chapter we proceed to examine existing languages and environments in more detail. As we discuss each system, we will substantiate this claim with concrete examples.

3 Languages

In this chapter we examine the most commonly used object-oriented languages. Many papers have been published comparing different aspects of a variety of languages (e.g. [Blaschek 1989, Henderson 1993, Kristensen 1996, Schmidt 1991]). Most of those, however, concentrate on technical aspects of the languages themselves, rather than assessing their suitability for teaching. Mazaitis [Mazaitis 1993] has compared several attempts to incorporate object-oriented programming into the curriculum and comments on experiences with languages and environments used. She writes in her summary: "However, the experiences of teaching the courses revealed common problems: difficulties with the language chosen, inadequacies in existing support tools, and the amount of time students need to become proficient with a new paradigm, environment, language and set of tools." In this chapter we assess each language and identify the potential difficulties in relation to teaching.

When comparing languages, there is always the question of how many and which languages to look at. Since language comparison is not the main topic of this work, it is not our aim to survey as many languages as possible. We want, however, to survey the *important* languages. So which languages are important?

The languages that are important in our context are modern or mainstream object-oriented languages. We will not examine procedural languages, since one of our premises was that we want to teach object-oriented programming. This places languages like *C* or *Scheme* outside our area, although some people elsewhere still argue for their use in teaching.

But even restricting ourselves to object-oriented languages, there are too many to consider them all. Countless languages with small follower communities exist (e.g. *Dee* [Grogono 1994a] or *Turing* [Holt 1988]), and many new highly interesting experimental languages are being developed all the time (e.g. *Self* by Sun Microsystems [Ungar 1987] or *Dylan* by Apple Computers [Apple 1994]).

So the first real question is: How do we select the languages to survey?

Our decision was to take advice from work already completed by other people: languages that other institutions have chosen for their first year teaching should be worth considering. Many people have thought about this question before: Which language should we use for teaching in first year? A language chosen by one or more other institutions must have something speaking in its favour. Languages that were not selected by any institution should also not emerge as a winner if we repeat the examination. So we assume that we can safely omit them from our survey.

The languages that we, according to this process, found to be the important ones are *C++*, *Java*, *Eiffel* and *Smalltalk*. Each of these has been examined in detail for their suitability for first year teaching by being evaluated against our requirements. This chapter gives a summary of these evaluations.

3.1 C++

C++ [Stroustrup 1991] is fast becoming an industry standard and probably the most popular object-oriented language (although Java is gaining ground quickly). Countless books have been written about C++ and many implementations are available. There are several conferences and journals exclusively aimed at the discussion of C++ related topics.

C++ has the advantage of being very industry-relevant. It is currently one of the most important languages for a graduate of a computing degree to be familiar with. It is adopted in industry projects because of its flexibility, efficiency and wide availability.

1 *Clean concepts*

Whatever can be said in favour of C++, it is not that it represents any abstract concepts in a clean way. Two of the most influential design decisions in the development of C++ were to retain full backwards compatibility with C and to regard efficiency in time and space as a main goal. (Stroustrup stated that "time and space overheads above those for C are considered unacceptable for C++" [Ellis 1990].) Both those characteristics have the effect that the representation of many important language constructs is heavily influenced by low level considerations that distinguish the concrete representation from the abstract concept. Often, subtle implementation details need to be understood to use fundamental constructs correctly.

2 *Pure object-orientation*

C++ is a hybrid language that supports object-oriented programming as well as non-object-oriented programming, leading to the temptation to develop solutions that are not really object-oriented. This is particularly a problem for those already familiar with C, which is the case for an increasing number of students entering computer science courses. Learning object-oriented concepts becomes more difficult because the language does not enforce or encourage their use.

3 *Safety*

One of the most serious criticisms of C++ is its lack of type safety. Sakkinen [Sakkinen 1992] states that "almost all its higher level constructs and protections can be corrupted and circumvented at will by low-level manipulations" and comes to the conclusion that "the C language is so unsafe that striving to a total or almost total upward compatibility from C cannot result in a good general-purpose object-oriented language." Although C++ has attempted to improve some of the problems often criticised in the type system of C, it remains a weakly typed language that features numerous constructs with the potential to easily circumvent the type system.

The explicit dynamic storage allocation in C++, in combination with the lack of garbage collection, also greatly increases the risk of errors. Often C programs that have been tested and used for some time have "memory leaks" and other bugs that are caused by improper storage handling. In C++ this problem is more serious, since deallocation is often associated with a function call. A missing deallocation omits this function call and this can cause a much greater variety of unwanted effects than just disappearing memory.

4 *High level*

C++ includes numerous constructs for low level manipulations. Bit operations and pointer arithmetic, for instance, are frequently used in typical C++ programs.

The use of explicit storage management, mentioned above, not only allows but forces the programmer to think at an unnecessarily low level.

5 *Simple object/execution model*

The object model supported by C++ is clearly the most complex of the languages included in this survey. This is caused in part by C++'s support for pointer and non-pointer ("automatic") variables, as well as the semantics of some important constructs.

This point can be illustrated by an examination of the object creation mechanisms in C++. There are three different methods of object creation – automatic, explicit, or by assignment – which all differ in minor aspects. In the first case objects are destroyed automatically, including an automatic call to the destructor function; in the second case objects are not automatically destroyed; in the third case the objects are created without a call to the constructor function and calling the destructor can be a problem (but it may be called automatically, anyway). The subtleties of this are extremely difficult to explain to students.

An example of overly complex operation semantics is the definition of the dynamic dispatch mechanism. Functions are dynamically dispatched only if the variable through which the object is accessed is a pointer variable and the operation called has been explicitly defined as a dynamically dispatched function ("virtual func-

tion" in C++ terminology). This definition is extremely confusing at times. Combined with the fact that it is permitted to overload a non-virtual function, this definition opens up a whole range of possibilities for errors and misunderstandings. The flexibility inherent in this construct has no benefit in a teaching situation, since the justification of its existence consists exclusively of efficiency reasons.

6 *Readable syntax*

Readability is another negative point for C++. Its syntax is overly terse and has as one of its most obvious features its favouritism of symbols over keywords. This is partly caused by the philosophy of C, which tried to avoid a large number of keywords, (a philosophy that has been carried over into C++) and was then maintained through the goal of backwards compatibility: Since language keywords are not syntactically distinguished from user defined identifiers, they live in the same name space as identifiers. So every addition of a keyword potentially renders a set of existing programs illegal.

As a result of this, C++ often overloads the same keyword for different purposes or uses unintuitive symbolic constructs. An example is the definition of deferred functions (functions declared but not defined in a class, which must subsequently be defined in a descendant class). These are called "pure virtual functions" in C++. The syntax in C++ is:

```
virtual void f () = 0;
```

By employing a syntax similar to variable assignment, the meaning of the construct is hidden and the declaration becomes unreadable to a non-C++-programmer. Even experienced programmers in other object-oriented languages would have difficulty understanding the meaning of this construct. The simple replacement of "= 0" with a keyword such as "deferred or "abstract", would result in a reader familiar with object-oriented concepts understanding the code.

There are numerous similar examples in the language.

7 *No redundancy*

The decision to keep C++ upwards compatible with C has led to the evolution of a large number of redundancies and surprising interactions of concepts, and greatly increases the complexity of the language.

In many cases several different language constructs exist for the same semantic concept and these sometimes differ in subtle ways. This makes reading as well as writing C++ an unnecessarily difficult task. One example has been given above when we discussed the object model: redundant constructs exist for object creation.

Another example of confusion introduced through redundancy are the differences in relation to reference arguments as compared with constant pointer arguments. Consider the following two declarations:

```
void function1 (int *const arg);
void function2 (int& arg);
```

Both declarations serve the same purpose and behave in similar ways. When calling the functions, however, a different syntax must be used:

```
function1 (&i);
function2 (i);
```

In this example, we have two constructs to accomplish the same task, but with different syntax. This is clearly a violation of good language design principles.

8 *Small*

Because of its numerous redundancies, C++ is not a small language. In addition, many constructs interact in subtle ways which make it necessary to learn many details and dependencies of constructs (e.g. the interactions of arrays and default constructors).

9 *Easy transition*

Transition is not an issue for C++. Since it is an industry-relevant mainstream language no transition is necessary.

10 *Correctness assurance*

C++ allows the programmer to write correctness assurance constructs through “assert” statements and comments. Pre and post conditions are, however, not explicitly supported by the language, so it is left to programmers’ discipline to make consistent use of this technique. For a teaching language this is not a desirable situation.

11 *Environment*

Numerous environments are available for C++. They range from simple command line shells with stand-alone compilers and editors to graphical integrated environments. Many vendors offer compilers and environments, and several free versions are available.

The list of problems mentioned here could easily be extended. Many other problems have been pointed out and discussed in much detail in the literature (e.g. [Joyner 1996, Pohl 1988, Sakkinen 1992]). Detailed discussion of all of them would exceed the space we want to devote to this language.

Overall, C++ is one of the worst candidates for our needs.

The problems associated with C++ are widely reflected in comments of teachers who have tried to teach C++ to undergraduate students (e.g. [Berman 1994, Mazaitis 1993]). While the adoption of C++ as a first programming language was rapidly increasing in 1994 to 1996 (especially in the United States, greatly helped by the selection of C++ as the language to be used nationwide for the "advanced placement" test of final year school students [Abelson 1995]), it has since declined. Warnings and experience reports seem to have led to a greater reluctance to use C++ in introductory courses. When the GI ("Gesellschaft für Informatik", German professional computer science society) recently published a special edition of its journal "Informatik Spektrum" discussing languages for teaching object-orientation [GI 1997], C++ was not seriously considered anymore. Almost every one of the six main articles in that journal commented on the weaknesses and difficulties of C++ in this context.

To provide a somewhat more balanced view, we should also mention that not all people share the view of C++'s inherent problems for teaching (e.g. [Biddle 1994, Decker 1994]). Numerous papers have been published with tips and tricks for teaching C++ in a certain way, so that specific problems may be avoided, overcome or at least lessened. D'Souza [D'Souza 1995], for example, suggests that it is not harder to learn C++ than Smalltalk, if it is taught properly (but he agrees that it is harder to teach a good C++ course). This is, in our experience, certainly not true. Personal experience with teaching C++ has shown an incredible number of examples of cases where students had severe problems in completing their work because of errors caused by ambiguities, syntactic quirks or semantic pitfalls that do not exist to that extent in any other language included in this survey.

Although the above discussion of problems with C++ is not exhaustive, it is obvious that the issues raised disqualify C++ as a candidate for a first year teaching language.

3.2 Smalltalk

Smalltalk [Goldberg 1989] is sometimes referred to as "the most object-oriented" language. It carries object-orientation further than other languages. Smalltalk states that "everything is an object" and very consistently sticks to this rule. Even control structures in the language are considered objects. This consistency makes it attractive as a teaching language.

1 *Clean concepts*

Most fundamental concepts of object-orientation are supported in a clean and consistent manner. Smalltalk adds almost no overhead in its concrete representation that is not reflected in its abstract model.

Some constructs, though, which are now accepted as being important to object-orientation, are not well represented in the language. An important example is the poor support for implementation and interface distinction. In Smalltalk, all methods are automatically public and all data automatically private. This,

combined with the lack of special constructor functions and an encapsulation model that prevents class methods from accessing instance data, make it necessary to write initialisation methods as public operations, even though they should never be publicly called. They are typically tagged with a comment declaring them private, and it is hoped that clients follow this recommendation. The same problem occurs in other situations.

This lack of support for an important concept is understandable in the light of Smalltalk's age – at the time of its definition not much experience with object-orientation existed. It is nonetheless unfortunate in the context of a teaching language.

2 *Pure object-orientation*

Smalltalk is a pure object-oriented language. It enforces the development of code in an object-oriented style and, consequently, the programmer must adopt an object-oriented way of thinking about problems and solutions. Because of the extension of the object concepts to language elements themselves (e.g. classes are objects) it can be considered "purer" than other languages.

3 *Safety*

The main drawback of Smalltalk is its lack of static typing. Sakkinen [Sakkinen 1992], in a discussion of programming languages, distinguishes languages for "exploratory programming" and languages for "software engineering". The former aim at great dynamism and run-time flexibility, the latter have static typing and other features that aid verifyability and/or efficiency. It should be clear from the discussions in preceding chapters that we aim to prepare students in the programming course for a software engineering view of the world, thus favouring languages of the second kind. Smalltalk clearly is a representative of the first group. It was developed with a requirement in mind that the user be able to rapidly change the application structure [Goldberg 1995a].

Smalltalk is not statically typed and, as a result, type errors will not be detected until run-time. In the worst case, errors may not be detected at all, if, for example, the program is not thoroughly tested and not regularly used (as is often the case with students' assignments). When an error is detected, it can be very difficult to locate its cause. Because of the dynamic nature, the time and location of detection of the error can be far removed from its actual source. (This, of course, is also possible in statically typed languages, but the chance of it happening is far greater with dynamic typing.) The error messages in those cases are necessarily less precise and less helpful. There is a high probability that students will be confused and unable to understand the problem given their typically poorly developed debugging skills.

4 *High level*

Smalltalk consistently operates at a high level. It provides automatic garbage collection and hides low-level details.

5 *Simple object/execution model*

The execution model is clear and consistent. All objects are accessed via references, all methods are dynamically bound and operations behave in a uniform manner.

6 *Readable syntax*

Smalltalk's syntax is another characteristic, which we view as a drawback. It is alternately referred to as "simple" and "obscure". Both are true in some sense. It is simple in that it is composed of only a few syntactic concepts and structures. Thus, it is simple in the same sense in which LISP syntax is simple. In another sense, Smalltalk's syntax is obscure. This, unfortunately, is the sense that is important to our discussion: readability. The unification of concepts in Smalltalk, the view, for instance, that even control structures are objects, results in a syntax that is less readable and less intuitive than the ALGOL-style syntax adopted by many other languages (e.g. Pascal, Modula, Eiffel).

Although syntax is to a great extent a question of experience (and thus the argument that "they just have to get used to it" is often brought forward), we should not ignore two certain aspects of experience. Firstly, many students entering university courses today have prior experience with a programming language, and we would be foolish not to try to build on this experience. Secondly, we want the experience gained by students in our course to be as helpful and relevant to future work as possible. In both respects, adopting an ALGOL-like syntax seems to be beneficial.

7 *No redundancy*

Smalltalk has a very small set of language constructs (most operations are represented through method calls) and avoids redundancy.

8 *Small*

The size of Smalltalk can be viewed in two ways. The language itself (in terms of the number of constructs) is small; the Smalltalk system, however, is large. Smalltalk usually offers a huge class library. Since everything in Smalltalk is an object, extensive use must be made of the library from the very beginning. Virtually all publications about teaching Smalltalk, although generally positive about the system as a whole, report problems coping with the size of the class library. Tempte [Temte 1991], in a paper describing his experiences with teaching Smalltalk, writes: "It is easy to underestimate the difficulty of learning to use Smalltalk effectively. Smalltalk is not an isolated language but a programming

system which uses very simple language syntax ... in conjunction with a very large class library within an interactive environment. Software development requires facility (sic) not only with the object-oriented paradigm but also with the library and the environment." He goes on to state that more time than expected had to be devoted to teaching about the structure of the library. Similarly, Skublics [Skublics 1991] reports that a survey of students after a course using Smalltalk indicated that they found the existing class library overwhelming; LaLonde and Pugh [LaLonde 1990] report students to be "more apprehensive" because of the "sheer amount of code provided by the Smalltalk library" (although they were very positive about the use of Smalltalk in general).

We stated earlier that the teaching language should be small, so that the students do not feel overly intimidated or lost in the programming system. This requirement must also include those class libraries which students necessarily encounter. We do not at all argue against the use of class libraries in general. On the contrary: we think that experience with the use of library classes is essential to foster a culture of good programming practice and software reuse. But similar criteria to those which we apply to the evaluation of the language proper must also be applied to the libraries used in the first year course. In this respect the Smalltalk environment is overwhelming. Because of the nature of the Smalltalk system, a considerable amount of time must be spent getting acquainted with the structure of the class library. A related criticism is the complexity of the tools provided in the environment. The feeling of being lost in the system is as often caused by complexities of the class browser as the library itself. These issues will be further discussed in chapter 4.

9 *Easy transition*

Transition to other object-oriented languages may, on the surface, seem less than ideal, since Smalltalk's syntax and class model is considerably different from most other languages. On the more fundamental issues of object model and object-oriented techniques, however, Smalltalk provides a clear picture that is easily transferable to other systems.

10 *Correctness assurance*

Smalltalk does not provide any support for pre and post conditions or class invariants.

11 *Environment*

It must be counted in favour of Smalltalk systems that they always include an integrated development environment. Even though we criticise it as being too complex in some parts, it provides many aspects that are clearly helpful and desirable in a teaching environment. The manner of object interaction, for instance, provides an immediacy for experimentation and testing that goes far beyond that provided by most other programming environments.

Smalltalk's view, on the other hand, of programming as modification and extension of a global universe which merges the view of libraries and the current application, often leads to confusion. This aspect is further discussed in chapter 4.

3.3 Eiffel

Of all the languages surveyed, Eiffel [Meyer 1992] probably comes closest to fulfilling our language requirements. It is statically typed, supports object-oriented concepts in a clean way, avoids redundancy and has a clear, easily readable syntax. This combination makes it a better candidate than any other language.

Comments by teachers who have used Eiffel in university courses reflect this view. The reaction is generally positive as far as the language itself is concerned, with a list of minor criticisms [Nørmark 1995, Ryba 1997].

1 *Clean concepts*

Eiffel represents most concepts in a clean way. There are some exceptions though. Eiffel has, for example, two different storage modes for objects, the normal (reference) mode and "expanded" (value) mode. This is unfortunate from a pedagogical point of view. Its only purpose is to increase runtime efficiency, with no justification at a conceptual level. The effect of supporting these two modes is that programmers are forced to think about implementation details during class design (since some designs, such as self referencing structures, can only be implemented with one of the alternatives).

2 *Pure object-orientation*

Eiffel uses object-concepts throughout and does not support hybrid programming.

3 *Safety*

The system is mostly type safe. Eiffel is statically typed and most errors are detected at compile time. Some typing holes exist which introduce the possibility of fatal errors at runtime, but these cases are rare enough not to pose a serious problem for a first year teaching context.

4 *High level*

Automatic garbage collection is a standard feature of Eiffel systems, avoiding one of the worst areas of low-level operations (manual memory management). Some low-level operations exist, but they are unobtrusive – programmers are not forced to deal with them. Eiffel can be considered as fulfilling our "high level" requirement.

5 *Simple object/execution model*

Eiffel's object model is not as simple as would be desirable. It is unnecessarily complicated through the use of two different storage modes (discussed above).

Its attempt to support a very large number of constructs sometimes leads to overly complex structures. Nørmark [Nørmark 1995] points out an example: the common idiom in which a redefined routine in a subclass calls the original routine in the superclass. To do this in Eiffel, it is necessary to inherit the superclass twice (!), redefine the routine in one inherited version, rename it in the other and select the first one. The fact that a technique such as repeated inheritance is necessary for such a common pattern is extremely unfortunate. This use of inheritance does not fit well with the main purpose of inheritance as specialisation which we try to convey in first-year courses.

This example shows that the complexity of the language cannot be ignored by teaching a subset[1]. As Meyer himself put it [Meyer 1992, p500]: "... the idea of orthogonality, popularised by Algol 68, does not live up to its promises: apparently unrelated aspects will produce strange combinations, which the language specification must cover explicitly". Advanced features of the language (repeated inheritance) affect the appearance or behaviour of other features (superclass calls).

6 *Readable syntax*

Eiffel's syntax is based on the tradition of ALGOL and Pascal and makes extensive use of readable and meaningful keywords. Like other languages in this tradition, it is easily readable and enforces a very clear code structure. For a teaching language, this is an ideal style.

7 *No redundancy*

Redundancy is mostly avoided in Eiffel. In most cases there is one well defined way of using a technique.

8 *Small*

While Eiffel mostly avoids multiple constructs for the same concept, it supports a large number of concepts. This makes Eiffel large, even though it has little redundancy. It contains numerous constructs which cannot be included in a first year course. The way to deal with this problem would be to teach a subset of Eiffel – a solution which we have found to be undesirable (see discussion of principle 8, section 2.2).

---

[1] Eiffel is complex in a different way than C++. C++ is redundant. Complexity often arises from too many different ways to achieve the same thing. Eiffel is not redundant, just large – it covers many constructs and techniques. Eiffel's complexity is therefore justifiable, since it has a purpose. Nonetheless it is undesirable for introductory courses.

9 *Easy transition*

Eiffel defines its constructs (small scale constructs such as loops as well as object-oriented concepts) in a clear way close to accepted mainstream views. The skills learnt with Eiffel should be easily transferable to many other languages.

10 *Correctness assurance*

Sophisticated support for pre and post conditions as well as class and loop invariants is included in the language. Design by contract is supported better than in any of the other languages surveyed.

11 *Environment*

The most fundamental problems with Eiffel are not with the language itself, but with the libraries and the programming environment. The libraries are far too extensive and too specialised for an introductory course. They are described by teachers and students as overwhelming and difficult to navigate [Mazaitis 1993, Nørmark 1995]. The collection class library, for example, is extremely hard to understand and difficult to use. The linked list class, which will be used in many first year projects, has more than 50 interface routines!

The most severe problem is the environment. Currently, only a handful of Eiffel implementations exist. Of the few that are available, even fewer include a graphical development environment. The most sophisticated and most widely used Eiffel environment is “EiffelBench” from ISE. But even this environment, the most promising for our purposes of those available, is unsuitable for use by first year students. In a report about using it in a fifth-semester course, Nørmark [Nørmark 1995] described it as low in quality and unreliable. He describes numerous severe problems with the environment: compilation is too slow (even though three different compilation modes are provided), the different compilation modes (which are only meant to save time) are inconsistent, and it is difficult to use. In a student survey, 37% of the students said that they chose not to use it (and used a Unix editor and shell instead). Of those who used the graphical environment, 64.7% found the experience “negative” or “very negative”, 35.3% judged it as “neutral” and no one (0%) described it as “positive” or “very positive”. Nørmark concludes that it “is not good enough for teaching purposes”.

The problems with the environment are not only implementation deficiencies which will be solved once reliability is increased. Many problems lie in the design of the environment itself. This issue will be further discussed in chapter 4.

Considering the importance we attach to the environment of an object-oriented language in a first-year course, this is a serious problem. The environment is needed to hide some of the complexity inherent in the implementation of object-oriented technology (such as the need to deal with multiple files) from the student. The Eiffel environment, on the other hand, adds complexity. Eiffel (at least as it is currently available) must therefore be considered unsuitable. Unfortunately, since

the language is large and not easy to implement, it is unlikely that better alternative implementations will become available in the near future.

3.4 Java

Java [Arnold 1996] is very quickly becoming one of the most popular object-oriented languages on the market. Never before has a language spread so quickly and been taken up by so many programmers in such a short amount of time. (Never before has a language been pushed with such an immense organised marketing effort, either!)

Unfortunately, the reasons for its popularity have as much to do with the marketing push and with coincidental side aspects as with the quality of the language itself. Three aspects, among all others, can be identified that lead to the great success of Java:

1 Java was the first language that was integrated into web browsers and could thus be used to write “applets”, programs running in a web page. With the boom of the world wide web at the same time, Java became popular for all things connected with the web.

2 Java translates to machine independent byte code, which is then interpreted. This principle is far from being new, but since Sun has managed to convince virtually all operating system vendors to include Java virtual machines in their system, instant platform independence can be achieved. This coincides with a situation where more than enough processing power is available for many applications even in common desktop PCs. Thus, developers are starting to consider platform independence (implying cost and time savings) more important than maximum possible efficiency.

3 Java has cleverly been marketed as “a better C++”. It uses syntax similar to C++, and has been compared to it from the very beginning. While this marketing approach is misleading, since Java is fundamentally different from C++ in the design of just about every interesting aspect of object-oriented programming (and, from a language design point of view, closer to languages such as Modula-3), it has greatly helped its acceptance. Firstly, compared with C++, Java really does look clean and simple. (But then again, so does just about any other language.) By managing to be compared only to C++ most of the time, Java has managed to present itself clearly as the better choice. Secondly, because of its apparent similarity to C++, many more programmers are willing to learn to use the language. Generally, programmers are very reluctant to switch to new languages. Few have the time and the energy to keep up with the developments as new languages are published year after year. The fact that the low level constructs are similar to C and C++ made many people believe that they know half of the language already, and learning the rest would be an easy task. (This might or might not be true – generally the syntax and low level constructs, such as loops and conditionals, are not the most difficult thing to learn in a language.)

In a more sober analysis Java still looks like a well designed language, but not like the final solution to all our problems, as the hype makes one believe.

1 *Clean concepts*

Many of the important concepts of object-orientation are represented in a fairly clean way. On the positive side is the handling of multiple inheritance: Java supports separate type inheritance through a concept called *interfaces*. It allows only single class inheritance, but multiple type inheritance. This is a very good compromise which allows the programmer to use much of the functionality provided by multiple inheritance while avoiding most of its problems.

On the negative side are numerous small problems. Simple types, for instance, are not regarded as classes. This sometimes causes problems, so a second type – a class type – is defined for each simple type. As a result, there are types *boolean* and *Boolean*, one of which is a class and the other is not. The duality exists for all the predefined scalar types.

The language forces some of the more advanced concepts into the foreground very early. This is most prevalent in the very first function: the *main* function that every application needs to provide. The required signature for this function is

```
public static void main (String[] args)
```

There are several concepts used in this line that a teacher might not want to introduce at this point: static functions, the *void* return type, array parameters. Yet this is necessarily the first code that students see. (It gets worse if the class uses input: then an exception declaration has to be added to the signature.)

So while each concept on its own is well represented (except for syntactical aspects, discussed below), the fact that they cannot be initially avoided introduces problems.

2 *Pure object-orientation*

Java is basically a pure object-oriented language. Used properly, all code is part of a class, and classes are the basic unit of code structure. The notable exception is again the main function. It is declared static (making it a class function), although it is logically not part of the class, but used to start up the application. As long as the main function is only used to initiate the execution, the application is purely object-oriented. Unfortunately, some text books start by introducing small programs entirely written inside the main function. This example has nothing to do with object-orientation and should be avoided.

3 *Safety*

Java is type safe with a combination of static and dynamic type checks. Most constructs are statically type-checked with some constructs being checked at runtime, potentially throwing exceptions.

4 *High level*

Most language constructs operate at a high abstraction level. Most importantly, Java provides automatic memory deallocation through garbage collection.

One commonly supported concept in object-oriented languages that is missing in Java is support for genericity. Genericity (sometimes called parametric polymorphism) is the ability to instantiate class patterns with type parameters so as to easily generate a family of classes that share their code and functionality. It is a powerful mechanism that is essential in the construction of a good collection library. The collection library, in turn, is essential for first year programming to provide interesting examples early in the course, and to encourage reuse. Collections, such as lists or bags, are typically among the first library classes used by students in their programs. They are ideal for experiencing the benefits of reuse, since they provide important functionality to students in an easy way, allowing them to write programs that they would not otherwise be able to write.

The lack of genericity affects students in two ways: firstly, they cannot learn about this important concept, which they will later encounter in other languages. Secondly, the collection library (implemented with inheritance instead of genericity) does not provide static type safety. When implementing collections with inheritance, the element type of the collection is typically defined to be the common superclass of all objects, called *Object* in Java. This mechanism has the disadvantage that it does not guarantee monomorphic collections – that is that all elements in any one collection are of the same type – and makes it necessary to explicitly cast the type back to the original before elements can be used again.[2]

5 *Simple object/execution model*

Java's object model is reasonably simple and straightforward. The restriction to storing all objects by reference greatly simplifies the language. Some more obscure constructs exist (such as static initialiser blocks, code written in a class, but outside any function), but they can easily be avoided. The inheritance mechanisms are well thought out and elegant.

6 *Readable syntax*

The syntax is a weak point in Java. It uses C/C++ syntax which we have already criticised above. This syntax is known to inhibit readability in several ways. Firstly, its favour of symbols over keywords, which allows a very terse program text, makes it unintuitive and harder to read. Secondly, the flexible style of this syntax (its mix of statements and expressions, for instance) leads to the

---

[2] These remarks apply to the Java collection library at the time of writing. Java is still very much in a definition phase, and it is likely that the definition of the standard collection library will be changed. The elegance and simplicity of the solution using generics, however, cannot be achieved with any definition relying on inheritance instead.

development of distinct coding styles. Each style may be valid and consistent, but maintenance (the need to read other people's programs) is negatively affected. It reintroduces the need for style guidelines. The fact, for example, that it allows an arbitrary mix of public and private definitions in a class does not serve a useful purpose. Its effect is that the interface of that class may be scattered throughout the whole text, making it unnecessarily hard to read. Java also retains some of C++'s more obscure keywords. The use of the term *static* to declare class methods, for example, is based on a technical detail and is unintuitive at a higher level.

A more detailed description of the problems associated with the C style syntax can be found in [Joyner 1996, Pohl 1988, Sakkinen 1992].

7 *No redundancy*

Java generally avoids redundancy in its constructs. It retains, however, some redundancy in the low-level operators which it copies from C++. One example is the provision of four ways to increment a counter:

```
count++;
++count;
count += 1;
count = count + 1;
```

The reason for the existence of these constructs is that it makes code optimisation easier for certain architectures. Compiler technology has come a long way since this was really an issue: today good code optimisers can generate identical code from all four of those statements.

8 *Small*

Java includes several advanced concepts and constructs which prevent it from being a small language. They include separate type inheritance, support for threads and synchronisation, exceptions and nested classes. People will argue about whether this is an advantage or a disadvantage. Some of these (e.g. type inheritance as mentioned above) can be useful. Others might be too advanced for first year teaching. Generally, teachers will not always agree on which of these concepts should be covered in the first year and which should not. A first year course will not normally have the time to cover all of them, so teachers will necessarily have to fall back on teaching a language subset.

9 *Easy transition*

Java itself is quickly becoming an industry relevant language, so transition to another language might not be as important as with some of the other languages. Several universities currently teach C++ after about one year of Java, arguing that the syntactical similarity makes the switch easy. Whether the transition to C++ really is easy or not is an open question. On the one hand, the low level syntax similarities allow very direct application of already acquired knowledge. On the other hand, Java uses the same syntax for some constructs with different semantics.

Here, the use of similar syntax may be more misleading than helpful. Not much experience with this issue has been gathered and published yet.

10 *Correctness assurance*

Pre and post conditions are not supported in Java. This, together with the lack of support for genericity, is the most unfortunate omission in the language.

11 *Environment*

The most commonly used environment for Java is the Java Development Kit (JDK), provided free of charge by Sun Microsystems. It provides a compiler and a runtime system with a very basic command line interface.

Many other environments have been developed now, or are under development. Prices, quality and availability vary greatly – these issues will be discussed in the next chapter.

In addition to the characteristics discussed above, Java's considerable popularity must be counted as an advantage. It provides great motivation for students.

Early reports of teaching Java (e.g. [Allen 1998, Goedicke 1997]) are mostly positive (although, because of comparison mostly to C++, of limited scope). Some problems are repeatedly mentioned [Allen 1998, Clark 1998]: user interaction, especially text-based input, is too confusing; students have to deal with pieces of code that cannot be explained (at that stage), leading to discouragement and apathy; the relationship between simple types and their object equivalents (e.g. `int` and `Integer`) causes problems; and exception handling is obtrusive and cannot be initially avoided.

Overall, Java is not a bad language for teaching (just as Smalltalk and Eiffel are not bad languages either), but it has significant shortcomings that prevent it from being an ideal teaching language.

3.5 Other languages

Some other languages that are sometimes named as candidates for teaching might be worth mentioning. These languages did not make it into our shortlist and were not analysed in detail. Here we present some reasons why they were not considered more seriously.

Beta

Beta [Madsen 1993] is a very modern object-oriented language with interesting language constructs. Its outstanding characteristic is the use of a single syntactic structure (the "pattern") for the representation of a wide variety of concepts. Routines, classes and processes, for example, are all represented by a pattern. This generality provides a powerful tool, since operations applicable to a pattern (passing as a

parameter, for example) are automatically available for a range of constructs. We felt that this unification does not aid the student in understanding the respective roles of the different concepts. The unification, in the context of first year teaching, becomes a disadvantage, since the resulting techniques remain largely unused, but simple concepts are clouded. Several other aspects, such as the fact that Beta does not enforce information hiding, contributed to the decision not to consider Beta further.

Sather

Sather [Omohundro 1991] can be seen as a cross between C++ and Eiffel. Its development started explicitly to create a “simpler Eiffel”. Unfortunately, it adopts some of the characteristics that we mentioned as problematic in the discussion of C++.

These are:

- the inclusion of untyped objects and a reduction in the level of static type checking. Some type checks are static while others are dynamic.
- the need to explicitly identify dynamic dispatch. This is done in Sather on the basis of a variable holding an object reference. Only specifically marked variables will cause method calls to be dynamically dispatched.

Overall Sather operates at too low a level for good conceptual development. The reason for this approach is the aim for high efficiency of the resulting code, which is not a major concern for us.

Ada

Ada [DoD 1983] was, at the time we conducted our initial survey, not seriously considered because it did not provide essential object-oriented constructs. Since then, a new version of Ada (Ada95) has been released that supports object-orientation. This has certainly made it more interesting, but other characteristics still speak against Ada. The main problem is its size. Ada is a huge language (in terms of language constructs). While this might be desirable for an all-purpose programming language, it is certainly too daunting for first year students to learn. Defining and teaching a subset would be unavoidable – a situation that we identified as undesirable in chapter 2.

Ada95 also has disadvantages in the style in which its object-oriented features are implemented. Object-orientation is added onto the pre-existing “package” construct, with the result that some undesirable features remained in the language while some object-oriented features are implemented in unusual ways. For example, the receiver class of a message must be named as a method parameter, rather than the method being implicitly part of the class. On the other hand, functions may be written in a package that do not have the class as parameter – these are then independent (non-class) functions. This is further complicated by the fact that the programmer must be aware of the controlling parameters for dynamic binding. Dynamic binding is not automatic, and constructor functions are not available in a straightforward way. All of these details make Ada95 unsuitable for an introduction to object-orientation.

Oberon

Oberon [Wirth 1988a] is not a pure object-oriented language, but a hybrid. Code may be written in an object-oriented or a non-object-oriented style. The prerequisites discussed earlier stated that such languages are undesirable.

3.6 Summary

A survey of the languages commonly used for teaching object-orientation reveals problems with all of them. The problems are of a different nature and of different scale for each of these languages. Two methods of analysis are available to us: experience reports from teachers who have used the language in practice and a comparison of the language against our requirements listed in chapter 2.

Experience reports are mixed in their conclusions, but a general trend can be found. Object-orientation itself was seen unequivocally as a powerful and valuable teaching tool, while almost all authors reported problems with specific languages and systems they used. Most of the studies reported difficulty in switching to the object-oriented paradigm from the procedural approach.

In analysing the languages, the language itself and the programming environment used both have to be considered. A well designed language is only half of what we need and is rendered useless by the absence of a suitable environment. With some languages, the environment was the source of the most serious problems. Thus, programming environments will be discussed in more detail in the next chapter.

4 Environments

The programming environment is the second major determining factor in the perception students gain of the programming process. It has a large influence on the degree with which students can concentrate on the programming problem rather than become distracted by secondary issues. The programming environment can fundamentally shape the way programmers think about programming [Dumas 1995]. Yet, in discussions about introductory programming languages, the environment is often ignored. While teachers discuss at length benefits and drawbacks of particular languages, arguments about the suitability of particular environments are rare.

When using an object-oriented system for first year teaching, the environment increases in importance compared to the use of a procedural system. Since even small programs typically involve several files, issues such as multi-file-editing, compilation dependencies and linking become immediately relevant. A good environment that manages these issues is essential in order to let students focus on the programming task. In fact, it seems that many of the problems reported in the literature with teaching object-orientation have their roots in the use of unsuitable environments.

In this chapter we discuss the background and characteristics of some existing development environments, from early "traditional" systems to modern object-oriented integrated packages. We will point out shortcomings of existing systems and discuss what a good object-oriented environment should look like, emphasising again issues especially important for first year teaching. Some of the arguments presented here have been published in shorter form in [Kölling 1996b].

4.1 Background

Concurrent with the development of object-oriented languages we have seen major improvements in program development environments. Whereas early systems simply provided an editor and a compiler, modern programming environments provide facili-

ties such as source code control, library management, support for group work, version control and integrated edit/compile/test/debug environments.

The focus of recent research in software development environments has been on a wide variety of problems that confront professional software engineers. They include emphasis and support for collaboration between multiple programmers [Smith 1997], debugging facilities [Baecker 1997, Ungar 1997], testing [Clarke 1988], low-level language (syntax) support [Fry 1997], cross development [Dumas 1995], design [Winograd 1995] and the software process [Notkin 1988].

As object-oriented languages have grown in popularity there have been attempts to bring together environment and object-oriented technologies [Haarslev 1990, Holt 1994, Meyer 1993]. Environments were developed that support the development of programs in object-oriented languages, sometimes supported by object design tools. In most cases the approach has been to adapt an existing software development environment to an object-oriented language. Such attempts have not, in general, managed to capture the potential advantages offered by object-orientation.

The most significant reason for the failure to fully exploit the possible benefits is that existing systems concentrate on abstractions that are appropriate for procedural languages. Consequently they provide support for development of procedural programs, management of source files, organisation of test data, etc. An object-oriented development environment should provide support for classes and objects as the fundamental abstraction. Attempting to use mechanisms designed for procedural program development to develop objects is not necessarily appropriate.

As a simple example, consider testing. A *procedural program* development environment will provide support for testing *procedural programs*. This would include setting up of input data, capture of output data and comparison of the actual output with some expected output data. An *object* development environment will provide an *object* test facility. This should allow interactive invocation of the interface routines of an object – a quite different paradigm than that required for procedural programs. An object-oriented test environment has several potential advantages, which include support for incremental development and the removal of the need to write test programs.

The lack of truly object-oriented development environments has created major difficulties for teaching object-oriented technology. In particular, students have major conceptual difficulties and tend to write procedural programs in an object-oriented language. This is particularly prevalent when using hybrid languages. If we expect our students to fully embrace object-orientation then we must provide them with an appropriate program development environment.

In systems traditionally used in many universities, such as Unix, program development has been based around a textual command line environment, where a set of separate tools is provided to support the development process. These tools (typically an editor, a compiler, a debugger and a *make* facility) are based on concepts developed in the 1960s and have not changed much since their introduction. They are basically stand-alone tools and have only been slightly enhanced to cope with the

increased complexity issues and new programming paradigms. The *make* command, for example, a tool intended to help manage the complex compilation process for large systems with source code that is spread over multiple files, often becomes a complexity problem itself.

This situation has been dramatically improved by the appearance of integrated graphical programming environments, which are most prevalent on personal computers. These environments are able to significantly reduce the management overhead in software development, integrating editors, compilers and debuggers into one coherent system, thus significantly reducing the complexity of the overall software development task. All tools are seen as part of the overall process to build an application and link smoothly together. A debugger, for instance, can use the editor to point to source lines corresponding to code currently being executed.

4.2 Environments for object-orientation

The next logical step in improving software development tools is to unify the new language facilities with advanced development environments. Although Smalltalk was integrated into a sophisticated environment from the time it was first implemented, more recent object-oriented languages seem to have initially overlooked this aspect of software development and have tended to focus on text based interfaces. Only relatively recently, when actual use of new object-oriented languages has become more widespread to a large variety of users and projects, has this deficiency been noticed. This has provided an impetus for the development of graphical, integrated environments for object-oriented languages. Several environments for the more popular languages, such as C++, Eiffel and Java, have now been produced.

Fundamental deficiencies exist in most current development environments for object-oriented languages. The main problem is that the particular requirements and the potential of object-oriented systems have not been understood and utilised to assist the software development process. While there are numerous environments for object-oriented languages, few of them are object-oriented environments. This difference is important to understand.

A traditional program development environment can easily be adapted to support an object-oriented language. But this straightforward approach results in half-hearted solutions. The object-oriented paradigm and associated software development techniques have both their own requirements and potential benefits. Applying tools developed for procedural systems to an object-oriented language fails to fully meet the changed requirements and cannot exploit all the benefits that object-orientation offers. A real object-oriented environment will have a different set of tools and exploit the object paradigm to offer functionality not available in procedural systems. The object-oriented model should not stop at the application / environment boundary. It should be extended to include the environment to allow users to use the same conceptual models when thinking about the interaction with environment components as they do when thinking about interactions of objects in the application domain.

The main shortcomings of existing environments can be divided into two groups: insufficient object support and insufficient structure visualisation and manipulation techniques. If the environment is being used for teaching first year students, a third problem arises: complexity. We will explore the first two issues in this section and then discuss the aspects specific to introductory teaching in section 4.3.

4.2.1 Object support

Traditional procedural development environments facilitate the design and construction of *programs*. The basic entity they operate on is source code and their functionality revolves around the convenient manipulation of source code. The ultimate goal is to produce a program, an algorithm description with exactly one entry point that can only be built and executed after all its parts are (in some sense) completed. No notion of runtime objects exists within the environment, since those objects cannot exist independently from an active execution of a program. All data available outside the execution is in the form of files. Consequently, the environment has only to deal with source and data files.

When these environments were adapted to object-oriented languages, source files were replaced by class descriptions. Typically more source files exist in the object-oriented environment than in the procedural form. In addition, these files have more relationships with each other, e.g. usage and inheritance relations. Tools have been added to the environment to manage these class files and some of their relationships. The general paradigm of the environment, however, has not changed. The environment is still used to build an application with exactly one entry point that can be compiled and executed only after all its parts have been completed. This is the program-oriented paradigm.

The object-oriented paradigm is based on the idea that objects exist independently of each other, and that operations can be executed on them. Consequently, a user in a true object-oriented development environment should be able to interactively create objects of any available class, manipulate these objects and call their interface routines. The composition of objects at the user level should be possible [Booch 1986, Evered 1995, Gold 1991, Goldberg 1984]. This has also been referred to as an *instance-centred environment* [Gold 1991].

Such a facility, if provided, leads to the possibility of incremental application development, familiar from Smalltalk systems. Any individual class can be tested independently as soon as it is completed. Testing then becomes much more flexible than in procedural systems. In most current object-oriented environments, objects have to be wrapped in a non-object-oriented main program or script to create and invoke the first object or objects. A direct call of object interfaces is not possible, since object instances are not supported by the environment. In short: the programmer must fall back to the procedural paradigm to start and test a program. Thinking in two mindsets is required: one for thinking about the model of the application itself and one for thinking about environment interactions.

To avoid this problem, classes and objects should be the main abstractions used for user-level interaction in the environment. They should be treated as user level objects on which the user can perform operations by interacting with them through their interfaces.

Classes have been made the main mechanism of code structuring in several environments, partially meeting our demand. The style of interaction, however, often does not resemble anything close to an object-oriented model. Rather than interacting with classes by invoking operations (such as "edit", "create object" or "create new class") on the class, the user often deals with files instead. The files store the class's representation, and the operations executed by the user ("open file", "save file") operate on a different conceptual level than the one the user should be thinking about.

The ability to interact with objects has been neglected in most existing environments. The only widely available programming environments fulfilling many of our demands are Smalltalk environments. These, however, suffer from other problems, most of which have been discussed above (section 3.2).

A few environments, however, have been developed explicitly with goals similar to those stated here. The one that comes closest in its stated objectives is EiffelBench, an integrated development environment for the Eiffel language. Eiffel and EiffelBench have the advantage of not being derived from a procedural predecessor. The developers have thought about appropriate mechanisms for working in an object-oriented context. Meyer, in a description of EiffelBench [Meyer 1993], discusses in detail the question of what characteristics an object-oriented environment should have. He states five principles, most of which we agree with. The realisation of these ideas in EiffelBench, however, seems to violate most of his own principles. Meyer's principles are:

1 *Method-environment consistency*: A development environment meant to support a particular method or language must rely on a consistent set of user interaction conventions which closely parallel the concepts promoted by the method or language.

2 *Data abstraction*: In an object-oriented environment, the basic way of working must be through direct manipulation of visual representations of developer abstractions.

3 *Object-oriented tools*: In an object-oriented environment, each tool must be based on an object type (not on a type of operation).

4 *Semantic consistency*: An object-oriented environment must enable its users, for any symbol (textual, graphical, or otherwise) representing a development object in the user interface, to select the object through its symbol and apply any operation that is semantically valid for the object, regardless of the symbol's context.

5 *Typed environment*: In an object-oriented environment supporting a statically typed language and methods, the environment's visual conventions should display and enforce the type constraints on development objects.

While these principles are a good guideline, they are not realised in the EiffelBench environment (although the developers claim that they are). Principle 2, for example, is a very important one: developer abstractions should be directly manipulatable. Yet EiffelBench offers no facility to interact with object instances – clearly one of the most important developer abstractions. The objects the environment recognises are development entities, such as classes and variables, but not the objects with which the application under development is concerned. And even this restricted set of objects is handled in an inconsistent way. The interaction mechanism passes object representations to object tools – a mechanism that much more closely resembles the passing of the object as a parameter than the invocation of an operation on the object. The environment interaction style therefore is procedural, not object-oriented.

Principle 1 demands method-environment consistency, yet at the language level operations are invoked by selecting a method from the object's interface, while in the environment it is done by passing the object as a parameter to a tool. Principle 5 touches on a related subject: the type in the language is represented by its class, which, in turn, is defined by its name and features (the interface). If operations were selected directly from the interface of each developer object (rather than passing the object as a parameter) then the type check would be implicit (no routine could be called that is not in the interface). The requirement demanded in principle 5 is, in fact, only necessary because EiffelBench does not support an object-style invocation of operations. The mechanism used in EiffelBench also makes it necessary to invent an explanation for other apparent inconsistencies. A routine, for example, can be passed to the class tool (an apparent type violation). This is explained with inheritance-based conformance, although it is hard to see how a routine is a descendent of a class. Overall, the principles stated here highlight the right problems, but the realisation in EiffelBench fails to properly achieve any of the goals.

The problems identified with this example exist in most development environments. The ability to interact with objects for testing and development is generally missing – in most systems the situation is even worse than in EiffelBench. The effect of this lack of interactivity is especially serious for teaching systems. For initial learning of the object-oriented concepts, mechanisms to support experimentation and direct manipulation are particularly valuable. The ability to interactively create several objects from a class, and then to call interface routines on each of these objects, serves much better to clarify the roles of classes and objects in programming than a textbook could ever achieve.

4.2.2 Visualisation support

The second major shortcoming of object-oriented programming environments is the lack of appropriate visualisation mechanisms. Graphical visualisation techniques (e.g. as in [McDonald 1990]) should be used to display relationships between classes and objects. For example, inheritance and usage relations as well as call structures could

be shown. While thinking in terms of diagrams is common at the design level and most CASE tools provide support for graphics to model program structures, this is not well represented in programming environments. Even though the relations between objects are the most important factor in the design of an object-oriented system, little support exists in development environments for their management with modern visualisation and manipulation techniques. This lack of support discourages the use of graphical representations by students.

Again, EiffelBench provides a good example to illustrate this point. Meyer [Meyer 1993] describes in detail how EiffelBench tools can be used to inspect the class hierarchy of a system. A user can navigate from a class to its superclasses, from there to its clients, on to its subclasses, and so on. At each stage a textual view of the current class is presented which may be used to get information about the related classes. If the classes were represented graphically, including their inheritance and uses relationships, more information could be conveyed quickly and much more easily. Not only sub- and superclasses would be obvious to the viewer, but also other relations (such as siblings, which are not directly mentioned in each class) would be immediately visible.

An environment should make use of both graphical and textual representations. Studies (e.g. [Petre 1995]) have shown that neither text nor graphics can be considered generally superior for representation tasks. Both have their place. The advantage of adding a graphical notation is the ability to provide richer secondary cues. Graphical secondary notation (such as proximity, grouping and white space) can offer immediate access to information that is difficult to extract from the textual representation.

Manipulation using the graphical or textual representation of a class should be possible interchangeably. This means, for instance, that it should be possible to graphically edit the inheritance relationships of the classes in an application. The changes made graphically should automatically be reflected in the source code of the classes. The same should work the other way around: if a class is specified as an ancestor in the source code of another class, this relationship should automatically be reflected in the graphical representation.

Most existing environments for object-oriented languages today lack all of these features. All command line based environments, such as Unix shells, obviously lack facilities for visualisation. Class and object relationships, which are a fundamental part of the programming process, are not sufficiently visualised and poor modelling techniques for these are provided. Also, the integration of the tools involved in the development process is typically poor.

Graphical systems for most languages support only some of these requirements. Most graphical systems provide good tool integration, but lack support for object-oriented characteristics as described above. Some advanced professional object-oriented development environments, such as Visual C++ or Delphi, use graphical support only to build the user interface of an application, but neglect the internal structure of the program itself. This is not the kind of graphical support that is helpful to beginning students. The use of a graphical user interface builder might be helpful to construct

good looking programs (which can be a positive factor for motivation), but it should only be a second step in the process of learning how to develop computer programs. The first and fundamental issue is to understand the abstraction process – the underlying structure used to model the problem domain. The graphical user interface detracts from these issues and conveys a completely misleading picture about the character of object-oriented programming.

Graphical techniques should be used to visualise exactly the things we want students to concentrate on in the first course: how to structure a program. A colleague in a discussion recently mentioned a problem that he had in his course. He was teaching Eiffel and remarked that most of his students had difficulties in grasping the concept of classes and objects. Considering the environment they used (an editor and compiler run from a Unix shell), this is not surprising. How can we expect students to think in terms of classes and objects if all they ever see on the screen are lines of code?

4.3 Environments for teaching

Some environments exist that provide graphical representations of program structure as discussed in the previous section. Many of these are dedicated CASE tools, but some are programming environments. The Mjølner Beta environment [Knudsen 1993] is an example of an advanced programming environment that meets many of our demands about structure visualisation and text/graphics integration. The reason we have not discussed those systems in more detail has to do with an additional problem: complexity.

Just as we argued in the context of the programming language that the requirements in a teaching context are different from those for an industry strength production language, the requirements for the environment differ as well.

The problem with existing advanced environments is that they were developed for professional software engineers (with very few exceptions). In the context of users who are professional software engineers a powerful but complex environment is appropriate. A software engineer may spend several weeks or months becoming really familiar with a good environment – a price that is worth paying if the tool is to be used for the next few years. Examples of this are the Dylan environment [Dumas 1995] and the Mjølner Beta environment. Both are (in very different ways) advanced, modern object-oriented environments, and both are clearly aimed at programming professionals. The Dylan environment, for instance, provides sophisticated mechanisms for customisation and user defined groupings – mechanisms that are valuable for experts but tend to confuse beginners.

The main focus of recent research in software development environments has been on support for professional software engineering issues, such as support for distribution, collaboration, testing and support for the complete software process. Little has been done to address the problems associated with beginning students and, as a result, none of the existing environments is appropriate for teaching in first year.

In the development of a software system, trade-offs have to be made. One of these trade-offs is the one between ease of use/intuitivity and the power of the system. Generally, the more powerful the system is made, the more difficult it becomes to use. Many design decisions would be made very differently if the system is designed explicitly for beginners instead of professionals.

For teaching, we need a system that is easy to learn and intuitive in its use. It must be powerful in the areas that are important to us (namely direct interactivity and visualisation) and still provide a simple interface. Designing such a system includes as many decisions about what functionality to leave out (because its importance rates lower than the goal of maintaining simplicity) as those about intuitive interfaces to important functionality.

4.4 Some environment examples

We cannot, in the scope of this work, discuss a comprehensive list of environments in detail. There are too many of them, and new ones appear every month. Individual environments should be judged against the principles we have discussed above. We will, however, comment on some individual environments as an example of some of the most common shortcomings of popular systems.

Smalltalk environments

Some of the most interesting systems in our context are Smalltalk environments. They provide a browser to support the use of the library classes. Programming as a process of modelling with a combination of user-defined classes and reuse of existing libraries is presented elegantly and in a very consistent manner. They also provide the high level of interactivity and object support that we demand from a good teaching environment. The effect of this is highlighted by a quote from Adele Goldberg: "Smalltalk denotes fun and success. Developers learn how to create software systems by changing existing applications, and redefining existing tools. Tangible results are immediate and rewarding" [Goldberg 1995b]. This immediate feedback and reward and the sense of fun created by it represents the most powerful support in teaching programming to beginning students for which a teacher can wish. Every teaching environment should try to create this sense of fun through interactivity and immediacy. Of special interest in this context is the Portia environment [Gold 1991], an extension of traditional Smalltalk environments with greater emphasis on direct manipulation of objects.

Smalltalk, however, lacks other important facilities: no visualisation tools for class relations are available. The main problem with this lies in the Smalltalk language itself: since it is not statically typed, it is not possible to extract usage relations from its source code. No indication exists before runtime as to the call relationships between classes. Inheritance relationships as shown in the browser do not present the relationships of one application but rather the whole Smalltalk environment and so the browser is not used as an application modelling tool. Smalltalk blurs the distinction between the environment and the application under development.

Reports about the use of Smalltalk systems for teaching also point to another problem: its size. While the language itself (in terms of the number of constructs) is small, the class library and tools are large and often confusing. Several authors reported difficulties with the students ability to cope with the environment [LaLonde 1990, Skublics 1991], especially that experimentation and self directed learning was not working well because students were overwhelmed by the system. They also found that the functionality of the browser should be limited, since its power and flexibility caused more problems than it solved.

EiffelBench

We have already discussed many of the characteristics of EiffelBench above, when we used it to illustrate the arguments about object support and visualisation (sections 4.2.1 and 4.2.2). In short: while it was developed with appropriate goals and intentions, it fails to support objects and visualisation in a satisfactory way. In addition to this it presents other problems. The most severe is the complexity of its interface. Students (and many experienced programmers) generally have difficulties coming to grips with the meaning of the many interface controls, most of which are labelled with obscure icons that do not convey any intuitive meaning. Another problem is the quality of this environment. After using it for several years for teaching, Nørmark [Nørmark 1995] described it as low in quality and unreliable. He names compilation speed, incorrect compilation, primitive text editing capabilities and a difficult-to-use interface as some of the problems.

Borland C++

Borland C++ is a typical example of an environment that evolved from an earlier, procedural predecessor and inherited many of its characteristics. It has been extended to support object-orientation, but in many cases this appears as a half-hearted, second-best solution. For this review we evaluated Borland C++ version 5.0[3].

The environment is based around a project which, in turn, is a collection of files. Files can be made to loosely correspond to classes, but nothing enforces such a relationship. Also, as is the convention in C++, classes typically consist of at least two files (a header file and an implementation file). The problem becomes apparent when we consider the connection with the class diagram. Borland C++ provides a simple, automatically generated diagram showing classes involved in the current project. This diagram cannot be edited, but a class name can be double-clicked to open its source. This operation, however, only finds and opens the header file – it is left to the user to find the implementation.

Borland C++ provides no support for runtime objects or testing. The implementation is stable, although it still suffers from a number of errors (such as occasional mistakes in its dependency analysis and resulting incorrect compilation).

---

[3] All tests of environments were carried out on a 120 MHz Pentium processor machine with 32MB main memory under MS Windows 95.

It uses the Microsoft help system as a replacement of a class browser. As a result, the class documentation is separated from its implementation. This becomes a problem if teachers add their own classes to libraries – no help is available for those through the normal tools. Borland's help texts are often minimal, full of jargon and do not include fundamental information. The system is difficult to navigate and students typically do not manage to use the help system in their first semester.

Visual Age

Visual Age is an integrated development environment developed by IBM which is available for different languages, including Smalltalk, C++ and Java. The environment evaluated for this project was Visual Age for Java, version 1.0.

Visual Age borrows much of its interface principles from early Smalltalk environments. It uses a browser that can display the packages and classes in the system. The browser is divided into panes very similar to those known from Smalltalk-80: one pane shows a list of packages, another one a list of classes in a selected package, a third the functions in the selected class. A bottom pane displays the code for one selected function. The user always views and edits one function at a time. As soon as the function is saved it is automatically compiled. This is intended to create an "immediate execution" environment: as soon as a class is written, it can be executed. All of this is very similar to the interface of Smalltalk environments. There are, however, some differences.

One immediately noticeable difference is compilation speed. Because compilation is automatic and fairly slow, the user is regularly interrupted and forced to pause to wait for the system to catch up. When a data member is entered into a 20-line class, for example, the system is unresponsive for more than 30 seconds while automatic recompilation takes place. Overall, the speed of the system (or lack thereof) becomes a serious annoyance when trying to work with it for an extended period.

Visual Age also inherits most of the complexity from Smalltalk, and adds additional complexity of its own. The browser is used to display an overview of the complete Java universe (merging the view of the current application with the view of all existing class libraries), just as Smalltalk does. It offers several different views of this universe as well as different specialised browsers. Overall, the functionality far exceeds what is practical to be learnt by first year students. It is evident from the interface alone that Visual Age is aimed at a professional programmer who needs to spend considerable time to familiarise himself/herself with the environment.

Object creation and interaction is not supported in the environment. The tools are aimed at generating one monolithic application which can then be executed as a whole (with the advantage of Java that each class can have its own *main* function, effectively allowing different entry points to the application; this can be used for slightly more flexible testing). A graphical application structure is not provided. Classes in the libraries and the application are presented as a nested list. A graphical interface is provided for building graphical user interfaces for user applications.

Overall, Visual Age is a typical example of a system that does not support the object concepts at the environment level, and is clearly aimed at professionals, making it much too complex to use for first year students. It cannot be considered suitable for introductory education.

Visual C++

Visual C++ is an integrated development environment from Microsoft Corp. It exists in almost identical form for other languages (e.g. Visual J++, a Java version). The language independent part of the environment is sometimes referred to as the *Microsoft Developer Studio*. We evaluated Visual C++, version 5.0.

Visual C++ is clearly a very mature environment with many well thought out features. It has a highly flexible user interface which can be easily and quickly customised for personal preference or specific tasks. Toolbars can be easily enabled or hidden, they can be present in free floating windows or at fixed screen locations (which Microsoft terms "docked" toolbars). Output can often be presented in separate windows or in panes arranged in a single window, with easy and flexible pane arrangement. Extensive help is provided in easily accessible format.

Unfortunately, though, this maturity is only present in the traditional areas which were already present in procedural environments, not in specific support for object-orientation. All the areas that we emphasised most in our requirements, visualisation of class relationships, object support, direct object interaction and ease-of-use, are not supported well (or not supported at all) in this system.

Classes are displayed in a list, and no class relationships are graphically shown. Objects cannot be interactively created or invoked – they do not exist as an abstraction in the environment. Extensive file management is necessary to set up a project. The environment includes a large number of functions and options which are only interesting for professional development and have a strong intimidating effect on beginners. When creating a project, for example, the user must choose from a list of 14 possible project types (such as creating a program, a dynamic library, a static library, etc.). The user must also carefully think about directory structures, file names and locations and absolute path names. A new user is immediately confronted with this complexity problem: when starting the system, the standard interface presents 9 menus with 96 menu items, 27 toolbar buttons, three window panes, one of which contains 6 overlapping sheets of information. In addition, numerous contextual menus can be popped up which offer even more functions. Overall, it is obvious that this environment is aimed at a more expert user who needs the full flexibility of professional software development and is expected to spend considerable time learning to use the system. The target group clearly is not first year students, and trying to use it in such an environment would lead to complexity problems without offering the advantages of good object-oriented support.

4.5 Summary

The programming environment is an important, often neglected, part of the programming experience. It can shape in fundamental ways the mental models which students develop while learning to program. By doing this, it has a major influence on the ability of students to understand software development concepts and the degree to which they can cope with the problem of learning to develop computer programs.

Some of the main requirements for the environment are similar to those for the language: it should present the underlying concepts and tools in a consistent manner, and it should not be overly complicated to use. This means in particular that we need an object-oriented environment which is simple enough to be used for teaching.

Many of the environments on the market today are not object-oriented. They support object-oriented languages, but they fail to support and exploit object-orientation at the environment level. It is essential that we provide a more appropriate environment if we wish our students to produce truly object-oriented programs and to capture the potential of object-orientation.

Those environments which offer good support for object concepts are too complex to be effectively used in a first year teaching course. Problems with environments have been identified as the most common problems with teaching object-orientation to undergraduates. Mazaitis [Mazaitis 1993], investigating this issue, concludes: “It seems that before pure languages, Smalltalk and Eiffel, are more widely accepted, they must come bundled with support tools tailored to students’ needs.”

While some research has been done into teaching environments, some into integrated programming environments and some into object-oriented environments, those branches have not been combined to produce an integrated, object-oriented teaching environment. The effect of this is that currently, in most courses that teach object-oriented programming to beginners, environments that were developed for professional software engineers and often originally for procedural languages are used. As a result, object-orientation is not well represented and students are distracted from important issues by a need to struggle with overly complex environments.

5 The Blue System - An Overview

The preceding chapters were concerned with describing the problem of teaching object-oriented programming. We have discussed the characteristics of an ideal teaching system, and we have examined shortcomings of existing systems.

The remaining chapters of this work provide a solution. We describe the Blue system – our attempt to create a language and an environment that meets the requirements for an object-oriented teaching language. A brief overview of the language and the environment has been given in [Kölling 1996a] and [Kölling 1996b], respectively. The description here is much more detailed and includes extensive background information and argumentation on alternatives and reasons for decisions taken.

Blue is an integrated system – it is a programming language and it is a software development environment. Both of these aspects of Blue are important, and both are described in detail in the following chapters.

We have already, on several occasions, mentioned some of the potential advantages of integrated systems. Blue makes strong use of integration: the language and the environment influence each other. Because of this, we start by giving an overview of the Blue system as a whole. This overview serves to give an overall impression of the character of Blue, and acts as a basis to discuss separate aspects in more detail in the following chapters.

5.1 Getting started

Blue provides a complete environment for the software development task. All aspects of the work, editing, compiling, debugging and execution, are initiated from within the environment. To work in Blue, the Blue system is started from the operating system interface. From then on, Blue hides the underlying operating system.

A graphical interface is used to communicate with the user. From the main window, which is displayed when the system starts up, a Blue project can be opened with a standard dialogue. The main window then shows a graphical representation of the class structure of the current project (Figure 5.1).

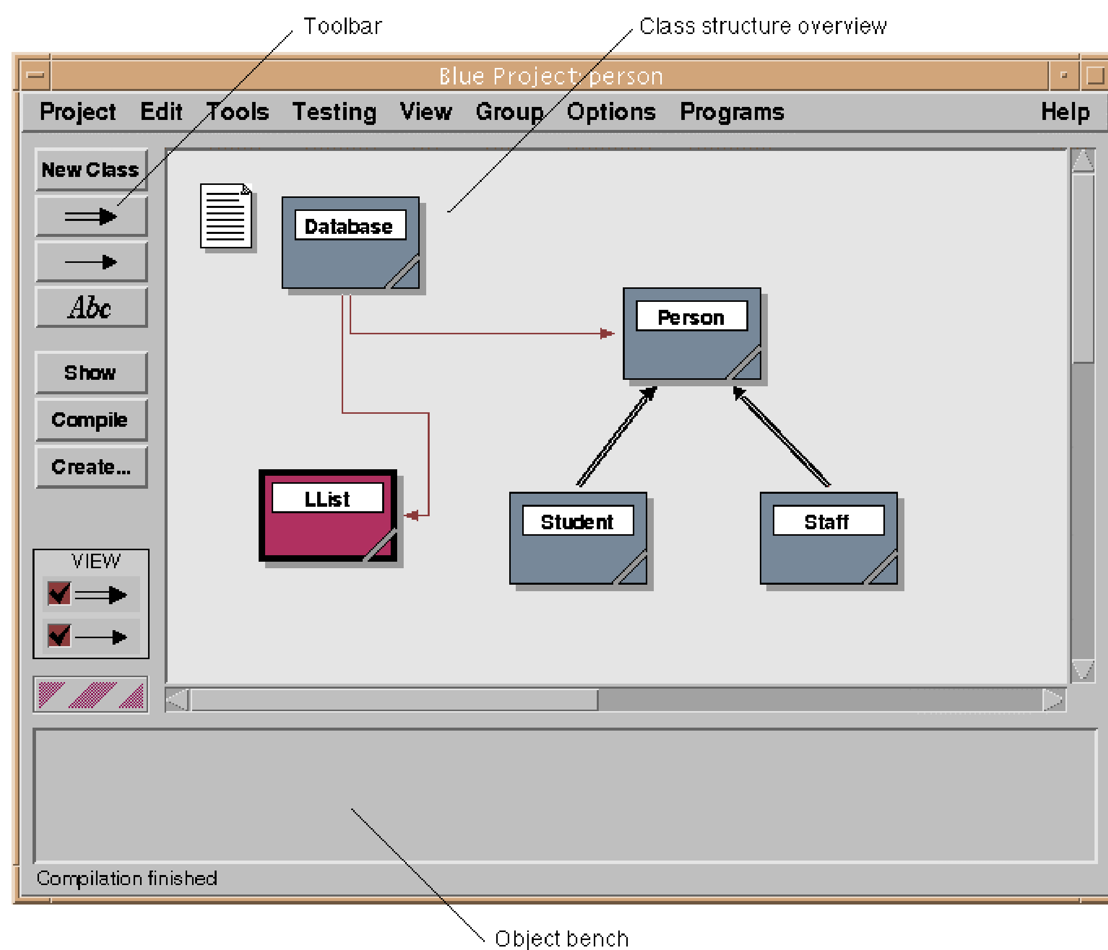

Figure 5.1: The Blue main window

Each class in the current project is represented by an icon on the screen. The appearance of the icon gives some information about the class: Colour and fill patterns indicate, for instance, whether the class has been compiled since being last edited and whether it was imported from a library. Arrows between classes indicate relationships. A single-lined arrow indicates a *uses* relation (a class declares variables or parameters of another, sometimes called *client* relation), a double-line arrow indicates *inheritance* (a class is a subclass of another). Classes can be introduced and

arranged on the screen by the user. Relationships can be set up by drawing arrows from one class to the other in a manner similar to a simple drawing program. Once the relationship is established, the Blue system manages the layout of the arrows.

5.2 Editing and compiling

Users can double-click a class icon to open an editor window showing the source of that class. Multiple editor windows may be opened at any time (Figure 5.2). A class that has just been newly created is automatically given a source skeleton which contains the basic elements of a Blue class. This skeleton can then be filled in to define a new class.

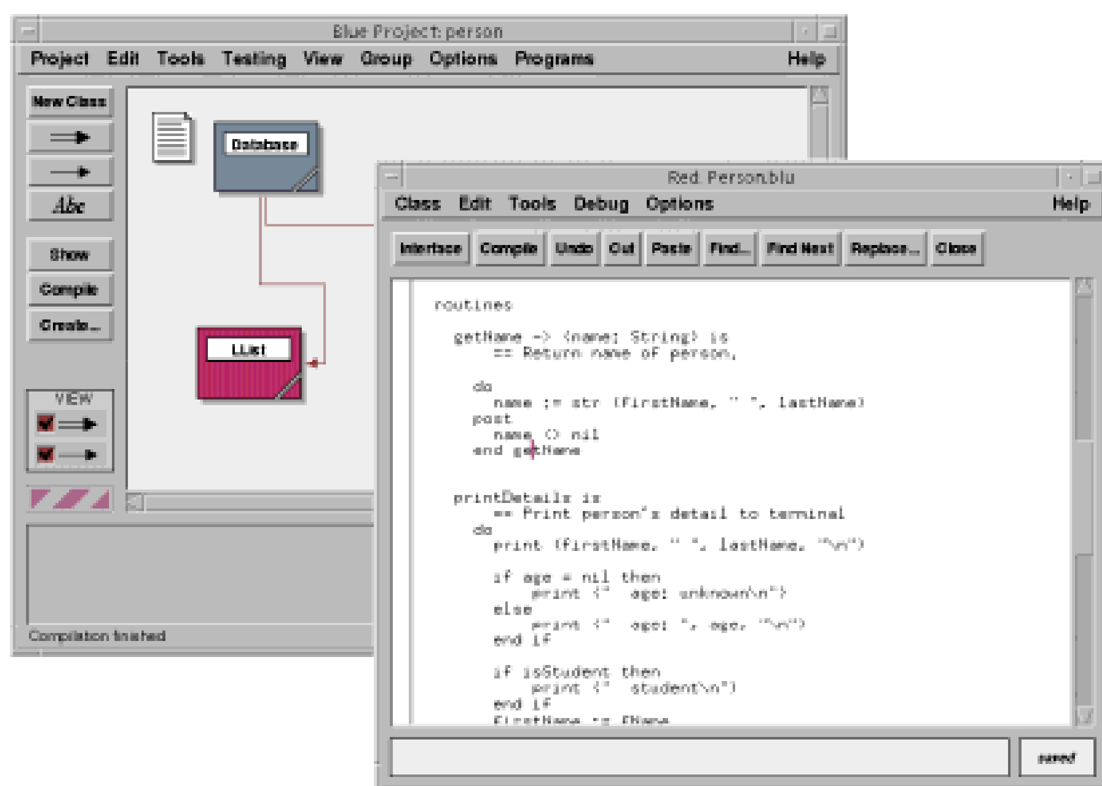

Figure 5.2: Editor windows show the source of classes

Relationships between classes are also represented in the textual form of the class (the source code). They may be edited graphically or in the text, and the other representation is automatically updated accordingly. Both representations are kept consistent at all times.

After editing the class, it can be compiled from within the editor by clicking on a button in the editor's toolbar.[4] If an error is detected by the compiler, the offending token is highlighted and an error message is displayed in the message area of the editor (Figure 5.3).

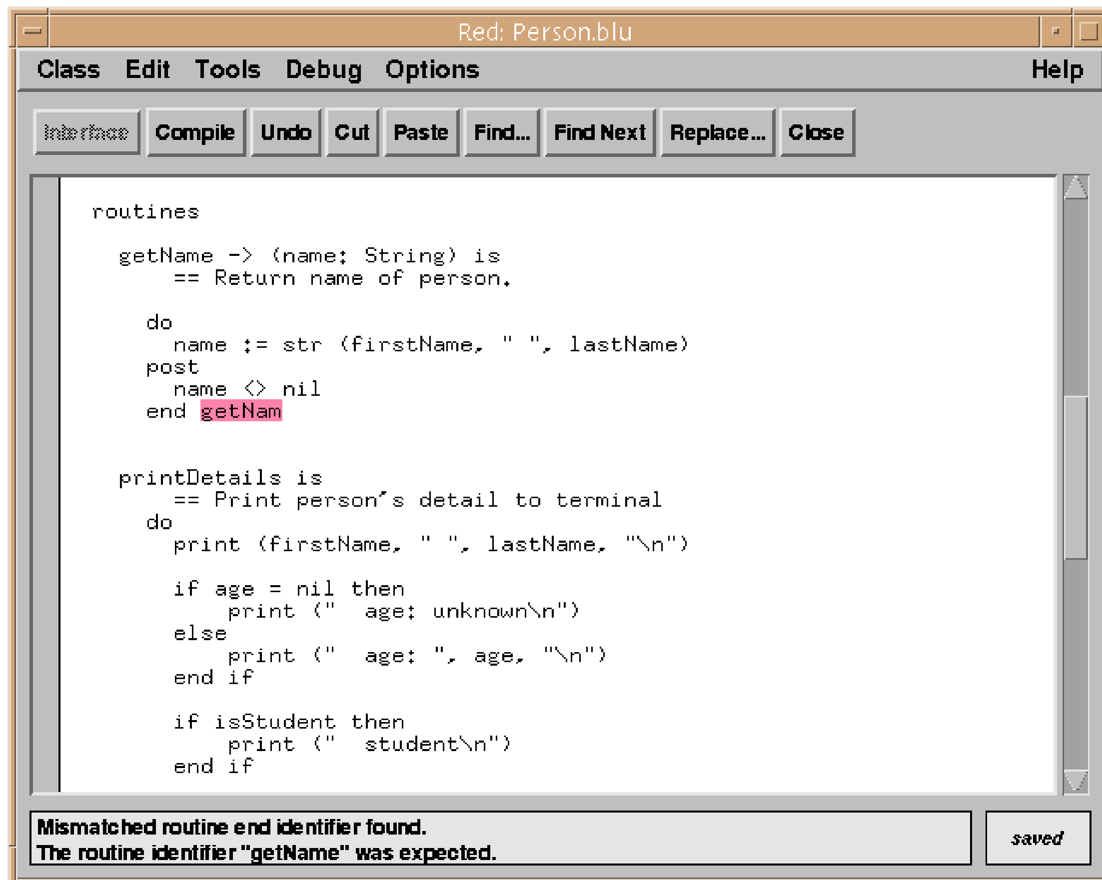

Figure 5.3: Display of compiler error message in an editor window

The main window also has a "Compile" button in its toolbar. Using this button causes the whole project to be compiled. This involves a complete dependency analysis and compilation of all classes that need to be compiled in the appropriate order. (Unix users may think about this function as *make*, with the difference that no *makefile* needs to be written.) Again, if the compiler detects an error in one of the classes, it opens an editor window to highlight the location of the error and displays an error message.

[4] All editor functions can also be activated via the keyboard. Many can be activated through menus, and the most commonly used are presented in the toolbar.

5.3 Creating objects

After a class has been successfully compiled, objects of that class can be created. It does not matter whether there are still uncompiled classes in the project – it is not necessary for the complete project to be compiled. Objects that are created in this way are represented by an object icon on the object bench (Figure 5.4).

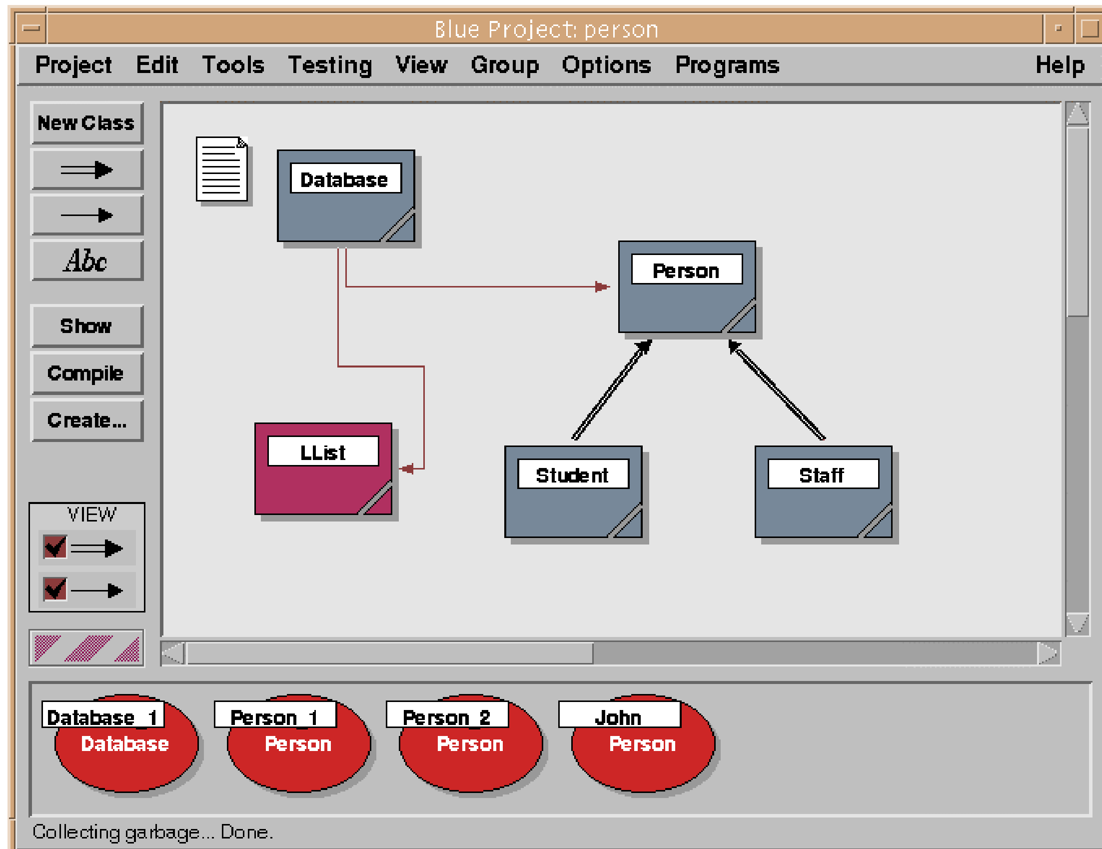

Figure 5.4: An object on the object bench

Each object on the object bench is available for immediate direct interaction. Any of its interface routines may be interactively called. A mouse click on the object will show a pop-up menu listing the available interface routines of that object (and two additional commands, *inspect* and *remove*) (Figure 5.5). An interface routine may be called by selecting it from that menu.

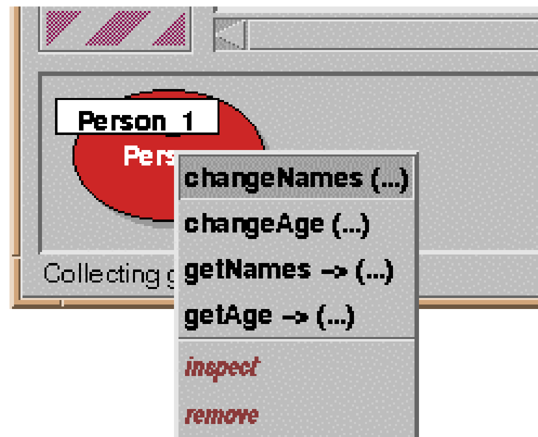

Figure 5.5: Object menu enables interactive routine calls

Routines in Blue have an optional parameter list and an optional list of return variables, both of with may contain one or more values. Figure 5.6 shows an example.

```
find_element (val: Value, reset: Boolean) –> (found: Boolean, elem: ListEntry)
    == Find the next element with value 'val'. If 'reset' is true, the search is
    == started from the beginning of the list. Returns 'found=true' if an
    == element was found, and the element itself in 'elem'. If no such element
    == exists, 'found=false' is returned and the value of 'elem' is undefined.
```

Figure 5.6: Interface of a Blue routine

The entry in the routine pop-up menu indicates whether the routine has parameters and/or return variables. If it has parameters, calling the routine will first result in a dialogue[5] being displayed that lets the user enter values for the parameters. Predefined values (such as integer or real numbers) as well as other objects from the object bench may be used. If the routine returns values, those will be displayed in a function result dialogue. Routines without return values may just change the internal state of the object, and thus have no immediately visible effect. To check whether such a routine worked as expected, the *inspect* command from the object menu may be used to open a dialogue that displays the values of the instance variables of the object.

[5] The term *dialogue* is used throughout for a user interface window which is typically displayed temporarily to let the user enter or view some data.

5.4 Interfaces

Once a class has been compiled, it might be used by another class. In this case, the implementation of the class is no longer of interest. It should be sufficient to know the *interface* of the server class.

For this situation, the editor provides an *interface* button. Pressing this button toggles the view from the implementation view (the full source) to a display of the class interface. The interface contains class and interface routine information, including signatures, comments and pre and post conditions. It also contains all inherited routines and information about the class from which they were inherited. For classes that have been compiled, the interface view is the default view. Library classes may be configured in a way that does not permit the viewing of the implementation and provides only the interface view.

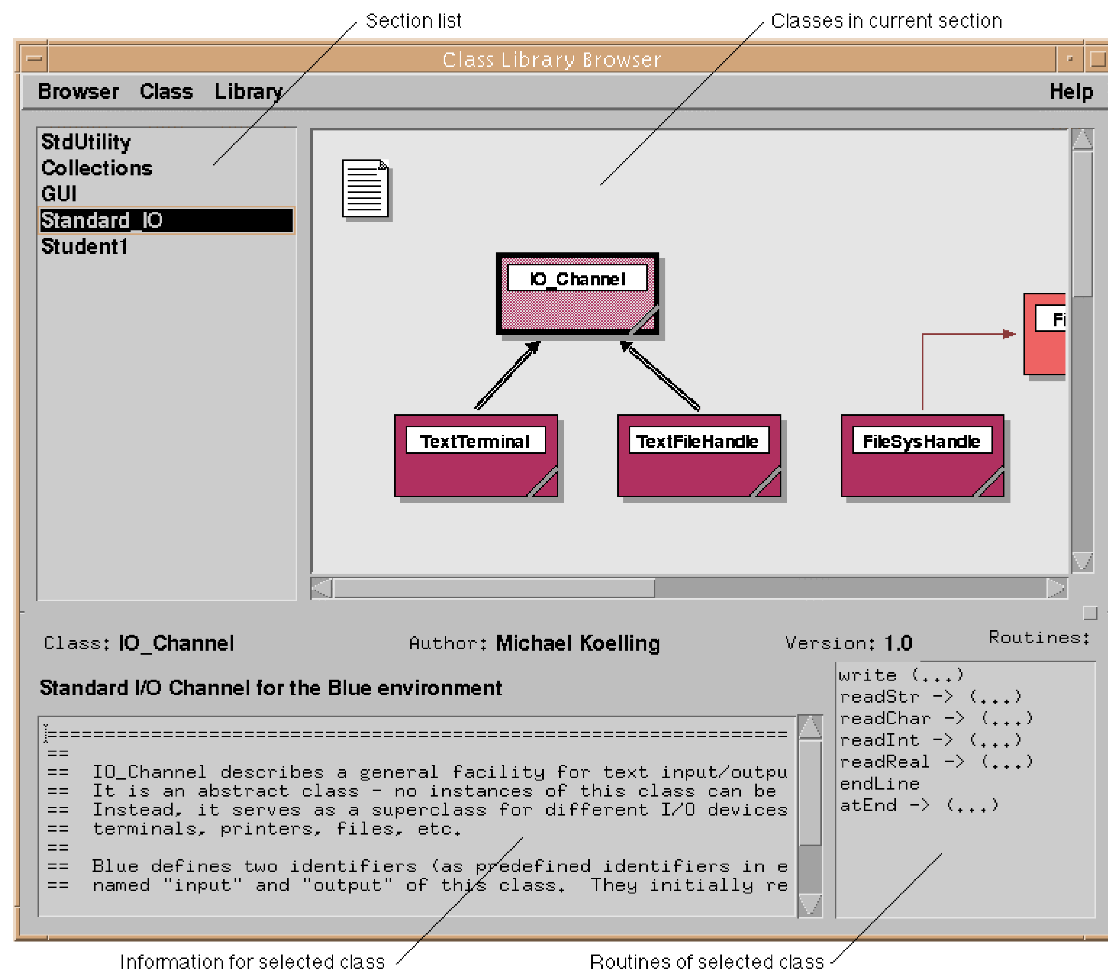

Figure 5.7: The Library Browser

5.5 Using library classes

Apart from creating new classes in a given project, classes may be taken out of a class library and added to the project. A class library browser is provided to find and inspect existing classes in the libraries (Figure 5.7). The browser supports browsing and keyword searching, as well as detailed inspection and the creation and maintenance of the users' own, personal libraries.

Libraries are sorted into different sections. The section list on the left of the browser window shows the available sections. The section view on the top right shows all classes in the currently selected section, as well as their relationships. Sections are usually arranged in a way that positions semantically related classes close together.

A class may be selected by clicking on its icon in the section view. The lower half of the browser window shows information about the class currently selected. The user can also double-click the icon to open an editor window displaying the full class interface.

Once a class has been chosen, it may be imported into the current project by selecting a command from the menu. Its icon then appears in the project window, and the class may be used like any other class in the project (although its source cannot be changed). Classes imported from a class library are shown in the project view in a different colour to indicate their special status.

5.6 Debugging

Two simple but powerful debugging tools are provided: single stepping and the inspection of variables and the stack.

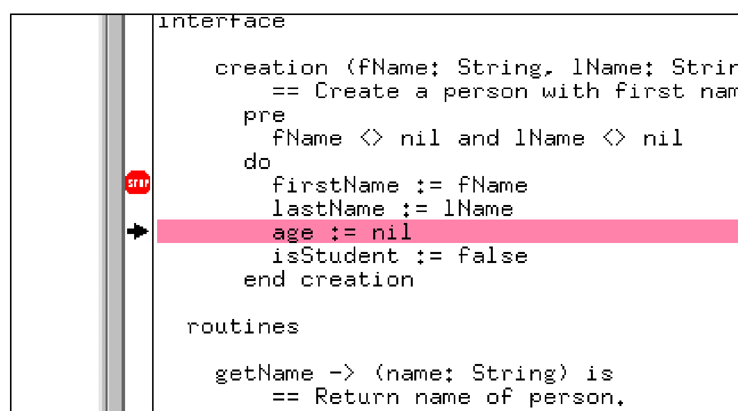

Figure 5.8: Source of a class with breakpoint

For both of these techniques, breakpoints are typically set somewhere in a class. A breakpoint may be set directly from the edit window. The class display has an execution bar at the left of the source line display. A simple click into this bar sets a

breakpoint in the corresponding line. The breakpoint is indicated by a stop sign in the execution bar (Figure 5.8).

When, during execution, a breakpoint is reached, an edit window with the source is displayed, and a control window with execution controls appears. These execution controls allow the user to step through the code. The current line of execution is marked in the source display with an arrow in the execution bar (Figure 5.8).

The control window can be expanded to show the function invocation sequence (the stack) and the values of variables.

5.7 Summary

The Blue environment provides facilities to graphically define and edit the class structure of an application. The classes involved can easily be edited and compiled with a few simple-to-use, powerful commands of the environment. The graphical representation or its textual equivalent may be manipulated, and both are kept consistent at all times. Complex problems, such as dependency analysis, compilation in the right order, and the processing of circular dependencies, are handled automatically without the need for user intervention.

Objects of any class may be created as soon as the class is compiled. These objects can be inspected, and operations defined on these objects may be performed. This provides a powerful method for interactive testing.

Reuse of previously written classes is easily possible through the use of a class browser that lets the user inspect and select classes for use in the current project. Class libraries include Blue standard library classes as well as libraries built by users or groups of users themselves.

Fundamental debugging techniques are available with minimal overhead of new tool interfaces. The edit window is used for the definition as well as display of breakpoints. The use of the debugging facilities is straightforward and intuitive.

6 The Blue Language

This chapter describes the major concepts of the Blue language. It is not a complete language reference. Many details are omitted, and some features are not described completely. The intention is not to provide a full language specification, but rather to introduce and explain the major design decisions that were made while developing the language. For a full language specification, see [Kölling 1997a].

6.1 Introduction

Before describing details about language constructs, it is helpful to obtain a general impression of the language by looking at some example code. Figure 6.1 shows part of an example class for this purpose. The example shown (a simple stack class implemented using an array) is a generic class with one generic parameter, named *ELEM_TYPE*. It can later be instantiated with any type as the actual generic parameter.

At the top of the class is a comment block describing the general purpose and functionality of the class (the double equals sign is a comment symbol). The body of the class is divided into three sections: the internal part, the interface part and the class invariant. The internal part is further divided into variable declarations and routine declarations (not used in the example). The interface part consists of a creation routine and interface routine definitions. Routine definitions include lists of parameters and return values, a routine comment, optional pre and post conditions and the routine body.

```
class Stack <ELEM_TYPE> is
==============================================================
== Author:   M. Kölling / Jeffrey H. Kingston
== Version:  1.1
== Short:    Stack of ELEM_TYPE with variable size
==============================================================
    uses
internal
    var
        elements: Array <ELEM_TYPE>
        numElements: Integer
interface
    creation is
            == create a new, empty Stack; O(1) cost
        do
            elements := create Array <ELEM_TYPE> (20)
            numElements := 0
        post
            isEmpty
        end creation

    routines
    push (elem: ELEM_TYPE) is
            == push elem; O(1) cost plus resizing cost
        do
            if numElements = elements.size then
                elements.setSize (2*numElements)
            end if
            numElements := numElements + 1
            elements.putElem (numElements, elem)
        post
            not isEmpty
        end push

    pop -> (elem: ELEM_TYPE) is
            == pop elem; O(1) cost
        pre
            not isEmpty
        do
            elem := elements.getElem (numElements)
            numElements := numElements - 1
        end pop

    isEmpty -> (empty: Boolean) is
            == true if empty; O(1) cost
        do
            empty := (numElements = 0)
        end isEmpty
invariant
    numElements >= 0
end class
```

Figure 6.1: An example of a Blue class

6.2 The object model

One of the fundamental decisions in designing an object-oriented programming language is the design of the object model. What are objects? How do they behave, what is their relationship to each other, and how do they interact? How are they stored, and how are they created?

6.2.1 What is an object?

The most fundamental question is: What is an object? The spectrum in existing object-oriented languages is wide. Some languages (e.g. Smalltalk) declare that "all information in the system is represented as objects" [Goldberg 1989, chapter 6]. Simple data types, classes and even control structures are implemented as objects. Other languages (such as C++) regard only data of some user-defined, "large" data types as objects. Instances of simple types (such as integer numbers and characters) are not objects, nor are the classes themselves. Even user-defined data does not have to be an object. Java takes a compromise route: Simple data types (e.g. integer) are not classes (and instances are therefore not objects), but there is also an integer object, for cases when it is needed.

The issues can be summarised in the following four questions:

Are control structures objects?
Are classes objects?
Are all user-defined data structures objects?
Are scalar/builtin types objects?

In the following sections we discuss each of these questions.

Are control structures objects?

Smalltalk takes the object-oriented paradigm further than any other language. Not only are instances of data objects, but the elements of the programming language themselves are objects, which can receive messages and return results like any other object. The language constructs follow the model of the data on which the language operates.

While this idea has a fascinating appeal to language designers (because of its striking elegance and uniformity) we believe that this view is not helpful for beginning students. One problem is that the treatment of control structures as objects (with message passing syntax for the use of control structures) results in very unusual syntax for program control. For example, the following Pascal statement

```
if (x > y) then
begin
   max := x; index := index + 1;
end
```

becomes in Smalltalk:

```
x > y ifTrue: [max <- x.
               index <- index + 1]
```

We believe the Smalltalk syntax not to be as easy to read initially as the Pascal syntax. Moreover, most other mainstream languages (C++, Java, Eiffel, Ada) use a syntax that is much closer to the Pascal variant. Because of the principles of readability, and the role of Blue as an entry point to other mainstream languages, Blue does not regard control structures as objects. Control structures are independent of data types.

An interesting approach to a related topic is implemented in the language Beta. Beta defines a *pattern* as a syntactical unit. A pattern is a very powerful construct which can be used to represent a wide range of language entities, such as classes, routines, processes and block structures. The idea is similar to the Smalltalk approach: using a unified model for representing various language entities results in a very elegant and powerful system. Because an instance of a class and an instance of a process, for example, are described in the same fashion, processes automatically become objects and can be treated in the same way as objects.

For Blue, we have decided not to attempt such a unification of concepts. While powerful once it has been mastered, it can cause considerable confusion for beginners. Both readability and clarity of concepts suffer under such a unification. Different language elements, such as a data item and a control structure, are inherently different and can be used in different ways. Equally, a routine is not the same as a class. Making them look the same does not help beginners understand their respective purposes and their differences. For the teaching of concepts it is best to represent each of the concepts in its own, clear style. The expressiveness of the language will necessarily suffer, but the clarity won by this decision is well worth the trade-off.

Are classes objects?

The question as to whether classes are objects is closely related to the previous one. In Smalltalk, for instance, objects are instances of a class. Classes themselves are also objects, and thus instances of another class (a meta class). By defining this, Smalltalk has a powerful mechanism for reflection. Since classes are objects, they may be dynamically created. In this way, Smalltalk can dynamically introduce new types (and thus new code) into a running application.

Most other languages (especially statically typed languages) do not regard classes as objects. Classes are compile time constructs, which do not exist at runtime. Objects are runtime constructs which do not exist at compile time. This distinction makes the respective roles of classes and objects very clear. For similar reasons to those given for the previous question, Blue adopts this view. The distinction of different concepts serves to add clarity and emphasises their respective purposes.

Java tries to get the best of all worlds: while classes and objects are orthogonal concepts, it adds a standard class library for reflection. This library allows the user to create and compile classes dynamically, thus adding the power of reflective systems.

Whether a mechanism like this would be helpful for Blue does not need to be decided at this point. Since this is implemented through a class library, it does not impact on the language design.

Are all user-defined data structures objects?

We have basically answered this question in the requirements section. One of the requirements we listed was that the language should be a pure object-oriented language. We did not want a hybrid language. Following this requirement dictates that all user defined structures should be objects. All code users write is part of a class. Blue does not support functions independent of classes.

Are scalar types objects?

Are instances of simple, built-in data types, such as integers, characters, booleans, etc., objects? There are many arguments either way.

Efficiency and syntax are frequently used arguments against the view of simple data as objects. It is not efficient, it is often argued, to implement integer numbers as objects. The memory and runtime overhead would be too great. The second argument is syntax: if integers were objects, for consistency reasons they should use object method syntax for invoking their operations. Adding the numbers 3 and 4, for example, should be written as

```
3.add (4)
```

rather than the more customary

```
3 + 4
```

This syntax, because of its unfamiliarity, might negate the advantage of having a unified concept and cause more confusion than the proposed benefits.

The main argument for the definition of simple data as objects is unification of concepts. The whole object model could then be explained by one general concept without the need for seemingly arbitrary exceptions. Passing an integer as a parameter, for instance, would be no different to passing a user-defined object. There is one conceptual problem, though. Objects must generally be created before they can be used. But do we really want to force a programmer to have to create the numbers 3 and 4 before they can be added? This would be impractical and counter-intuitive.

Another strong argument for viewing simple data as objects comes from generic composition. All object-oriented languages support polymorphism. It is available through inheritance and, in many languages, through genericity. Making a distinction between simple data items and objects might lead to problems with combining them. A generic list class, for instance, might accept any object type as its generic type parameter. Thus, it allows the creation of a list of persons as well as a list of buttons. But can a simple type be used where an object is expected? In other words, can we create a list of integers? If a distinction is made between simple types and objects, this might lead to problems.

Blue takes the view that all data are objects. In this aspect Blue differs from most mainstream object-oriented languages, with the notable exception of Smalltalk, which also views all data as objects. However, it differs from Smalltalk in many details.

The fact that all dynamic structures can be explained with a unified model is especially attractive for a teaching language. The disadvantages, however, must be addressed. The first was efficiency. There are two aspects to this. Firstly, since Blue is a teaching language, conceptual elegance takes precedence over efficiency. We can afford to make efficiency sacrifices. Secondly, the fact that simple types are conceptually the same as objects does not necessarily mean that they have to be implemented in the same way. Several optimisations are possible (and Blue implements them) for handling operations on simple, built-in object types in a more efficient way than general object operations. This can be achieved transparently without affecting the conceptual model.

The second disadvantage was syntax. To overcome this problem, Blue introduces a feature called “aliases”. An alias allows a second, more convenient syntax for selected operations. So, for example, the expression

```
3 + 4
```

is an alias for the operation

```
3.add (4)
```

This allows the use of familiar syntax and eliminates the need to understand object calls to write the first, simple programs.

Some languages, Smalltalk and C++ for instance, use another solution to the same problem. In those languages, the syntax for object calls is defined to allow the symbol + as a valid method call. Thus the problem does not directly arise. No choice needs to be made as to whether to use object call syntax or infix notation because the infix notation *is* object call syntax. Blue does not follow this path because of the more complex object call syntax, problems with coherent interpretations (such as operator precedence) and the resulting potential for misuse. Since infix notation is a valid routine call syntax, it is available to all objects, including user defined ones. This makes it possible to use it in places where its meaning is not intuitively clear, possibly resulting in less readable code. (This issue is further discussed in section 6.15.3.) The difference with the alias mechanism is that it is a construct that is available for only a few, pre-defined operations. It is not available to the programmer as a general purpose mechanism, thus ensuring that all other object calls share a common syntax. Aliases are discussed in more detail in section 6.10.

The third problem was that of object creation. A user does not want to create integer numbers before being able to use them. This problem is solved through the introduction of *manifest classes* in Blue, as described in section 6.2.4.

6.2.2 Storage of objects

It is common to distinguish two different types of storage for objects: immediate storage (the address of the object is represented by a variable) or storage by reference (sometimes called *pointer variables*, the address of the object is the *content* of a variable).

Some languages, such as C++, even require different syntax to access features of objects of these different storage types. Immediate objects are accessed through dot notation, e.g.

```
myObject.feature1
```

whereas objects stored by reference are accessed through an "arrow" symbol:

```
myObject->feature1
```

In a well designed language this distinction in syntax is not necessary. The compiler could know about the storage class of this object and automatically produce the right code (as it does, for instance, in Eiffel). But there are still important differences. Recursive or circular structures can be built only with references, not with immediate objects. Assignments and equality tests might behave differently, since they might or might not involve a copy operation. The question of identity of an object is fundamentally different (see detailed discussion in section 6.12.1). In short: in many cases it is necessary for a programmer to know exactly whether he/she is dealing with an immediate object or a reference.

Supporting these two concepts violates the principle of avoiding redundancy. Immediate objects provide no functionality that could not be achieved with reference objects as well. (The opposite is not true, though.) The sole reason for supporting immediate objects is efficiency. Immediate objects allow faster variable and routine access by avoiding one indirection. This argument is not sufficient for justifying the inclusion of this concept in a teaching language. Blue therefore supports only reference types. All variables store references to objects.

A second advantage of this decision is that the lifetime of variables and objects is separated. The lifetime of immediate objects, commonly stored on the stack, is typically bound to the lifetime of the variable. Reference objects increase orthogonality by separating these issues. CLU [Liskov 1981] takes the same approach, and a good summary of the reasons is given in [Liskov 1992].

Java follows a similar route. It also supports only references to objects. But it fails to achieve the same degree of uniformity because of the distinction between simple data types (which are not objects and are stored as immediate data) and object types. In Blue, the uniform use of objects by reference together with the definition of simple data items as objects, leads to a very simple and clean object model as the basis of the language. Variables always store references, assignment is assignment of references, and the default equality checks equality of references. Object identity is always preserved (assignment never duplicates an object).

6.2.3 Object creation

Through this object model, the creation of objects is greatly simplified. The creation of objects is always explicit.

Languages with immediate objects often define different kinds of creation mechanisms: explicit creation for reference objects and implicit creation for immediate objects (the object is automatically created when the variable comes into existence). C++ defines a third creation mechanism, a "copy constructor" that is implicitly executed through assignment.

Restricting storage to reference objects allows Blue to employ only one mechanism for object creation.

6.2.4 Manifest classes

One of the most difficult problems in unifying the object model is to accommodate for simple data types, such as integers, boolean values and enumerations. Ideally, we would like to view them as objects, but we do not want to be forced to create them before we can use them. We would like to be able to just write

```
a := a + 7
```

without the need for creating a 7-object first in a separate statement.

There are essentially two possibilities in the interpretation of the symbol "7" in order to achieve this goal: it could be interpreted as an object constructor, or as a constant reference to an object that already exists.

The language Dee uses a variation of the first alternative. Literals are regarded as constructors. There are several disadvantages with this approach. Firstly, the constructor rule has to be expanded (since it is no longer true that objects are only created through an explicit "create" expression) and secondly, equality becomes more complicated. Since potentially more than one 7-object can exist, equality cannot be taken to be identity. Consider:

```
if 7 = 7 then ...
```

We would certainly expect this condition to be true, but we would have created two different 7-objects! Dee solves this problem by defining the 7-symbol to be a constructor the first time it is used, and a reference to the constructed object for every additional use. While this solves the problem of having two 7-objects, the definition does not seem as elegant as we would like. Equality issues are discussed in more detail in section 6.12.1.

The second approach is followed by Smalltalk. Literals are considered constant references to objects. In Smalltalk it is, however, never explained where these objects come from, when and how they are created. They just magically exist in the Smalltalk universe. This introduces a special case in the object world, since most objects must

be created by the user, while some are already there. At the conceptual level, all objects are equal and nothing explains this difference.

In Blue, we deal with this issue by introducing the concept of *manifest classes*. We distinguish these from “normal” classes by referring to those as *constructor classes*.

These two kinds of class differ in the way their objects come into existence. Constructor classes are those with which we are familiar from most object-oriented languages. Initially, no objects of these classes exist. They can be created by executing a creation instruction which creates the object and executes a creation routine defined in the class. Manifest classes, on the other hand, define their objects by enumeration. All possible objects of these classes are automatically created at system startup, and the class defines identifiers by which these objects can be referenced. So instead of providing a construction method for objects, manifest classes provide the objects themselves. No additional objects can be created at runtime.

The classes “Integer” and “Boolean”, for example, are manifest classes. All their objects automatically exist, and references are provided to access them. The symbols “3” and “true” are constant references to the objects representing the integer number 3 and the boolean value *true*, respectively.

This approach has, of course, similarities with the solutions in Smalltalk and Dee. The difference, however, is that now the distinction is made at the logical level. The object model explicitly recognises these two different types of classes, and differences between numbers and complex objects can be understood independently of implementation concerns. They cease to be anomalies or special cases at a technical level. This reduces the danger of misunderstandings on the side of the programmer.

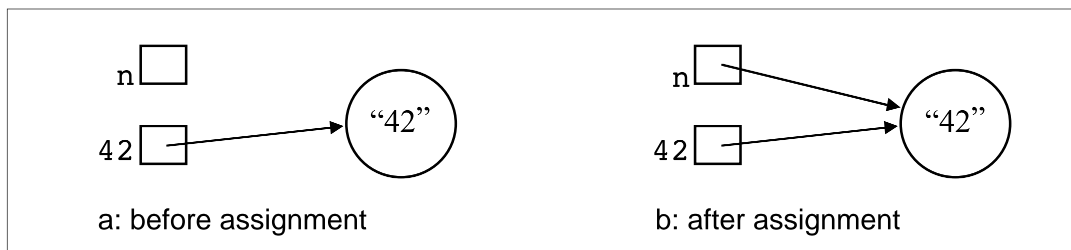

Figure 6.2: Assignment of objects

This definition, together with the storage model, behaves intuitively in all cases of arithmetic and assignment. For example, assume we have a variable *n* of class Integer. The symbol 42 is a constant reference to the object representing 42 (Figure 6.2a). An assignment

```
n := 42
```

assigns the reference to the 42-object to the variable *n* (Figure 6.2b). Since all assignments are reference assignments, and equality checks are reference equality checks, the expression

```
n = 42
```

will afterwards be true. It is important, though, that no further objects of manifest classes can be created, and that objects of manifest classes cannot be cloned. If it were possible for two 42-objects to exist, then two variables could both contain 42 and still not be equal. Allowing no additional creation for manifest classes avoids this problem.

Again, it is interesting to compare this to Smalltalk's view of simple types as objects. Their pre-existence makes them similar to Blue's manifest classes, but no explicit distinction between these and other classes is made. This leads to a problem with object equality. Because these kinds of object are not different from general objects, they can be cloned (as every other object can). Cloning a number object (e.g. the object representing 42) results in the existence of two distinct 42-objects[6]. To ensure that

```
42 = 42
```

is still true, the equal sign cannot be taken to mean object identity. To deal with this problem, Smalltalk defines two different equality operators: $a == b$ test object identity (a and b are the same object), and $a = b$ tests object equality (a and b have the same state). Manifest classes avoid the need for these two operators.

An important aspect of the manifest class model is the separation of concepts (an improvement in orthogonality): implicit creation is separated from predefined types. This leaves the possibility of the combination of implicit creation and user-defined types. Blue allows this combination for the definition of enumerations. Enumerations often do not fit in well with object-orientation and, as a result, most languages do not include them at all. Manifest classes allow us to provide a genuine object-oriented definition of enumerations that is seamlessly integrated into the object model. Enumerations are discussed in section 6.4.2.

Manifest classes are also loosely related to the concept of mutable and immutable objects in CLU, in that manifest objects are immutable while constructor objects are mutable. This is coincidental, though: it would easily be possible to define a language construct that allows the definition of immutable constructor objects (as CLU does). The concepts are in fact quite orthogonal. Blue does not support the explicit definition of immutable objects. But the fact that manifest objects are immutable allows very efficient implementation. These objects fit in the variables directly (manifest objects are not larger than pointers) without using the indirection of a reference, and storing them there is safe since they are immutable. Even though this means that assignment now does copy an object, this is not detectable by any program – it is an invisible implementation optimisation. This ensures that integer arithmetic, for example, can be implemented efficiently.

---

[6] Some Smalltalk systems implement this differently: the clone operation then does not, in fact, clone the integer object but is effectively a null operation. This, however, only shifts the problem to another area: instead of introducing a special case for equality a special case for the clone operation is defined.

Note that the distinction between manifest and constructor classes is a conceptual difference, not a technical one, and that it is on a class basis and not on a variable basis. In this respect it differs greatly from the immediate storage/reference storage distinction in other languages. Our distinction influences how objects come into existence, which is a semantic definition, and is defined in the class, which already holds semantic information about the class and objects of it. There can never be confusion over how a variable can be accessed, whether recursive structures are possible or when objects get duplicated, as there often is with the duplicate storage model. All objects are accessed in a uniform manner.

Object vs. values

A related issue is the distinction of objects and values in programming, as it is discussed by MacLennan [MacLennan 1982]. Values are *abstractions* which are immutable. It does not make sense to discuss identity of values, to duplicate them or to share them. Objects, on the other hand, represent entities in the model to be represented by the program. They can be instantiated, changed and destroyed, and they have state. Integers and enumerations, such as *2* or *red*, are values.

This distinction is usually not represented clearly in programming languages. The problem is that *program objects* are used to implement both values and objects in the model. In our approach, manifest classes represent values in the problem domain while constructor classes represent objects in the problem domain. It has been argued that the two relationships name↔object and object↔value should be treated separately and independently [Grogono 1994b]. We contend that the situation where two objects represent the same abstract value does not have a sensible meaning. It leads to confusion and should be avoided.

Blue provides unique, immutable objects for values and general objects for problem-domain-objects. The concept of a value here is bound to a simple data type, where general objects are typically compound objects with internal structure.

The missing combination – unique, immutable objects for user-defined compound objects – cannot be as easily dismissed as multiple objects for values. This is, however, an area for further programming language research and lies outside the requirements for a teaching language.

The discussion of the use of manifest classes has been summarised in [Kölling 1998a].

6.3 The type system

6.3.1 Type safety

Blue is type safe. It is strongly typed and also what we might call "largely statically typed". According to the definition given by Cardelli and Wegner [Cardelli 1985], a

language is strongly typed if it is type consistent, i.e. it is guaranteed that no type errors can occur at runtime. This may involve runtime checks to avoid those errors.

"Statically typed" is a stronger requirement. A language is statically typed if the type of every expression can be determined by static program analysis. "Statically typed" thus is not an interesting characteristic in the comparison of object-oriented language since, by definition, no object-oriented language can be statically typed. Part of object-orientation is the provision of polymorphism, either in the form of inheritance or through parametric polymorphism. This implies that the type of expressions denoting objects is not statically known. One of the achievements of object-oriented programming is, in fact, to have found a way to get rid of the straight jacket of static typing in a well defined manner that still guarantees type safety. Static typing is, however, desirable for most other language constructs.

When we say that Blue is "largely statically typed", we mean that dynamic type information is only used to support the basic polymorphism concept, subtyping with dynamic method dispatch, which is an essential characteristic of all object-oriented languages. No other construct involves implicit runtime checks (although there is one construct, the assignment attempt, that performs an *explicit* runtime check). Blue therefore, is as statically typed as it can be, consistent with its design principles.

Most importantly, though, since the dynamically typed polymorphism constructs (inheritance and genericity) are constructed in such a way that type safety is guaranteed (even though the exact type is not statically known), Blue is *statically type safe*. In practice, this is the characteristic that we are most interested in for a teaching language. It implies that all type errors are detected at compile time and can be reported early and at the real source of the error.

6.3.2 Types vs. classes

We often use the expressions *type* and *class* as synonymous. As long as no generic classes are involved, this is unproblematic – a one-to-one relationship exists between a non-generic class and a type. Each class provides exactly one type. We can interpret the phrase "x is of class A" as a short hand form of "x is of the type provided by class A".

The distinction between type and class becomes important, however, in the context of genericity. Here the terminology differs in different languages. In Blue, we call the generic source text a *class*, while each specific instantiation is a *type*. Consider, for instance, a generic class `Stack`:

```
class Stack <T> is
  ...
end class
```

In this example, `Stack` is a class while `Stack<Integer>` or `Stack<String>` are types. A generic class, thus, is a generator for a potentially unlimited set of types.

6.3.3 Predefined types

Blue provides six predefined classes: Integer, Real, Boolean, Enumeration, String and Array. Predefined classes differ from other classes in that they do not need to be explicitly imported via the "uses" clause. They are automatically known in all classes.

There is no character class in Blue. Where characters are needed, strings of length one can be used instead. This eliminates an entire predefined class and avoids common problems with character–string compatibility.

Strings are manifest classes. A string literal, such as `"This is a string"` therefore is a constant reference to a unique string object, not a string constructor. The reasons for this decision are related to equality considerations for strings. They are discussed in detail in section 6.12.1 (Equality).

The decision to include Array as a predefined type has been a contentious issue. The Array class is a generic collection class that holds a number of objects of another class. It has this in common with other collection classes, such as lists, stacks and queues. These other collection classes are provided in standard class libraries, which may be included into a Blue project and imported into a class via the "uses" clause. It could be argued that the class Array should be implemented in the same manner – as one of the collection library classes that may be imported through the library browser. It may even be argued that this would have educational value: often arrays are misused out of laziness. Their immediate availability and familiarity leads many students to use arrays in situations where the use of other collections, such as trees, lists or hash tables, would be preferable. Removing the special status of arrays (as, for example, Eiffel has done) might encourage the use of more appropriate data structures.

We decided nonetheless to give arrays their special status by pre-defining them. The reason is that we see arrays as fundamentally different to other collections. Arrays are, for instance, the only collection class that cannot be implemented in Blue itself. They are an intrinsically lower level construct than other collections. We see arrays not so much as a collection class but rather a means for the *implementation* of collection classes. Two educational considerations have led us to the decision to provide predefined arrays: firstly, we wanted to enable students to learn to implement their own collection classes (for instance, in a data structures course) without relying on other collection classes. Predefining the array makes clear that it is a lower level construct. Secondly, we wanted students to become familiar with the concept of arrays to prepare them for other languages that might follow Blue. Most languages support arrays as a basic data type, and we considered preparation for this essential. This is a case where we did not decide for maximum uniformity or simplicity, exactly because this is an educational language. In a modern general purpose language we would regard the provision of arrays through class libraries rather than as predefined classes a cleaner solution.

6.3.4 Type conformance

A variable *a* may be assigned to a variable *b* if the type of *a* conforms to the type of *b*. A type *A* conforms to a type *B* if it is either the same type or a subtype of *B*.

Subtypes are established by inheritance. Inheritance always creates both a subclass and a subtype relationship. These cannot be separated in Blue. Inheritance is defined in a way that ensures substitutability of subclasses for superclasses, and thus establishes a true subtype relationship. Consider the following definitions:

```
class B is
  ...
end class

class A is B
  ...
end class

var
  a : A
  b : B
```

These definitions declare a class *A* as a subclass of class *B* and two variables of these class types. Consider further the following assignments:

```
b := a
a := b    -- ERROR
```

The first assignment is legal (*a* conforms to *b* because *A* inherits from *B*), while the second one is not. Inheritance is discussed in detail in section 6.13. Type conformance with generic classes is discussed in section 6.14.4.

6.4 Classes

All code in Blue is written in classes. Classes are the basic unit of structuring code. Typically, code is displayed on a class basis. All routines which are part of a class are displayed together. (Note that this does not necessarily mean that each class is stored in one file. While this might be true, it is an implementation issue. Files do not exist as a concept in the Blue environment.) A Blue user may open a class and view its code. The storage technique used for the class is irrelevant. What is important is that each view of a class shows all routines that belong to a class, and only those that belong to that class.

6.4.1 Constructor classes

User-defined classes can be constructor classes or enumeration (manifest) classes. This section describes constructor classes.

Each class has a rigid structure. Figure 6.3 shows an overview; Blue keywords are shown in bold. The syntax follows the Pascal/Eiffel tradition that requires a fixed

sequence of syntactical elements, rather than the C/C++ tradition that allows the definitions of types, variables and routines in arbitrary order.

```
class classname is superclass
    == class comment
uses other classes
internal
    var
        variable declarations
    routines
        internal routine definitions
interface
        creation routine
    routines
        interface routine definitions
invariant
    class invariant
end class
```

Figure 6.3: Structure of a class

Requiring a strict order of class elements provides a solid framework in which to learn a good programming style. It makes it easy to find definitions when they are needed, and it makes it easy to decide where to put them. It is possible to provide a template for a class containing this framework to a user starting to write a new class. This greatly eases the difficulties of beginners when writing their first classes.

It is our experience that in every concrete teaching situation an attempt is made to structure code in a similar way anyway. Typically students are told to follow style guidelines that define where and when to write declarations and code. The use of style guidelines is a half-hearted attempt to convince programmers to use good practice. With beginners, whom we want to convince to use good practice, sometimes against considerable resistance, having a system that enforces a style that we consider good makes life much easier. A teaching system does not need to provide the greatest flexibility. It should rather decide what good programming practice is, and then it should enforce the use of it wherever possible. The class structure is only the first of several examples of this in Blue.

6.4.2 Enumeration classes

Language designers and users have long argued about whether enumerations should be included in a programming language. With the advent of structured programming and Pascal, it seemed accepted that they are a useful construct that increases code quality. Wirth, the creator of Pascal, however, seems to have changed his mind: his latest language, Oberon, does not include enumerations.

Opinions in published comments range from statements that enumerations are "useless" and that "programmers do not want to use them" [Pohl 1988], and that they are "superfluous" [Delft 1989] and "excess redundancy" [Lins 1990] to strong defences that state that they result in less work and more understandable and maintainable code [Cashman 1991, Sakkinen 1991].

We strongly believe that the inclusion of enumerations enables programmers to write more readable code. A routine which might fail to complete its task, for example, can return a meaningfully named result value, instead of returning an integer number. The inclusion of enumeration types in object-oriented languages, however, was typically awkward (as in C++, where they do not fit into the object model) or non-existent (Eiffel, Java). Blue, through the introduction of manifest classes, combines enumerations with objects and provides a unified model.

The structure of enumeration classes is shown in Figure 6.4.

class *classname* **is Enumeration**
    == *class comment*

    manifest *enumeration list*

end class

Figure 6.4: Structure of an enumeration class

"Enumeration" is a predefined abstract class. It provides (amongst others) predecessor and successor routines, which can then be used for all enumeration classes. For example, given the class

```
class Colours is Enumeration
    == some sample colours

    manifest red, green, blue, yellow, purple

end class
```

the following loop could be written:

```
col := red
loop
    exit on col = nil
    process (col)
    col := col.succ
end loop
```

Currently no further routines can be defined by the user for enumeration classes. Whether a language could be defined where user-defined manifest classes can have user-defined methods warrants further investigation.

6.4.3 Encapsulation

Blue provides, as most other object-oriented languages, information hiding through encapsulation. In most languages, encapsulation is based either on objects (e.g.

Smalltalk, Eiffel) or classes (e.g. Simula, C++) with Beta being a notable exception – encapsulation in Beta may be defined explicitly and is not necessarily bound to the class structure.

Blue supports class based, rather than object based, encapsulation. This means that objects of the same class can access each other's internal data. The internal part of a class is only protected from access by objects of another class.

This decision reflects the goal we have in mind with the education of students with Blue: software engineering. We want students to learn to understand software engineering issues, and try to provide a language in which those can be learnt.

From a software engineering perspective, information hiding is important for program maintenance. It guarantees localisation. The area of visibility of an instance variable, for example, is the class in which it is defined. This localisation is crucial for enabling changes in the implementation of programs. It defines the part of the program that may be affected by a change made by a maintenance programmer.

For this, in our view, most important aspect of information hiding, only class based encapsulation is necessary. Accessing data of an object from within another object of the same class is not a problem, since changes in the internal structure affects both objects anyway (since they are of the same class). The argument that encapsulation is needed to protect one object's implementation from internal changes in another object does not hold for objects of the same class.

On the other hand, access to internals of other objects of the same class allows us, for example, to write "clone" routines, which produce an object as a copy of another object. Since this cloning might involve copying of data that is not available through the interface, access to the internal part of the other object is crucial. We might use this to write, for example:

```
new_object := old_object.clone
```

We will argue below (section 6.12.3) that the ability to write such clone routines is needed since cloning cannot generally be automated in a programming language.

6.5 Routines and parameters

6.5.1 Structure

Routine declarations follow a similarly strict structure to the class as a whole. The skeleton of a routine is shown in Figure 6.5. The reasons for enforcing a strict structure are the same as for the class in general: it makes it easy to find declarations and enhances readability. It has been argued (for example in [Stroustrup 1991]) that declarations of variables should be permitted anywhere in the code. This would enable the programmer to declare variables at the point of the first assignment, thus avoiding the declaration of uninitialised variables. While the idea of avoiding

uninitialised variables is a good one, scattering variable declarations through the code is strongly detrimental to readability. Often, when modifying a code segment, a programmer has to check the name or type of a variable. Having a position within the code for declarations that can easily be located helps program maintenance. Blue enforces a separate variable declaration section and employs a separate mechanism to deal with the issue of uninitialised variables (see section 6.8: Variables).

routine-name (*parameter-list*) –> (*result-list*) **is**
    == *routine-comment*
  pre
    precondition
  var
    variable declarations
  do
    routine body
  post
    postcondition
  end *routine-name*

Figure 6.5: Structure of a routine

Routine name

The first word of a routine declaration is the name of the routine. It is not preceded by a keyword or symbol. This, together with the recommended indentation, lets the routine names stand out, making it easy for a programmer to find a routine declaration. Elevated to a principle, we could call this an example of "syntax structure for human readability" (as opposed to readability for a compiler, which is so often the motivation for syntactic constructs in programming languages). Since readability improves maintainability, which in turn is a major software engineering concern, we could also claim this to be "syntax structure for maintenance support" or "syntax structure for software engineering". To see the effect of this structure, let us compare it to the C++ syntax, where the routine name can be preceded by several other definitions:

```
// C++ declarations:
virtual const int* get (int n);
static void put (int i, char c);

-- Blue declarations:
get (n: Integer) -> (res: Integer)
put (i: Integer, s: String)
```

Both examples declare the routines "get" and "put". Comparing the two examples, it is obvious that, when searching through a class interface for the declaration of a routine, the Blue format is more convenient. In the Blue version these names are easy

to locate while quickly scanning down a page. The C++ format makes it much harder to identify the routine name in the definition.

The routine name is repeated at the end of the routine body. Again, this can increase readability, especially if routines are longer than just a few lines. Many teachers have recommended this as a programming style using many different languages (usually as a comment at the end of the routine). Again, styles should be supported by the compiler if possible to emphasise their importance and to form good habits. The inconvenience of additional typing can be avoided by providing an editor which supports the automatic insertion of the name at the end.

Routine comment

The routine comment is compulsory. Blue will report an error if no comment is present. (See section 6.7 for more details on comments.)

Pre and post conditions

Pre and post conditions offer a formal mechanism to specify (partially) the constraints and effects of a routine. They are supported through specific keywords at the beginning and the end of the routine body. Although the behaviour of pre and post conditions can be simulated with most other languages by using comments and assertions, their support as separate language elements emphasises their importance and leads to a much more conscious approach by students in dealing with this issue.

The use of pre/post conditions is also supported by the environment. When a class is first created, a class skeleton for that class is automatically generated. This class skeleton contains a first routine skeleton. The editor then supports the insertion of additional routine skeletons as a built-in editor command. Those generated skeletons include definitions of pre/post conditions. Our experience shows that, once a default placeholder is there, students often think about and write an appropriate condition, rather than deleting the existing default.

Details of pre and post conditions and the "design by contract" principle are discussed in section 6.6.

routine-name (*parameter-list*) -> (*result-list*) **is**
        == *routine-comment*

    pre
        precondition

    post
        postcondition

Figure 6.6: Structure of a routine in interface view

Routine interface

The routine signature, the comment and the pre/post conditions are part of the routine interface. The post condition is located at the end of the routine when looking at the implementation. This reflects the logical sequence of evaluation: the post condition must hold after the execution of the body. In the interface view, however, (which is automatically generated by the environment) the post condition appears directly under the pre condition. Figure 6.6 shows the structure of a routine interface.

6.5.2 Parameters and result variables

Routines have an optional parameter list and an optional list of result variables. Only one kind of parameter passing mechanism exists: all parameters are passed by value. (Remember, though, that all variables hold references – the *reference* is passed by value, resulting in pass-by-reference semantics for the objects themselves.) Result lists are, as their name suggests, lists of variables. Thus a routine can return more than one value. Consider the following routine.

```
findElem (index: Integer) -> (found: Boolean, elem: Element) is
     == Return element at 'index'. If it exists, 'found' is true
     == and 'elem' is the element. If not, 'found' is false and
     == 'elem' is nil.
  var
     e : Element
  do
     ...            -- code left out
     found := true
     elem := e
  end findElem
```

Here, the routine returns two values, *found* and *elem* (which may be useful if *nil* is a valid value for *elem*, and thus cannot be used to indicate failure). The result values are named (defining a *result variable*) and values are returned by assigning to those variables. The compiler reports an error if the routine body does not contain an assignment to each result variable, and the runtime system reports an error if any of the variables are undefined at the time of the return of the routine.

This mechanism provides a substantial simplification compared to many other languages. There is only one way to pass information into a routine, and there is one mechanism to get information back. Many languages provide different kinds of parameters (value, reference, in/out parameters) in addition to function results. This creates unnecessary complications for students. Conceptually, it is hard to justify why a function returns one value as a function result, but if it wants to return two values, it must use reference parameters (or one reference parameter and a function result). Two different mechanisms are used to serve the same purpose. Providing one kind of parameter and one kind of (multiple) return mechanism overcomes these inconsistencies and reduces the number of required concepts.

Purists sometimes argue that a function should return only one result. If several pieces of information are needed, the function should return an object containing that information. While this is possible, the overhead in practice is relatively high.

Creating an extra class for the result objects is necessary, and in reality is often avoided. We believe that there is no justification for forcing the programmer (even a beginner) to add additional complexity if a clean, simple mechanism can be provided to achieve the same task.

6.5.3 Multi-assignments

To be able to return multiple values from a routine call, it is necessary to have a way to receive multiple values. The multi-assignment provides this mechanism.

Assignments have a list of variables on their left hand side and an expression list on their right, e.g.

```
a, b := 42, 99
```

Each of the values on the right is assigned to the corresponding variable on the left. All expressions on the right hand side are evaluated before any assignment takes place. A routine call with multiple return values evaluates to a value list and can thus be used on the right hand side of a multi-assignment. The *findElem* function defined above, for instance, may be called in the following manner:

```
success, element := findElem (2)
```

The multi-assignment is also useful in another situation. Swapping values of two variables becomes very easy:

```
a, b := b, a
```

The assignment is the only place where a routine call which returns multiple values is legal. In lists of actual parameters, where technically a routine call resulting in multiple values could be allowed to represent multiple actual parameters, we decided against allowing those calls. We considered that case more likely to result in confusion than usefulness.

The parameter and return value mechanism together with the multi-assignment has several advantages:

- It reduces the number of concepts (avoids redundancy).
- It increases readability of the routine declaration.
- It clarifies semantics of the routine call.

The first point has been mentioned above: by avoiding different kinds of parameter passing mechanism, the number of (partly redundant) concepts is reduced.

The second point may be open to discussion, but we believe the Blue style to be more readable than, say, the Ada95 style of parameter declarations (Ada95 was chosen for comparison here, because it places emphasis on syntactical clarity of parameter passing modes). Consider:

```
{ Ada: }
procedure m (a: in out Integer; s: out string, b: in Boolean)

-- Blue:
m (a: Integer, b: Boolean) -> (new_a: Integer, s: String)
```

Blue forces an ordering onto the parameter list: *in* parameters are named first, results (*out parameters* in Ada terminology) are separated. Ada allows arbitrary ordering of parameters. Supplying two separate lists is more easily readable than mixing declarations in one list. Note that the Ada *inout* mode of parameter *a* is represented in Blue by two variables: *a* in the parameter list and *new_a* in the result list.

A more important detail is the third point: the routine call. Consider the Ada case:

```
m (p1, p2, p3);
```

This example shows a call of the routine declared above. At the location of the call it is not determinable which of the parameters are *in*, *out* or *inout* parameters. Compare this with the Blue alternative:

```
p1, p2 := m (p1, p3)
```

Here, it is clearly determinable at the routine call that *p1* and *p2* are changed by the routine, while *p1* and *p3* are passed in (*p1* thus has an *inout* behaviour).

The argument is not merely readability or convenience. The Blue style adds useful semantic information to routine calls without introducing additional language mechanisms (in fact, while reducing the number of used mechanisms). This is an additional quality not found in any of the languages surveyed.

6.6 Design by contract

The “design by contract” principle described by Bertrand Meyer in his introduction to Eiffel [Meyer 1988] is a powerful mechanism to encourage clear specifications of routines and clean programming principles. It also often enables the early detection of program errors. All of these aspects are strongly desirable for a teaching language, and Blue provides language constructs to support these principles in a manner very similar to Eiffel.

The language elements needed to support these principles are pre and post conditions and class invariants.

6.6.1 Pre and post conditions

Pre and post conditions are part of a routine definition. They have been briefly mentioned in section 6.5, where their position in relation to other elements in the routine specification was shown. Pre and post conditions are optional. They typically consist of a boolean expression which is checked at runtime and results in a runtime error if it fails. If such an error occurs, the source of the class containing the condition

is displayed, the condition is highlighted, and an error message indicates that a pre or post condition was violated.

A condition consists of a single boolean expression (as opposed to conditions in Eiffel, which consist of a list of expressions). This is technically equivalent, since multiple conditions can easily be concatenated into a single expression by joining them with a logical "and" operator. This was done to maintain an unambiguous context-free grammar for the language. (Since Blue does not use semicolons at the end of statements or conditions a list of boolean expressions may lead to ambiguity in the grammar.)

Another difference between Blue and Eiffel is that in Blue pre and post conditions may contain comments. These may be used for conditions that cannot be expressed in source code (e.g. that a particular item has been sent to a printer, that some information was written to the terminal, etc.). To be exact, pre and post conditions in Eiffel *can* contain comments, but these comments do not become part of the interface of the routine, and thus are not considered to be part of the pre or post condition. In Blue these comments form part of the routine interface - see section 6.7. Conditions are always checked at runtime. There is no mechanism to switch checking of pre and post conditions off.

As in Eiffel, pre and post conditions are inherited by subclasses and can be modified only in ways that ensure that the subclass still meets the superclass specification.

6.6.2 Class invariants

A class invariant is a boolean expression, defined at the end of a class, that must be true at all stable states of the class. A stable state is each state in which the class could be observed from the outside – more precisely: a stable state exists at the beginning and the end of every externally invoked routine call (see [Meyer 1988, p158] for a more detailed discussion of invariants and stable states).

As opposed to invariants in Eiffel, the invariant in Blue is never part of the interface of the class. Eiffel distinguishes between internal and external invariants. Conditions which use internal instance variables or routines are considered implementation conditions and do not appear in the interface ("short form" in Eiffel terminology) of the class. Conditions which only contain references to public variables or routines ("features" in Eiffel) are considered public invariants and become part of the "short form".

Interface invariants mainly serve the purpose of making statements about instance variables. Statements about routine values need not be included in the invariant, since they can be made in the post condition of the routine. The *uniform access* principle that Eiffel states is broken in this respect. Consider a feature "count" in an Eiffel list class:

```
count : Integer
```

In Eiffel, this feature could be represented by either a routine (a parameterless function) or an instance variable. This is known as the principle of uniform access – it should not be observable from the outside how the feature is implemented. This gives the implementor freedom of choice and the possibility of later changing the implementation without affecting clients. We might add the invariant

```
invariant
  count > 0
```

If count is indeed implemented by a variable, this is the only way to express this constraint. If count is implemented by a routine, however, we might alternatively write:

```
count : Integer          -- Eiffel code
  ensures
    Result > 0
```

In this version, the same constraint is expressed in the interface by a post condition rather than an invariant. Here the uniform access principle breaks down, since Eiffel does not allow post conditions to be expressed for variables. This is in fact the whole purpose of interface class invariants: to express those constraints for variables that would have been expressed in post conditions if it were implemented as a function. (It would, in fact, be more consistent if Eiffel allowed those conditions for variables and removed invariants from the interface. This would textually move the constraints closer to the feature to which they are referring, and uphold the principle of uniform access.)

In Blue, since variables cannot appear in the interface, there is no need for invariants to appear in class interfaces. All invariants are implementation invariants used for error checking rather than for specifying class semantics. They appear in the implementation view only.

6.7 Comments

All comments in Blue are line oriented. They start from the comment symbol and extend to the end of the line. There are no block comments in Blue.

This definition avoids misunderstandings through long comments. Using a block comment over a larger area might be misleading if a maintenance programmer looks at lines of code in the middle of the comment. The start and/or end of the comment might be outside the currently visible screen area, and the code might give a wrong impression, namely that source code is present while in fact it is commented out. Line comments avoid this problem. Since every single line needs to be preceded by the comment symbol, commented sections are immediately recognisable.

The typical criticism of this style is that it is tedious to insert or remove long blocks of comments in this way, e.g. when a whole routine is to be commented out. This, however, is a pure editing problem, and it should be treated as such. To overcome it, the editor should provide convenient ways to insert and remove blocks of comments. In the Blue environment, a block of source code may be highlighted and (un)commented with one keystroke.

Editing problems should not be solved by compromise in language design, especially if clarity and readability may suffer as a result.

6.7.1 Compulsory comments

Blue enforces the presence of some comments in a class. A comment has to be present at the beginning of each class (below the header) and in the header of each routine. The purpose of these comments is to describe the entire class and each routine, respectively. While the compiler actually enforces only the presence of a comment symbol (it can never enforce the presence of a meaningful comment), and thus this enforcement is theoretically almost meaningless, it serves in practice as a strong encouragement to write comments. Enforcing the presence of a comment sends a strong signal to the student, indicating that comments are considered to be an intrinsic part of the class. In our experience, students take language elements much more seriously if the compiler supports them, than they do if only given as style guidelines.

```
class Person is
==================================================
== Author:      M. Kölling
== Version:     1.1
== Date:        12 July 2001
== Short:       Person class for university management project.
==
==  The class Person implements objects representing a person in a university
==  management project. It contains information common to all persons in
==  the university ...
==
==================================================
...
end class
```

Figure 6.7: Format of a class comment

Writing comments is further encouraged by the environment. The automatically generated class skeleton contains comment patterns to be filled in, and some comments are displayed in the interface view (see interface comments, next section).

The class comment also includes four keywords that are recognised by the environment: *author*, *version*, *date* and *short* (Figure 6.7). They are used to provide information about author, version and modification date of the class as well as a one line description. These specifications, when present, are recognised by the class browser and can be used for specific searching of the class library. A search can be made, for instance, for classes written by a specific author. When a class is displayed in the browser, these specifications are displayed in appropriate fields.

6.7.2 Interface comments vs. implementation comments

Blue supports two kinds of comments. *Interface comments* are defined by the double equals sign (==). Their purpose is to provide information about the functionality of the class or its routines. Interface comments may appear only at specified locations in the source: in the header of the class, in the header of each function and in pre and post conditions. They are visible in the interface of the class. Implementation comments are defined by a double minus sign (––). Their purpose is to provide information about the implementation of a class. They may appear anywhere in the source and are ignored by the compiler. Both kinds of comments are terminated by a new line.

Figure 6.8 shows the implementation view of a routine that makes use of interface and implementation comments. Figure 6.9 shows the interface view of that same routine.

```
printInfo (cols: Integer) is
        == Print out the information in this container formatted into 'cols'
        == columns on screen. Items are sorted alphabetically.
        – – To print the information the internal tree is traversed and each
        – – node is displayed separately. To achieve alphabetical sorting, items
        – – must be processed in infix order.
    pre
        cols > 0
    do
        ...     – – code goes here (left out for space reasons)
    post
        == Items have been displayed on screen
    end printInfo
```

Figure 6.8: Interface and implementation comments in implementation view

```
printInfo (cols: Integer)
        == Print out the information in this container formatted into 'cols'
        == columns on screen. Items are sorted alphabetically.
    pre
        cols > 0
    post
        == Items have been displayed on screen
```

Figure 6.9: Routine in interface view

6.8 Variables

All variables, if they have a value at all, hold references to objects. In addition to that, they may be *undefined* or *nil*, as explained below.

6.8.1 Undefined variables

Variables may or may not be initialised at declaration. For example:

```
var
  name : String
  n :    Integer := 42
  cnt :  Integer := getNumElems (1, true)
```

Variables are a combination of type, state and value. If a variable is not initialised, its state is *undefined*. An attempt to use an undefined variable is noticed at runtime and results in a runtime error. The use of a variable is its appearance on the right hand side of an assignment or in an actual parameter list. Routine results are checked at the time of return from the routine.

This catches a common error made by beginning students. Checking for uninitialised variables is a case where Blue, by concentrating on a teaching situation, can provide better support for students than other systems. Undertaking the checks costs in both time and space. In a language intended for commercial production, efficiency would take precedence over error checking, and those checks would not be justifiable. (In fact, the only other language known to us which defines such checking is CLU, but this part of the language definition was never implemented in the compiler [Liskov 1992]. Java defines related, but slightly different semantics: access to variables which *may* be uninitialised is considered an error. The checking is done by using compile time flow analysis, and cases that cannot be decided at compile time are simply disallowed.)

Alternatives to this mechanism would have been to force initialisation at point of declaration, or to automatically initialise all variables to some default value. Both schemes have drawbacks.

Forced initialisation

Forcing initialisation effectively means that declarations should be allowed anywhere in the code. At the point of Blue declarations (before the start of a routine) sensible initial values are often not known, thus making forced initialisations meaningless (because they would only force the programmer to initialise variables to some meaningless value). Meaningful initialisation values are often only known halfway through a routine's body. To achieve meaningful variable initialisation at declaration point, the declaration must be allowed at the point at which the variable is first used. C++, for instance, follows this approach.

We have already argued that variable declarations scattered through the code are detrimental to readability. (This is magnified in languages with nested blocks that

allow multiple nested declarations of the same variable name.) C++, by allowing scattered declarations and *not* demanding variables to be initialised (together with the inability to check for the use of uninitialised variables) provides the worst of all worlds.

Automatic initialisation

The other alternative, automatic initialisation, also is not appropriate for a teaching language. There are three possible scenarios, all of which are negative:

1. The user forgets to initialise the variable; the variable is automatically initialised to a value that is semantically *illegal* in the context (e.g. *nil* for a variable that is expected to refer to an object). In this situation, the initialisation is no better than no initialisation at all. The program will fail, as it would have without any initialisation.

2. The user forgets to initialise the variable; the variable is automatically initialised to a value that is semantically *legal* in the context (e.g. 0 for an integer variable). In this case, the user will be "protected" from detecting a program error. Because the value is legal, it will not result in an (immediate) error, but it may nonetheless be wrong and cause wrong results. It is only detected later. This is the worst case for a teaching language, because programming errors may go unnoticed altogether.

3. The user intentionally omits the initialisation of the variable because the default value is the intended initial value (i.e. the user exploits the automatic initialisation). The only real effect of this case is to save typing of a few characters (the explicit initialisation), thereby reducing readability (since the reader must know the default value for each particular type of variable). It is also not clear whether the initialisation was consciously or unconsciously omitted. It seems worth the effort on the writers part of including the initialisation explicitly in these cases to clarify the semantics to the reader.

The Blue mechanism combines the best features of all schemes: all declarations are placed in a well-defined location, variables may be initialised if a sensible initial value is known (with the initial value clearly readable in the source text) and the use of uninitialised variables is always detected.

6.8.2 Nil

A variable of any type may hold the special value *nil*. Since *all* variables hold references to objects, all variables, including simple types such as Integer or Boolean, can hold the nil value. Nil indicates that the variable currently does not hold any

reference.[7] Nil differs from all other legal values of the type (i.e it is not, for instance, 0 for integer variables).

Nil therefore is special, since it does not seem to conform to the strongly typed system that Blue otherwise enforces. Consider:

```
var
  a : Integer
  b : MyClass
...
a := nil
b := nil
```

The type of nil cannot be determined - it is assignment compatible with all types. Most other languages face the same issue, and there are different technical tricks to provide an answer to this problem. (Note that this only poses a problem for the theoretical foundation, the type system. It is not normally a problem in practical application.)

Blue, like Pascal, uses an overloaded nil value. Nil is considered to represent a value of every type (therefore a potentially unlimited number of nil values exist) and the overloaded assignment automatically assigns the nil value of the right type to the variable.

Beta, on the other hand, defines a class named “NoClass”, which implicitly inherits from all existing classes. Nil is of this class and therefore, by subtyping rules, assignment compatible to all types. Eiffel uses the same construct with a class named "None".

An early development version of Blue answered this question by using the *state* and *value* distinction. A variable has a state in addition to its value. We have already discussed the *undefined* state. A variable has the state *valid* if it has been assigned a value. (Once a variable leaves the *undefined* state, it can never return to that state. No test for *undefined* exists. A programmer in a well written program always knows when a variable is undefined.)

The earlier version considered *nil* to be a state rather than a value. This subtle distinction avoids the type problem. To meet the requirement of consistent syntax (“same syntax for same semantics, different syntax for different semantics”), however, the change to the nil state should not be written as an assignment (since it is a semantically different operation). We therefore defined built-in commands to set and test for nil:

---

[7] Note that this does not mean that simple types must be implemented as a reference type. In fact, in the current Blue system they are not. However this is an implementation optimisation and does not affect the conceptual model.

```
set_nil (a)
if is_nil (a) then
  ...
end if
```

A side effect of this definition was that nil could not be easily passed as a parameter. This problem prompted us to view nil as a value rather than a state, and to accept the overloading of its type.

6.9 Statements

The statements in Blue are

- assignment
- assignment attempt
- procedure call
- return
- assertion

and control structures (which are discussed later).

6.9.1 Assignment and assignment attempt

The assignment is a multi-assignment and was discussed in section 6.5.3. An assignment attempt is written with a `?=` symbol:

```
a ?= b
```

The assignment attempt uses the same symbol and the same semantics as Eiffel [Meyer 1992, p 330]. It is used to attempt an assignment from a superclass variable to a subclass variable. Consider:

```
class A is
  == This is the superclass
  ...
end class
```

and

```
class B is A
  == B inherits from A
  ...
end class
```

If we have variable declarations:

```
var
  a : A
  b : B
```

then an assignment

```
a := b
```

is legal because of the subtype relationship. (A subclass in Blue also establishes a subtype relationship. Subclassing and subtyping are inseparably linked – see section 6.13: Inheritance.) An assignment

```
b := a
```

on the other hand, is illegal because it is not type safe. The assignment attempt, however,

```
b ?= a
```

is legal. On execution the dynamic type of `a` is checked, and if it is of type B or one of its subtypes the assignment is executed. If its dynamic type does not conform to B, then nil is assigned.

We debated for a long time whether the assignment attempt should be a part of Blue. There were doubts about its usefulness and necessity in a language. One can argue that usually, when an assignment attempt is used, the program is badly designed. Improving the design can, in most cases, lead to the elimination of the assignment attempt from the code. Let us consider an example.

Assignment attempts are necessary when a programmer knows or suspects that an object is of a subtype of the statically known type, and needs to access operations of that subtype. This is the case if we use a polymorphic list. As an example here, we can use a list of animals in an environmental simulation. "Animal" would be an abstract superclass of, say, the classes "Shark", "Sheep" and "Lion". The simulation would hold a list of animals, and once in every time interval it would go through the list to activate each particular animal to do whatever it is supposed to do. Let us assume that sharks swim, sheep eat and lions hunt.

In a naïve implementation, we might need to "cast" the animal object back to its dynamic type in order to call the routine `shark.swim`, `sheep.eat` and `lion.hunt`. The following code fragment illustrates this technique:

```
animals.initScan
loop
  anAnimal := animals.getNext
  exit on anAnimal = nil            -- end of list
  aShark ?= anAnimal                -- try shark
  if aShark <> nil then
    aShark.swim
  else
    aSheep ?= anAnimal              -- try sheep
    if aSheep <> nil then
      aSheep.eat
    else
      aLion ?= anAnimal             -- try lion
      if aLion <> nil then
        aLion.hunt
      end if
    end if
  end if
end loop
```

This is unfortunately the way in which the assignment attempt is mostly used by inexperienced programmers, and it is indeed bad design. This pattern is inelegant, unnecessarily complicated and a hindrance to system maintenance and extendability. It does not make use of one of object-orientation's most powerful features: dynamic dispatch. For this example, a better design would use an "act" routine in the abstract class "Animal" (as a deferred routine), call the "act" routine for each animal in each time interval, and leave it up the animal itself to determine what kind of action it wishes to perform (swim, eat or hunt). Here is the modified code.

```
animals.initScan
loop
  anAnimal := animals.getNext
  exit on anAnimal = nil           -- end of list
  anAnimal.act
end loop
```

This code is clearly shorter, more elegant, and does not need to be changed when new animal subclasses are introduced to the system.

The argument so far suggests that we exclude the assignment attempt from the language. It is regularly misused and seems to invite bad design.

There are, however, valid uses of the assignment attempt. An example is found in the collection class library in the Blue system (a standard library that is part of the system distribution). Here we have an abstract class "List" with two subclasses, "LList" and "IndexList". "LList" and "IndexList" are different implementations of the List type (a linked list implementation and an array implementation, respectively). Both share a common interface.

One of the routines in the List interface is "append". This routine joins two lists into one by appending one list to the other. The parameter to this routine is of type "List". This means that not only two array lists or two linked lists may be merged, but also an array list can be appended to a linked list and vice versa.

In the implementation of this routine, the assignment attempt is used. This allows the code to distinguish between the two possible parameter types (linked list or array list) and to use different techniques to join the two lists. Using an assignment attempt could have been avoided by using the common interface operations to access the list elements separately. This, however, would have significantly affected performance of the operation. It would have resulted in the provision of a routine with exponential runtime behaviour in a standard collection library where a routine with constant execution time should have been available. This is not acceptable.

Note that we have previously argued that performance is not of paramount concern in our system. This, however, applies to *constant* performance degradation of student programs, not to bad programming. We can accept a system that executes programs, say, 20 times slower than a professional system, but we cannot accept a system that uses exponential algorithms where linear or even constant time algorithms should be used. The reason is that this (order of execution time) is something that we might want to teach to our students. It is important that our own libraries then make use of good algorithms.

For the assignment attempt, this means two things:

- It is not used very frequently, and it can easily be misused.
- There are some examples where its use is necessary.

The problem with excluding the assignment attempt from the language, though, is that its effect cannot be achieved with any combination of other existing constructs. The argument, used at other places in this discussion, that infrequently used constructs should be excluded because their replacement by a (less convenient) combination of other constructs is acceptable, cannot be applied here. The assignment attempt is not only a convenient form to express an operation, it is the only form available for these semantics.

We have therefore decided to include the assignment attempt in the language, although this is one of those rare occasions where students must be warned to think very carefully when they use it, about whether its use is really necessary.

6.9.2 Procedure call

Procedures (routines without return values) may be called by stating their name and parameter list. For example:

```
printValues (32)

initData

drawRect (4, 4, 100, 100, true, red)
```

The actual parameter list must match the formal parameter list in the routine definition. If no parameters are present, no parentheses are used.

6.9.3 Return from routine

A return statement may be used anywhere in a routine to exit immediately from that routine:

```
show is
    == display this element on standard output

  do
    if name = nil then
      return
    end if
    ...
  end show
```

If the routine has result parameters, all the result parameters must be assigned before returning from the routine. While some people have argued against allowing a return from the middle of a function we are convinced that it can in many cases lead to a simplification of code structure. Some examples of this are shown in [Roberts 1995]. For first year students, and in fact, for general programming, it is counter productive to force the construction of code patterns more complex then necessary for the sake of theory (since theory, namely formal proof of correctness, is the main argument against

allowing this construct). We will continue this line of argument in section 6.11.3, when we discuss similar criticism in the context of loop exits.

6.9.4 Assert statement

Assertions serve the purpose of supporting correctness and increasing locality of error detection. By using assertions, errors in programs can be detected in some cases that might otherwise go unnoticed for a longer time. They also help in detecting errors earlier, thus indicating the error closer to its real source. This can greatly reduce debugging times.

Assertions take a boolean expression as a parameter and cause a runtime error if that expression evaluates to false. For example:

```
assert (head <> nil)
assert (obj.isValid)
```

The semantics of assertions in Blue are the same as in many other languages or libraries. Their inclusion in the language encourages reasoning about correctness and good programming style.

6.10 Aliases

An interesting issue arises from conflicting goals of two different requirements: on the one hand we want uniformity and a small number of underlying concepts. On the other hand we want the language to be simple and intuitive. There are cases, especially when pre-existing knowledge is involved, in which these two goals can come into conflict.

An example is the treatment of simple types and mathematical operations. We have argued, on the one hand, that simple data items should be objects, and that they should be treated as such. This leads to a uniform object concept which is advantageous for understandability. Convention, on the other hand, uses infix notation for some mathematical operations. When using those operations, these two goals contradict each other. Adding two numbers m and n, for example, should, according to the first principle, be represented as:

```
m.add (n)
```

This, however, looks unfamiliar to students, although they are comfortable with the concept of addition. The following notation is more familiar:

```
m + n
```

The attempt to make a language easy and intuitive should take existing knowledge into account and exploit it to achieve its goal. A simple program, for example, such as “Hello world” or a program adding two numbers, should be easy to write without too many unnecessary explanations. Achieving that goal would be easier if the familiar infix notation were to be used.

To overcome this apparent conflict, Blue uses *aliases*. Aliases provide an alternative syntax for some constructs. In the case above, for example, both alternatives are legal. The first one (`m.add (n)`) is the "real" notation that conforms to the object model. The second notation (`m + n`) can be regarded as a shorthand notation for the first one.

Another example of an alias is a print statement to perform text output operations. A *print* command is available for this purpose:

```
print ("Hello world.")
print ("The answer is: ", n)
print ("a=", a, "  b=", b, "c=", c)
```

This print command is an alias that calls the *write* operation of the *terminal* object. (*terminal* is a predefined constant which refers to an object of class *TextTerminal*. This object controls the standard I/O terminal.) Thus the three print instructions listed above are shorthand notations for the following statements:

```
terminal.write ( str ("Hello world.") )
terminal.write ( str ("The answer is: ", n) )
terminal.write ( str ("a=", a, "  b=", b, "c=", c) )
```

The operation `str` used in this statements is itself an alias that is a shorthand for calling the *toString* and *concat* functions on its parameters. For example:

```
str (a, b, c)
```

is an alias for

```
a.toString.concat (b.toString.concat (c.toString))
```

In other words, the `str` function converts all of its arguments to strings and concatenates them to form one result string[8]. This string may then be printed out, using the terminal routine *write*, which expects one string parameter.

The concept of aliases serves several purposes. Aliases can be initially introduced to students as statements in their own right. A student can, for instance, be shown how to write a hello-world routine:

```
helloWorld is
    == Print the string "Hello world" to standard output
  do
    print ("Hello world")
  end helloWorld
```

The meaning of this instruction is immediately and intuitively clear. Equally, adding two numbers is easy to understand:

[8] The *str* alias may be used with every type that defines a *toString* operation. *toString* is defined for all predefined classes and may be defined in user-defined classes.

```
add (num1:Integer, num2:Integer) -> (sum:Integer) is
    == Return the sum of num1 and num2
  do
    sum := num1 + num2
  end add
```

Both of these examples can easily be explained to beginners by initially making use of intuition and pre-existing knowledge. The underlying object model is initially hidden, avoiding the need for long explanations or hand waving.

Later, when objects and operations on objects are introduced, and students understand the principle of calling operations on objects, these aliases can be resolved. Students can then be told that those aliases are shorthand notations for normal object operations. This approach provides both an easy entrance into programming and a consistent model where everything falls into place once the student reaches the stage of examining advanced issues.

Aliases are only available for a small number of operations on the predefined classes. Users cannot define their own aliases. Their purpose is only to ease the first stage of programming. The use of aliases as a general, user-defined mechanism would be detrimental to the general readability of the code.

6.11 Control structures

Three control structures are supported in Blue: a conditional (*if* statement), a selection statement (*case* or *switch* statement) and an iteration instruction (*loop* statement).

6.11.1 Conditional

The conditional uses the common *if-then-else* syntax. For example:

```
if n > 0 then
  ...
else
  ...
end if
```

The *else* part is optional. The bodies of the *then* and *else* parts are statement lists, not single statements as in some other languages (Pascal, C++). The advantage of single statement bodies is that an *if* with a single conditional statement can be written without the need to indicate the end of the *if* statement, e.g. in C++:

```
if (n > 0)                // C++ code
  cout << "positive";
```

More than one conditional statement can be grouped with a block symbol (`{}` in C++, `begin/end` in Pascal). The following is a C++ example:

```
if (n > 0) {              // C++ code
  cout << "positive";
  res = 1;
}
```

In Blue, the *if-then* instruction forms an implicit begin of a compound statement that must always be terminated with an *end if* keyword, e.g. for a one line conditional:

```
if n > 0 then
  print ("positive")
end if
```

and for a conditional with more than one statement:

```
if n > 0 then
  print ("positive")
  res := 1
end if
```

Always requiring the *end if* keywords adds uniformity and clarity to *if* statements. Some programmers object to the extra line that is required when an *if* statement has only one conditional line. That inconvenience is easily offset, however, by the substantial gain for beginners in avoiding situations such as the following code in C++:

```
if (n > 0)                // C++ code
  cout << "positive";
  res = 1;
```

In this case, only the first statement is conditional, although indentation suggests that the programmer intended to include both lines in the body of the *if* statement. This error leads to legal (but wrong) code and can thus be hard to debug.

One advantage of single statement *if* statements, however, is the possibility of an elegant syntax for sequential *ifs*. The following code segment (C++)

```
if (...) {
  ...
}
else {
  if (...) {
    ...
  }
  else {
    if (...) {
      ...
    }
  }
}
```

can be conveniently reformatted as

```
if (...) {
   ...
}
else if (...) {
   ...
}
else if (...) {
   ...
}
```

With compound *if* statements, as in Blue, the original structure

```
if ... then
   ...
else
   if ... then
      ...
   else
      if ... then
         ...
      end if
   end if
end if
```

would turn, after an attempt at restructure, into

```
if ... then
   ...
else if ... then
   ...
else if ... then
   ...
end if end if end if
```

The disadvantage, when formatting it as a sequential structure, is the need for the collection of *end if* keywords at the end. To overcome this problem, Blue (as many other languages) uses an *elseif* keyword to replace the two separate *else if* keywords:

```
if ... then
   ...
elseif ... then
   ...
elseif ... then
   ...
end if
```

6.11.2 Selection

The term *selection* refers to a construct that selects one out of several code segments for execution. It is represented in many languages as a *case* or *switch* statement. The Blue selection statement uses the keyword *case* and value sets as case labels:

```
case val of
   {0..10}:    setColour (blue)
               print ("low")
   {11..80}:   print ("medium")
   {81..99}:   print ("high")
   {100}:      setColour (red)
               print ("top")
end case
```

The value after the *case* keyword can be an arbitrary expression. The label sets may contain single values, value ranges (as shown above) or lists of values. In addition to this, a default case may be provided through the *else* keyword. Example:

```
case terminal.getChar of
   {" ", "\n", "\t"}:
        handleWhitespace
   {"a".."z", "A".."Z"}:
        handleLetter
   {"0".."9"}:
        handleDigit
   else
        handleError
end case
```

If the value does not match any of the labels and no default branch is provided, none of the statements nested in the case statement is executed. This is not an error.

This selection statement is very flexible and convenient to use. The real question here is: is it needed?

The selection statement violates at least one of our design principles: avoiding redundancy. Each case statement can also be implemented as a series of if statements. This is a case of conflicting principles where a choice has to be made, based on the perceived importance of each principle in each individual case. The major argument against the inclusion of the case statement is the avoidance of redundancy. The statement is not essential for general programming. However, this must be balanced with the requirements of simplicity, readability and for easy transition to other languages.

Using case statements can, in some cases, lead to more readable code than does the use of chained if statements. It adds expressiveness and convenience that make programming of some selections easier than it would be with if statements. Also, most other languages have a case statement in some form. Since Blue programming is not a stand-alone skill, but is meant as a preparation for the use of other languages, the case statement serves as a preparation for similar constructs in other languages.

Since the goal was to design a small language, but not necessarily the smallest language possible, there is some room for trade-off. We considered the case statement simple enough not to contradict the goal of a “simple”, “easy-to-understand”

language, and decided in this case to follow the argument of convenience and easing of the transition to other languages and included the statement in the language.

This is, however, not an isolated case. Many constructs exist in languages where the arguments may be made both ways in a similar manner. The decision cannot always be the same. If all disputable constructs were included, the language would automatically become large and redundant. If all were excluded, we would specify a minimal language that might be academically interesting, but not a good tool for introductory programming. Designing this language is largely a decision about where to draw the line. There is no golden rule in making this decision. Much comes down to experience, convictions and opinions. We have been largely guided by our experience in teaching first year students, and our own ideas as to what should be taught in an introductory computing course. This way, some concepts (such as concurrency) have been excluded from the language altogether, since we do not want to teach them in our first year course. The goal that the language be small is more important than the attempt to cover every possible aspect. The question as to what the important concepts are that must be included into a first year language would be answered differently by different people. Some institutions, for example, teach concurrency in their first year course. We tried to reach a compromise that we hope is widely acceptable.

This question can be discussed in context with several other constructs. The next section (iteration), for instance, discusses another borderline case where the decision was made the other way.

6.11.3 Iteration

There is only one kind of loop structure in Blue. The loop is defined with the keywords **loop** ... **end loop**.

The loop construct can be used to achieve the semantics of while, repeat and for loops as well as more general definitions (Figure 6.10).

```
loop
    statement-list
    exit on condition1
    statement-list
    { exit on condition2
    statement-list }
end loop
```

Figure 6.10: Structure of a loop

Each statement list can be empty. A loop may contain one or more exit statements. By replacing several loop constructs with a single one, we apply the principle of avoiding redundancy, thus easing teaching and learning of the language, without losing readability. It has also been argued (e.g. in [Roberts 1995]) that internal loop exits

make it easier to implement some common algorithms and should be supported in an introductory language.

The issue of inclusion of an explicit *for* loop was also somewhat contentious. Again, similar arguments to those for the inclusion of the case statement could be made: convenience and the preparation for other languages (many of which have for loops) speaks for it, redundancy speaks against it. The alternatives to these constructs (case statement and for loop) have prompted us to decide differently in these two cases. The alternative to the case statement (chained if statements) is a clumsy programming idiom that can be considered bad programming style (especially considering that some compilers optimise case statements much better than chained if statements. While this efficiency is not paramount in a first programming course, it might be desirable to form the right habits from the start to avoid bad programming styles later).

The alternative to the for loop on the other hand, the use of a general loop with explicit counter incrementing, is a completely acceptable idiom. We felt that nothing would be won through the inclusion of the for loop. On the contrary, the explicit increment of the loop counter might help some students to more easily understand the operation of a loop. In this case we did not see a justification to violate the principle of redundancy.

6.12 Expressions

Expressions in Blue are

- equality (=)
- type equality (is)
- function call
- set membership (in)
- object creation (create)
- self reference (this)

Most of them are very straightforward: they are not very different from equivalents in many other languages. Only some of the more interesting considerations will be discussed here. They concern equality, set membership and object creation.

6.12.1 Equality

The question as to what equality means must be addressed by every object-oriented language. The semantics of equality checks are, in fact, not as simple as it might seem at first glance [Grogono 1993]. There are at least three different possibilities which are in use in different languages: *identity*, *shallow equality* and *deep equality*. The expression $a = b$ can mean any one of these in different contexts in different languages.

Identity means that $a = b$ is true if a and b refer to the same object. It would not be true if a and b refer to different objects, even if all attributes of a and b are the same. In Figure 6.11, a and b are identical. Identity is also referred to as *reference equality*.

Shallow equality means that $a = b$ is true if all the attributes of the objects to which a and b refer are identical. Shallow equality can be implemented easily with a bitwise comparison of the memory space that represents the two objects. Identity implies shallow equality. In Figure 6.11, c and d are "shallow equal", but e and f are not.

Deep equality means that $a = b$ is true if each attribute in a and b is either identical or deep equal. e and f in Figure 6.11 are "deep equal"[9]. Deep equality is implied by both identity and shallow equality. In this sense deep equality is the "weakest" form of equality and identity is the "strongest".

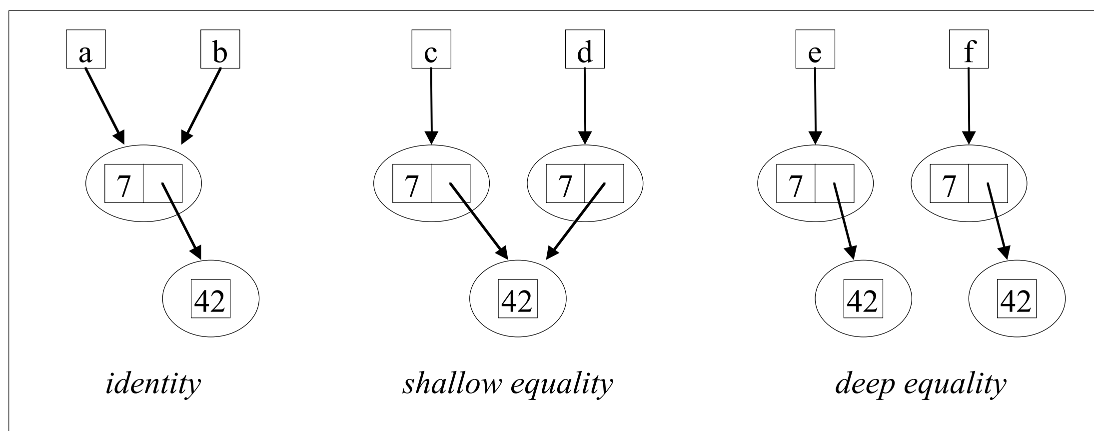

Figure 6.11: Objects and equality

These three types of equality are often used because they are convenient: all three equality checks can easily be generated by a compiler (in case of deep equality, the compiler might need to use tag bits to prevent infinite loops if the structure to be compared has circular references). There is a problem though: none of these might be appropriate in certain circumstances.

The first kind, identity, is useful in many situations, and available in every object-oriented language. It is not enough, though. Especially in languages that support immediate variables (variables that do not store a reference to an object, but the object itself), equality cannot be defined as identity, since every assignment for those objects involves a copy operation. No two immediate objects would ever be equal. But shallow and deep equality also typically do not make much sense, at least not for non-trivial objects. Even with reference semantics another kind of equality is needed. In non-trivial systems equality of objects is often defined as something between deep and shallow equality.

---

[9] In this figure, integers are represented as immediate values, not as objects. This is consistent with the majority of programming languages, but not true for Blue. In the figure, two occurrences of "42" are considered identical. In Blue, both would contain a reference to the 42-object.

To check whether we want to regard two objects as equal in a certain context, we might require some attributes to be compared by identity and others deep, while some attributes may be allowed to be different altogether. In a library system, for instance, two books might well be considered equal if they have the same title, author and ISBN, although their library specific item-number might be different [Grogono 1991]. We call this kind of equality *semantic equality*. A common distinction for this purpose is between *intensional* and *extensional* equality (used, for example, in [Grogono 1991]). Two objects are intensionally equal if they have the same representation and extensionally equal if they have the same abstract value. Intensional equality corresponds to deep equality and extensional equality is equivalent to semantic equality.

The problem with semantic equality is that it cannot be automatically generated by the compiler. A mechanism must be provided for the user to define semantic equality.

Now we can come back to the language: what do we mean when we write *a = b* ?

One of the main considerations is that, at least for simple types like numbers, we want equality after assignment. Consider:

```
a := b
if a = b then
  ...
end if

a := 42
b := 42
if a = b then
  ...
end if
```

In this example, we would hope that *a* is in fact equal to *b* in both *if* statements. Anything else would be highly confusing. This means that the definition of the equal-operator is heavily influenced by the definition of assignment and the object model.

In Smalltalk, even integers can be cloned, so equality cannot be taken as reference equality (since otherwise the same integers might not be regarded as the same). If literals are regarded as object constructors, equality by identity also poses a problem. Consider:

```
a := 42
b := 42
```

Here, *a* and *b* would refer to different objects (two distinct 42-objects which have just been constructed). In many other languages (e.g. C++, Java) integers are always stored as immediate values, thus they cannot be identical.

In all those languages, equality therefore cannot mean identity, but is typically defined to be shallow equality. This is fine for simple types, because they do not have a complex structure. We stated earlier, however, that for complex objects shallow equality is typically useless. Identity and semantic equality are needed here.

The effect is that in all these languages equality can mean different things in different contexts.

In Smalltalk, the operator == defines identity. The operator = has the same value by default, but is – also by default – redefined for all simple types to be semantic equality (which is equivalent to shallow equality in most implementations). Thus, by default, the operator = is semantic equality for simple types and reference equality for everything else (but can be redefined to be semantic equality by the implementor of a class).

Dee, which is interesting to compare because it is one of the few languages which also defines simple types as objects, has a similar discrepancy: the = operator means shallow equality for simple types and semantic equality for user-defined classes (with the responsibility for implementing the = operation lying with the user – no default is provided). This approach, where the = operator calls a routine of the object, has one serious problem: it cannot work if that object is *nil*. Consider:

```
if a = nil then
   ...
end if
```

If the = operator calls a routine on object *a*, then in the case where this condition is true it is also invalid (and results in a runtime fault, such as an exception) since *a* does not refer to an object.

Dee also has a routine called "same", which may be inherited from a predefined "Any" class, to check for identity.

In C++, the == operator represents shallow equality if the operands are immediate types and identity if they are pointer types. It may be redefined by the user to represent semantic equality.

The drawback of these definitions is that the same operator must be made to represent different functionality in different contexts in order to behave sensibly for both simple types and complex objects.

In Blue, this problem has been overcome. Through the introduction of manifest classes and the exclusive use of reference variables, we can define the = operator to always represent identity. This works as expected for simple types since manifest classes cannot be created or cloned at runtime (see section 6.2.4). For complex objects, identity is the desired equality in many cases (since all objects are referenced indirectly and assignment does not duplicate the object). In those cases where semantic equality is different from identity, an "equal" routine may be defined in the class. This routine is in no way special – it is a normal interface routine of a user-defined class. Two objects can then be compared as in the following code example:

```
if a.equal(b) then
   ...
end if
```

The advantage of this definition is consistency: The equal operator (=) has the same semantics in every context, while semantic equality, which is inherently context specific, is invoked through a user-defined name.

Strings and equality

An interesting case to investigate is the handling of the class "String" in this context. Programmers coming from older languages tend to group strings with complex objects, not with simple types (this is especially true for string objects that can grow and shrink dynamically). This grouping is based on the knowledge of how those data types are stored: since a string of this kind cannot be stored on a stack, it is considered a complex object. This grouping is based purely on technical considerations and has no justification in the logical model.

In Blue, strings are, in fact, manifest classes. The reason lies in the way we think about equality.

String literals, such as `"This is a string"` can be interpreted in two different ways: they can be defined as constructors for string objects (Dee uses this approach) or as constant references to string objects (i.e. manifest objects as in Blue).

The reason for viewing them as manifest objects is that we tend to regard two strings made up of the same characters to be the same. Consider:

```
name := "Leah"
if name = "Leah" then
  ...
end if
```

Intuitively, we would assume the condition in the *if* statement to evaluate to true. In our mind, there is only one string `"Leah"`; we typically do not regard two of those strings as different.[10] (This is fundamentally different for other kinds of objects: we all know that, for instance, two people, even though they have the same first name and the same last name, can still be two different people!)

For this reason, the class "String" is a manifest class; all strings logically exist, and string literals are references to those string objects. An effect of this definition is that strings are immutable. The Blue string class defines no procedures that change the string. If, for example, the characters in a string are to be converted to upper case, Blue uses a function, as shown in this example:

```
command := command.toUpper
```

The routine "toUpper" is a function that returns a reference to a different string. Similarly, all other string manipulation routines, which are typically defined as procedures in other languages, are defined as functions in Blue. (Of course, a good

[10] Dee achieves the expected semantics with the trick mentioned earlier: only the first occurrence of the string is considered a constructor. The second one references the same string.

compiler may include optimisations in the code to avoid the copying of strings in many cases.)

6.12.2 Set membership

Blue provides set literals. These literals can be used in boolean expressions to check membership of a value in a set of values. For example:

```
if a in {1, 2, 3} then
  ...
end if
```

Sets can be expressed by enumeration (as shown above), by using ranges of values or a combination of both. Valid sets include:

```
{1, 5, 8}
{3..15}
{"a.."z", "A".."Z"}
{Monday..Friday}    -- using enumeration "Weekday"
```

Set literals are only used in connection with the "in" operator and in case statements (see above). They are included in the language because they can significantly increase readability in cases where one value has to be checked against a set of given values. Their use in case statements makes those a very flexible and useful construct. Allowing their use in general boolean expressions is consistent and convenient.

Set literals are, as their name suggests, only available as literals. No variables can be declared of these types. (Where general set types are needed, the "Set" class from the standard collection library should be used.) They can be regarded as a convenient form of writing combinations of other boolean expressions (=, <, >), provided to increase readability. The expression

```
a in {0, 10..20, 30..40}
```

for example, is equivalent to

```
a = 0 or (a >= 10 and a <= 20) or (a >= 30 and a <= 40)
```

6.12.3 Object creation

How do objects come into existence?

There are several different techniques in use in different languages. They can be explicitly created by the programmer, they can be implicitly created by the runtime system, or they can be implicitly created by certain operations. Some languages also provide a cloning operation.

C++, for instance, uses all of these methods. Explicit creation is done using the *new* operation. Variables that do not contain references (called *automatic* variables in C++) create an object automatically as soon as they come into existence (e.g. local variables of a routine each time the routine is called). And the assignment operator, by

default, may create a new object (depending on the exact type of the value being assigned). C++ has, for this case, a mechanism called the *copy constructor*. This causes a confusing combination of object creation mechanisms that often makes it difficult to keep track of object identity.

Languages that only use references to objects (and do not store objects in variables directly) typically define much simpler object creation rules. There is no need for automatic creation or copy constructors.

The issues connected to literals and the creation of simple types have already been discussed in sections 6.2.3 and 6.2.4. Here we concentrate on the creation of user-defined objects.

Object creation in Blue

The only way objects in Blue are created at runtime is via the *create* instruction, which has to be explicitly issued by the programmer. Objects are never implicitly or automatically created. An example of this construct is:

```
p1 := create Person
```

Creation of an object also involves the execution of the creation routine. There is always exactly one creation routine with the special name *creation*. If this routine has parameters, then the actual parameter list follows the type name in the creation instruction. Consider a “Person” class with the following creation routine interface:

```
creation (name: String, age: Integer)
    == Create person with "name" and "age"

  pre
    name <> nil and age <> nil
```

In this example, the creation routine expects two parameters, “name” and “age”. A creation instruction for an object of this class needs to provide values for those two parameters:

```
p1 := create Person ("John", 21)
```

The specifier after the *create* keyword specifies the *type*, not only the class. The difference between the two is not obvious in our examples so far, but it will become apparent as soon as we take generic classes into account. A list of Persons, for instance, is created like this:

```
pList := create List<Person> (99)
```

(It does not matter what the parameter 99 means in this example - it is a fictitious parameter shown to demonstrate a parameter list in the creation of a generic class.)

More details about the distinction between types and classes, and of generics in general, are given in section 6.14.

The creation routine is defined in a fixed location in the class’s source. It appears as the first definition after the “interface” keyword, before the “routines” keyword. This rule visually separates the creation routine from other routines in an attempt to hint at

the fact that this routine is different from normal interface routines (since it is implicitly invoked by the “create” operation and cannot be called at any other time). An example can be seen in the code example at the beginning of this chapter (Figure 6.1, page 65).

Creation expression vs. creation statement

Several alternatives for the syntax of creation instructions were considered. The first question is: Should the creation instruction be a statement or an expression? Eiffel, for example, chose the statement solution. A creation instruction in Eiffel may look like this:

```
!!p1.make ("John", 21)
```

The double exclamation mark is the creation operator, “p1” is a variable and “make” is the name of a creation routine. The main difference in using a statement for creating an object is that the variable becomes a syntactical part of the creation instruction and the assignment of the new object to the variable a semantic part of the operation. In the Blue example we have achieved the same effect by combining the creation instruction with an assignment to a variable. The assignment, however, is a separate instruction. The creation instruction itself is just the expression

```
create Person ("John", 21)
```

By assigning the result of this expression to a variable, we can achieve the same as the Eiffel expression shown above:

```
p1 := create Person ("John", 21)
```

Using an expression to create the object has advantages both in clarity and flexibility. Clarity is improved because the assignment symbol makes it very clear that the value of the variable on the left hand side is changed. The same symbol that indicates the assignment of a value to a variable is used. In the Eiffel version, the syntax does not clearly suggest that the variable “p1” is changed by this operation.

Flexibility is improved by allowing the creation of an object without the immediate assignment to a variable. The new object could, for example, be passed as a parameter. Consider the following example:

```
myHash := create HashTable (create StringComparator)
```

In this case, a hash table is created that expects a comparator object as a parameter to its creation routine. This comparator object is created as well, and immediately passed to the hash table. With creation instructions as statements, this nesting of creations is not possible. We would need separate instructions and a temporary variable to achieve the same result.

Routine syntax vs. keyword

Another question to consider is whether creation should be invoked by calling a creation routine with (more or less) normal routine call syntax, or whether a new keyword (or symbol) should be used. Blue, as we have shown, uses the keyword

create. Let us consider the alternatives. Since the name “creation” is known to be the name of the creation routine, a call to this routine might be enough to signify creation of an object. This might be written as

```
p1.creation ("John", 21)
```

or

```
p1 := Person.creation ("John", 21)
```

The first example is very similar to the Eiffel construct. (We would not need a special creation operator, since the name of the creation routine is fixed and can be recognised. Eiffel allows several constructors with user defined names, so an operator is needed to indicate creation.) Its great disadvantage is syntactic inconsistency: the operation looks like an object routine call, yet the semantics of the operation are fundamentally different. This violates our aim of syntactic consistency. A result, for example, is that the rule *“An uninitialised variable may not be used for an object call”* does not quite hold any more. Now we would have to add the exception *“unless the routine called is a creation routine”*. Similarly, the rule that states that an object routine call does not change the variable that is used in the call (only the object to which the variable refers) and the rule that states that variable values are only changed by assignment would have to be extended with additional cases. These complications of the language rules are clearly undesirable.

Such problems are avoided in the second version shown above. Here syntactic consistency for assignment and variable access are maintained. Instead, we use the class name rather than a variable name to indicate creation. We effectively introduce operations (the creation routines) that belong to a class rather than to an object. This, on the other hand, violates our rule that the identifier before the dot in an object routine call is a variable. It also immediately poses the question as to whether it should be allowed to call other routines on the class rather than on an object, and thus suggests the introduction of class routines.

We decided that we do not want to include class routines in Blue. They seem to be very rarely needed (if at all) and do not justify the addition of a new construct. They provide nothing that could not be achieved with existing constructs.

This leaves us with a situation where this syntax for object creation would be a special case, standing alone against general syntactic rules. Special cases like this should be avoided.

The disadvantage of the solution chosen for Blue is that it requires the introduction of a new keyword. Some language designers are very reluctant to use keywords and make it one of their priorities to keep their number as low as possible. (This is very obvious in C and C++. Even in Eiffel, Meyer has chosen a symbol here instead of a keyword – the double exclamation mark. This seems inconsistent with the rest of the design of Eiffel and appears to be a strange exception in an otherwise well designed language.)

While we agree with the aim of avoiding unnecessary keywords, it clearly ranks lower than the aims of clarity and consistency. In this case the use of a keyword so clearly improves readability and consistency that it seems well worth the trade-off.

Cloning

Another mechanism that is provided by some languages to create new objects is *cloning*. Cloning creates the object by copying an existing object. There are two reasons that led us to consider cloning an unnecessary feature for a first-year language.

The first reason is that it is not often necessary to clone user-defined objects at all [Grogono 1994b]. Providing a built-in clone function seems like overkill for those few cases where it is actually needed. The number of uses does not justify a separate language construct.

The second reason is that in most cases, cloning cannot be correctly automated at all. The problem is very closely related to that of deep and shallow equality as discussed in section 6.12.1. As with equality, it is not clear when cloning an object, how shallow or how deep the object should be cloned. Eiffel, for instance, defines two different clone functions, one for deep cloning and one for shallow cloning. For most classes in real systems, however, cloning their objects with either one of them would be incorrect. Only a small subset of objects is correctly cloned by copying all data shallow or all data deep. Typically, a non-trivial object is correctly cloned by a combination of deep and shallow cloning of its fields. For correctness assurance of systems it is better to force the user to define the clone operation explicitly to ensure that the clone operation has been adapted to the particular class. Enabling incorrect cloning when the class designer has not provided a clone function is error-prone.

For these reasons Blue does not provide a built-in clone operation. Each class designer might decide to provide a clone operation when needed, but this operation is in no way special. It is a normal interface routine that internally uses the built-in *create* operation to create the new object.

6.13 Inheritance

Inheritance is one of the most fundamental, yet least understood constructs in object-oriented programming languages. Most authors agree that inheritance is one of the ingredients that make a language "object-oriented". A language without inheritance is, in most classifications in today's literature, not considered to be object-oriented.

The importance people place on inheritance is well deserved. This construct allows application designs which are significantly different, and usually superior, to those possible with previous imperative languages. Designs making intelligent use of inheritance often have advantages in clarity, flexibility and maintainability. Inheritance is also invaluable in supporting code reuse through libraries and frameworks. On the other hand, inheritance is often misused.

Discussing misuse of inheritance is a difficult topic, since the programming language community currently does not agree on what constitutes good use of inheritance, and what is misuse. Different authors argue quite strongly for different uses of inheritance, and no viewpoint has proven to be the clear winner in this discussion as yet. We will, nonetheless, point out some of the issues connected to inheritance and state our view on which the inheritance construct in Blue is based.

6.13.1 Inheritance - the Swiss army knife

Inheritance is, in some sense, the Swiss army knife of programming language constructs: a flexible tool useable for a variety of completely unrelated tasks. Evered et al. [Evered 1991] for instance, have identified sixteen distinct uses of inheritance. Several authors in recent programming language literature have argued that separate constructs should be developed to support some of the uses distinctly that are currently handled through inheritance [America 1990, Evered 1991, LaLonde 1991]. The inheritance construct, therefore, has been likened to the "goto" statement in earlier languages: It is very powerful and very flexible. Through its flexibility, however, it is possible to use it in ways that have negative effects on the system design. Thus, as languages evolve, the construct is replaced with several more restrictive constructs which reflect different "good" uses that the original one allowed. For the "goto" statement, these are loops, procedures, return statements, etc. Equally, the inheritance construct might be replaced with alternative constructs in future languages.[11]

The most important distinction in the different uses of inheritance seems to be the one between its use for subtyping and its use for code reuse. We discuss these two in some more detail.

6.13.2 Inheritance for subtyping

Inheritance is often said to establish an "is-a" relationship. Consider the following example:

```
class Person is
  ...
end class

class Student is Person
  ...
end class
```

This example declares two classes: the *base class* `Person` and a *derived class* `Student`. `Student` inherits from `Person`. A pair of classes related by inheritance is often also called *superclass* and *subclass*, or *parent* and *child class*.

---

[11] These languages are not really in the future. Several experimental and research languages already exist. Theta [Liskov 1995] and POOL-I [America 1990], for example, address this problem. These languages are, however, not widely used, and probably never will be. For mainstream languages, this development clearly still is in the future.

Here we have a clear case of an "is-a" relationship: we can say that a student *is a* person. In the programming context this means that the class `Student` inherits the attributes (variables and routines) from the class `Person`, and that a subtype relationship is established[12]. This subtyping enables polymorphism: an object of type `Student` may be used where an object of type `Person` was expected. This is logically correct, because we know that every student is a person. It can be guaranteed to be technically correct, because a student object has all the attributes expected from a person object. Note, however, that for this to work, the subclass only needs to inherit the *interface* of the superclass's attributes, not its implementation.

This subtyping and the associated polymorphism is one of the most important uses of inheritance.

6.13.3 Inheritance for code reuse

Inheritance may be used to reuse existing code in a new context. An example is a stack which inherits an array that forms the core of its implementation:

```
class Array<T> is
  ...
end class

class Stack<T> is Array<T>
  ...
end class
```

In this case, the stack inherits all the attributes of the array and can thus provide space to store data. It is not an "is-a" relationship, though. A stack is not an array. We can, for instance, not access element number 5 in the stack. This becomes even clearer in another example, taken from [Meyer 1992, p 279]:

```
class POINT                      -- Eiffel syntax
   inherit TRIGONOMETRY
...
end
```

Here the class `POINT` inherits the class `TRIGONOMETRY`. This is not meant to say that "Point is a Trigonometry" (thus creating a subtype), but is done purely to be able to call `TRIGONOMETRY`'s routines from within `POINT`. A further possibility with this use of inheritance is to inherit the same class twice to acquire two copies of a data attribute of the superclass. (This is actually used in [Meyer 1992, p 173].) Inheriting the same class twice has no meaning in the sense of inheritance as a subtyping relationship.

A side effect of this use of inheritance is that we have to modify the inherited interface. A stack might not want to allow the random access that its superclass, the array, provides. It might want to remove the corresponding routine from its interface.

---

[12] In the literature that discusses the different semantics of inheritance mechanisms it is also pointed out that an "is-a" relationship is not the same as subtyping. It is not our aim here to go into the details of this discussion; interested readers are referred to [LaLonde 1991].

Another stack, this time implemented by inheriting from a queue, might want to rename the inherited "dequeue" operation to "pop" and the "enqueue" operation to "push". In this case the two classes do not form a subtype relationship. Some mainstream languages provide mechanisms to do this (e.g. private inheritance in C++ and Smalltalk).

6.13.4 Problems with inheritance

The problem with inheritance in most mainstream languages is that inheritance for subtyping and inheritance for code reuse cannot be separated. In those languages, the examples given above (e.g. `Stack` inheriting from `Array`) automatically form a subtype relationship. This leads to a wide variety of problems that might result, depending on the details of the language, in runtime errors or protection violations (such as breaking of encapsulation). In effect, if a class wants to form a subtype relationship, it automatically also inherits the code. If it wants to inherit the code, it also becomes a subtype.

For Blue, we did not attempt to introduce a new set of constructs to properly address all of these problems. This area is clearly an open research issue at this stage. Different proposals to address these questions have been made, but none has yet become really successful in practice. We do not see it as our task to develop new language constructs, but to identify and simplify mechanisms proven successful, and to provide those for teaching.

Instead, we tried to define an inheritance mechanism that provides for those uses of inheritance that are more or less uncontroversial, and provides for them in a clear and simple way. We tried to discourage the use of inheritance where it is done purely for efficiency reasons and thereby distorts the underlying model.

For us, this effectively means that we defined an inheritance mechanism that supports "is-a" relationships and subtyping, but discourages inheritance for pure code reuse.

Typically, when code reuse is the aim of the use of inheritance, the same effect can be achieved by establishing a "uses" (client–server) relationship. The class `Stack`, for example, inherits from `Array` to make use of some of its functionality. It could also achieve this by declaring an internal instance variable of type Array. This variation is slightly less efficient, and it requires the programmer to qualify routine names with the object name in routine calls. It might also mean that inherited functions, which should appear in the interface of the subclass, need an equivalent function to be defined to pass the call on to the inherited function. On the other hand, it ensures that the interface of the subclass is clean – it contains no routines that should not be there – and that renaming or hiding of interface routines is not necessary.

6.13.5 Inheritance in Blue

Blue defines a simple, straightforward inheritance mechanism. Only single inheritance is supported, and the inherited interface cannot be modified. In particular, inherited routines cannot be renamed or hidden. This discourages the use of

inheritance where no real “is-a” relationship (specialisation) is intended. Routine interfaces cannot be changed. Parameter types, which can be modified in many other languages, must remain the same in Blue. Covariance (as, for instance, in Eiffel) is not supported because of the associated typing problems. Contravariance (e.g. in Sather) was not seen as generally useful. No variance (insisting on unchanged parameter types) is sufficient in almost all situations and avoids a wide variety of typing problems that would have been hard for first year students to understand.

One possible criticism of this decision is that the students do not get trained in other, perfectly valid uses of inheritance. Inheriting for code reuse without inheriting the parent’s interface is a powerful technique that cannot be achieved in Blue.

Our answer to this argument is that this is again, as with some other programming techniques, a question of maturity. We do not argue that those techniques are bad or should not be used at all. We do argue, though, that their application in a first course clouds the meaning of the inheritance relationship and makes it harder to understand the issues connected to the application of this flexible mechanism. We argue for a stepwise introduction of inheritance techniques: As a first step, inheritance is introduced as a clear specialisation (“is-a”) relationship. Later (in a second programming course, using a language other than Blue) other, more advanced uses of this construct can be introduced. We strongly believe that by restricting the inheritance construct and the number of cases where it is sensibly applied, we clarify the issue of inheritance and its related problems.

Syntactically, inheritance is specified by naming the parent class in the class header. For example:

```
class Car is Vehicle
  == A class to represent a car in a traffic simulation
  ...
end class
```

Here the class `Car` inherits from `Vehicle`. The effect of this relationship is that all instance variables and all routines (including the internal routines) are inherited by the subclass. The only exception is the creation routine – it is never inherited.

6.13.6 Inheritance and creation

Inheriting the creation routine interface is almost never useful. Each class initialises its instance data in its creation routine, and only in very special cases (in the absence of any instance data to be initialised) does a class not need its own creation routine. This means that the inherited creation routine must, at the very least, be redefined to initialise the new instance data as well as the inherited data.

But this is usually not enough. Redefinition in Blue, as mentioned above, is not allowed to change the interface of the routine. The initialisation of the instance data, however, is often done through the supply of values via parameters and necessitates a change in the parameter list of the creation routine (usually the addition of parameters). Thus, the creation routine is fundamentally different from other routines. It is also different in another respect: It can never be polymorphic. When creating an

object, it cannot happen that a user does not know the exact object type and that the object is accessed through a variable of one of its supertypes. Thus, changing the parameter list for the creation routine is not a problem of type safety or runtime safety. The parameter list can safely be changed here without compromising the static type safety of the system.

In this sense, the creation routine clearly is not a normal routine and requires special treatment.

The effect of the creation routine interface not being inherited is that a subclass can define its own creation routine with its own parameters. It is, however, often useful to call the parents creation routine from within the child's creation routine. In fact, this is almost always a good idea in well written classes. Some languages try to automate this (e.g. C++). There is a problem with the automatic execution of the parent's creation routine, though, namely the parameters. The system cannot automatically know which actual parameters to pass to the parent's creation routine. C++ tries to overcome this problem by demanding either a special creation routine without parameters, or a manually inserted call. Blue always requires a manually inserted call, since we do not believe that a parameterless creation routine leads to good programming practice. The effect would be that instance variables remain uninitialised, or that a separate initialisation routine must be used, thus separating creation from initialisation and increasing the chance of incorrect programs.

Blue provides a mechanism for calling the original of redefined routines from the parent class. For example:

```
class Car is Vehicle
  == A class to represent a car in a traffic simulation
  ...
interface

  creation (loc: Location, numberOfSeats: Integer) is
      == Create a car with ...
    do
      super!creation (loc)
      numSeats := numberOfSeats
    end creation

  ...
end class
```

The creation routine of `Car` receives a location and a number of seats as parameters. We assume that the parent class, `Vehicle`, has a creation routine that expects the location as a parameter. This routine is called with the "super" keyword. "super" has the effect that the implementation of the corresponding routine from the parent class is executed. This can be done in routines other than the creation routine if a routine implementation was redefined. In the next line, we assume that an instance variable `numSeats` exists in this class, which is initialised here.

The super keyword uses an exclamation mark, rather than the dot notation, to make the routine call. This is deliberately chosen to indicate the different semantics of the two constructs. Let us first consider the alternative, using dot notation here. The superclass call shown above would become

```
super.creation (loc)
```

Java, for instance, uses this syntax for this construct. The problem with this is a violation of the consistency requirement in our aim of readability. In that requirement we stated that the same syntax should be used for semantically similar constructs and different syntax for different constructs.

The dot notation is defined as expressing a call of an interface routine of another object, which is named before the dot. None of this is true here. In the case of the superclass call, we do not necessarily call an interface routine, it is not another object (but the current object) and the part before the dot does not name an object but specifies a scope. Thus, the superclass call mechanism is fundamentally different from object calls expressed through dot notation. This difference should be reflected syntactically.

6.13.7 Access protection

In Blue, subclasses have full access to internal routines and variables of the superclass. Some other languages allow the same access (e.g. Eiffel), some do not (e.g. Smalltalk). In C++ and Java, access can be regulated by the superclass implementor. The superclass can specify three access levels: *private*, which only the class itself can access, *protected*, accessible by the class and its subclasses, and *public*, which is accessible by all classes.

With Blue, we follow the Simula/Eiffel model which defines only two levels of access protection (named *internal* and *interface* in Blue) and allows subclasses access to the internal data. This reflects our view of the inheritance relationship as a much closer one than a client/server relation between classes. Elegant implementations would often be hampered by forbidding subclass access to internal features. Many patterns, for example most models of a graphical user interface (GUI) library, require internal routines of library classes to be redefined by user-defined subclasses. In the GUI example, these are action routines which are called through an internal mechanism as a result of the activation of the user interface component (such as a button click), and which can be redefined to perform an action associated with the component. The alternative, to move those routines into the interface so as to allow them to be redefined, represents bad interface design and should be discouraged.

The flexibility of the C++ approach of providing three levels of access is sometimes helpful to express constraints on a class. In general, however, we consider its value as not very high at the beginners' level. Its application is mostly in advanced class patterns, and we do not think that an entry level language increases its clarity by providing a third access level. We have, therefore, decided in this case on the side of simplicity of the language and restricted Blue to two levels of access.

6.14 Genericity

Blue provides a straightforward mechanism for generic classes (also referred to as *parametric polymorphism*). Genericity has been treated poorly in some recent programming languages. C++ did not include it in early version and added it (under the name *templates*) only in later language revisions. Its syntax is poor and the implementation in current compilers still often unsatisfactory. Java does not include genericity at all.

The reason for not including generic classes in the language seems to be mostly implementation problems. Especially with Java, which was developed by a commercial company, the "time-to-market" figure seems to have taken precedence over good language development.[13]

Genericity is an important and powerful concept in object-oriented programming. Maybe its most important role is in the ability to provide a good library of collection classes. Collections classes (lists, stacks, sets, hash tables, etc.) are immensely important for the teaching and development of good software and are an ideal example for teaching the idea of reuse. They can be written in a much easier and safer way with genericity than without.

Genericity, though sometimes regarded as an "advanced" concept, is very easy to understand and to use. It has two facets: the use of an existing generic class and the development of a new one. We are convinced that learning of good object-oriented software development benefits greatly from the ability to make use of generic collection classes. Lists, for example, should be used early in a first course in many cases where they are the appropriate data structure, rather than falling back on using an array, because the use of lists seems not teachable. Rather than teaching the use of an inappropriate data structure only because it seems easy (the array), we should make the use of the appropriate data structure simple. Genericity can do just that.

In practice, there is no need to teach the two sides of genericity together. We favour an approach where the use of existing generic classes is introduced very early. It is straightforward and easy to understand. The development of new generic classes can be left to much later in the course.

6.14.1 Unconstrained genericity

Unconstrained or general genericity is the standard case: a class can take any type as its parameter. Consider an example of a generic class `List`:

---

[13] Sun, the developer of Java, has now called for proposals for adding genericity to the language.

```
class List <ELEM_TYPE> is
  == Generic list class
  ...
interface
  ...
  append (element: ELEM_TYPE) is
    == append 'element' at end of list
    pre
      element <> nil
  ...
end class
```

The generic class definition lists a name in its header (here `ELEM_TYPE`), which defines a type. This type is instantiated when the generic class is used and may be used as a type name within the generic class. We can, within `List`, declare variables, parameters or return values of this type, as shown with the parameter of the `append` routine.

When another class wants to use `List`, it has to provide a type parameter wherever the List type is used. For example:

```
class MyClass is

  == Example using List class

  uses List

internal
  var
    myList : List<Integer>

interface

  creation is
      == Create an object of this example class
    do
      myList := create List<Integer>
    end creation

  ...
end class
```

In this example, we see the two locations where the type name is used: at the declaration of the variable `myList` and in the creation of a new object. In both cases, the only difference between a generic and a non-generic class is the appended type parameter in angle brackets. All access to routines of list objects is then fully type checked.

6.14.2 Operations in generic classes

In the generic class itself, the real type of the generic parameter is not known at compile time. (In Blue, generic classes do not need to be recompiled for every instantiation. Only one copy of the code exists for all instances of the generic class.) This means that operations on the generic type cannot always be type-checked. There are two possible solutions to this problem: the introduction of dynamic type checking for these instances, or avoiding the operations. Blue uses the second alternative. Static

type checking is such a valuable tool, especially in a teaching language, that it takes precedence over most other constructs.

As a result, only operations that are known to be valid for all types can be performed on the generic parameter. These operations are assignment and equality check. All other operations are illegal, since they cannot be known to be applicable. Assignment and equality check are enough to implement most general container classes.

6.14.3 Constrained genericity

A simple extension to genericity exists for those cases where additional operations on the generic parameter are needed: constrained genericity. For example:

```
class SortedList <ELEM_TYPE is Comparable> is
  == Sorted list of elements
  ...
interface
  ...
  insert (element: ELEM_TYPE) is
    == insert 'element' into the list at its appropriate
    == position
    pre
      element <> nil
  ...
end class
```

The difference in this example is the introduction of "`is Comparable`" after the generic parameter. The parameter is said to *be constrained by* the class Comparable. We assume here that `Comparable` is an existing class that provides, for instance, a `greaterThan` function.

The effect of this constraint is that `SortedList` can only be instantiated with `Comparable` or one of its subtypes. In return, we can now use the operations of class `Comparable` on the generic parameter. The insert routine, for example, can use `greaterThan` to find the position in the list to insert the new element.

It must be admitted that the use of constrained genericity is limited for general purpose classes through the restriction in Blue to single inheritance. A class that already has a superclass cannot be made directly to inherit from `Comparable` to be insertable in a sorted list. This problem can, however, be overcome in many cases. Often the superclass itself can be made to inherit from `Comparable`, thus passing the operations on to its subclass. Also, some cases of problem specific classes (as opposed to general purpose container classes) can make good use of this mechanism.

6.14.4 Genericity and conformance

Conformance with generic classes is a logical extension of conformance for simple classes. Conformance is established by a subtype relationship, which in turn is defined by the principle of substitutability. Consider:

```
class IntegerList is List<Integer>
  ...
end class
```

In this example, IntegerList is a real subtype of List<Integer>, and thus the types conform. But:

```
class Person is
  ...
end class

class Student is Person
  ...
end class

var
  personList : List<Person>
  studentList : List<Student>
```

Here, studentList cannot be assigned to personList, since List<Student> is not a subtype of List<Person>. (On first sight it might appear that they are substitutable, but this is misleading. If we were allowed to assign a List<Student> to a List<Person> variable, then a Person object could be entered into the Student list – which is an error since all objects in that list are expected to be at least of type Student.)

Two uses of a generic class with the same type parameters denote the same type, and are therefore assignment compatible:

```
var
  list1 : List<Integer>
  list2 : List<Integer>

...

list1 := list2      -- legal, both are of same type
```

Blue uses type equivalence by type identity (two structurally identical types are still two distinct and incompatible types). The definitions above refer to the same type. Blue does not create two distinct List<Integer> types.

6.15 Concepts not included in Blue

The previous sections in this chapter were mostly concerned with discussing language elements of the Blue language. We have tried to also point out alternative routes that could have been taken, to compare our solutions to those chosen in other languages and to discuss the advantages and disadvantages of those alternatives. We have tried to justify why particular decisions were made.

Since this discussion was structured along the lines of the Blue language definition, it did not include discussion of concepts that were excluded from the language altogether. This last section in this chapter serves to partly correct this shortcoming. We discuss some common constructs that could have been included but were not, and argue our case in the attempt to justify their exclusion.

6.15.1 Multiple constructors

Most object-oriented languages provide the ability to define multiple constructors. These constructors might have a common name and be distinguishable by their parameter list, or they might even have different names and be identified as a constructor by some other means (e.g. a keyword associated with the constructor function or a specific constructor section in the source definition).

We decided against providing the ability to define multiple constructors. The main reason is that the constructor, while looking very similar to an interface function, is a very special construct. It is important for the understanding of very fundamental concepts to understand this difference. Moreover, this understanding has to be gained by the student quite early in the course. It is therefore important to have as clear a distinction as possible between the constructor and other interface routines. Allowing multiple constructors increases the danger of confusing these different constructs.

There are examples where multiple constructors clearly are desirable to model a given problem, and most descriptions of languages providing multiple constructors give one of these examples when introducing the topic. The number of these examples, however, does not seem large enough to risk misunderstandings about fundamental concepts early in the course.

Providing user-defined names for constructors (as, for instance, in Eiffel) makes them look very similar to normal interface routines. Providing multiple constructors with the same name would introduce function overloading – a construct against which we argue in the next section.

6.15.2 Function overloading

In Blue, function names cannot be overloaded within a class.[14] This is true for both user-defined functions (a class cannot offer two functions with the same name) and for the overloading of standard functions (such as > or >=, see section 6.15.3).

The case for name overloading of user-defined functions is often argued by giving examples of semantically related functions with different parameter types. A class defining objects to which values can be added, for instance might offer two functions: one to add integer values and another one to add floating point values. The semantics of these functions would be the same (with only a small technical difference in their parameter types), so it could be justified to give them the same name, say “add”:

[14] We are referring here explicitly only to overloading *within one class*. Overloading between classes is, of course, possible: Different classes may have routines with the same names. This is essential to allow independent development and, in fact, a pre-requisite for one of object-orientation’s strongest mechanisms: polymorphism.

```
class FunnyNumber is
   ...
interface

   add (value: Integer) is
      == add an integer value to this number

   add (value: Real) is
      == add a real value to this number

end class
```

While this example seems sensible, it is not really superior to an alternative without overloading. In such an alternative we would name the functions “addInt” and “addReal”. The gain from function overloading is minimal – in fact, many people would argue that the second version is clearer. On the other hand, significant potential for misuse is avoided, where overloading might be unwisely used for functions that differ in their semantics more than here, and where similar semantics are falsely suggested.

From the point of view of supplying a useful development environment, name overloading also introduces additional problems. Enforcing different names for different functions makes searching for function definitions or function calls much easier for users by avoiding the location of name duplicates representing different functions.

6.15.3 User defined infix operators

Infix operators (such as +, <, >=) cannot be defined for user-written classes. Again, their potential for misuse and confusion is greater than the advantage which their application brings.

As in the case of function overloading, there are some examples where the use of infix operators (overloaded or not) for user-defined classes is reasonable. The commonly cited example is the definition of a class “Complex” for the representation of complex numbers. Since numbers can be added, the use of the standard arithmetic operators might be desirable here. This would allow the programmer to write

```
c1 + c2
```

for complex number objects instead of

```
c1.add (c2)
```

which is the standard syntax used without infix operators. The problem with these operators is that their good uses are few, whereas their misuses are many. Students might easily be tempted to use infix operators for the wrong reasons. Since the notation is short and concise, it might be used where the functionality of the operation does not really justify the automatic semantic association. But even if the use seems justified, it might not be optimal. Consider a list class that has an operation to add an element. A programmer might decide to use the + operator to name this operation. One can then write:

```
myList + newElement
```

Although the association of the addition symbol makes some sense, in that the element is added to the list, there are better ways to express this operation with named operations. For example:

```
myList.append (element)
myList.insertBeforeCurrent (element)
```

These examples are clearly more specific. Their names convey information not only about the fact that the new element is added, but also about where in the list it is added.

Most of the time (in fact, in all cases where the class represents an entity that is not directly taken from commonly used mathematics) the + symbol is not an optimal routine name. While we are trying to teach students to write readable programs, part of which is careful naming of operations, it would be counterproductive to offer a mechanism that works against this goal.

Another problem with user defined infix operators are the associations automatically connected to them, which are not guaranteed to be valid any more. If, for example, we allow the redefinition of the greater-than operator (>), we could write

```
a > b
```

for our own objects. In many cases this might be sensible. There is, however, no guarantee that the operator <= is the inverse operation (that is, that *not (a > b) implies (a <= b)*). There is, in fact, no guarantee that the inverse operation is defined at all. Yet, a user of the class might easily assume this relationship.[15]

In light of these arguments it seems that the absence of user-defined infix operators might well contribute more to clarity of programs than their presence.

6.15.4 Explicit blocks

Some languages, especially structured procedural languages like Pascal or C, provide explicit, user-defined blocks which may be used in addition to blocks implicitly created by procedures and functions. These blocks serve mostly to create a new name space in which new variables may reuse names of variables in outer blocks. The use of explicit blocks is especially beneficial in very long sequences of sequential code.

In well written object-oriented programs the routines tend to be much shorter than in programs written in a procedural style. Experience indicates that explicit blocks are never really needed to structure the code. The two-level structure provided by classes and routines provides a sufficient organisation of the name space.[16]

---

[15] Ada has, in fact, made an attempt to overcome this problem. When a < operator is defined in Ada, the >= operator is automatically defined as its negation.

[16] In large systems, a third level is very useful: a "package" or "cluster" that can group several classes into one unit. This, however, is another structure larger than a class, not smaller than a routine.

6.15.5 Routine parameters

Passing routines as parameters is a powerful mechanism in programming languages. However, it does not fit well with the object-oriented paradigm. In our model of object-orientation, all code that exists is part of a particular class. When executed, it is executed as an operation of an object of that class. Passing pieces of code around to other routines or other objects cannot be easily fitted into this model. We would necessarily have to compromise the underlying concepts.

As a side remark for those people insisting on the necessity of routine parameters we can add that they can be simulated using classes and inheritance. We have to define a superclass with the routine signature to be passed and subclasses with the routine in question being redefined in every subclass. This, admittedly clumsy, technique is consistent with the object-oriented model and can be used for those cases where routine parameters really are deemed essential.

6.15.6 Immediate objects

In Blue, as in several other languages, all variables contain references. *Immediate* or *expanded* variables (variables which hold the object directly, rather than a reference to it) do not exist.

The reason is the great simplification in the language through the omission of this construct. Immediate variables only serve efficiency; everything that can be done with them can also be done with reference variables. This cannot be said the other way: reference variables are essential, since some structures (e.g. recursive structures such as lists) cannot be implemented without references.

The resulting complexity with immediate variables does not only lie in the necessity for users to keep track of and distinguish these two storage modes. It also greatly complicates other parts of the language design. Two of the constructs affected are assignment and equality, which we have discussed in some detail above (relevant parts of this discussion are in sections 6.2.3, 6.2.4, 6.12.1 and 6.12.3). Problems arising have to do with the assignment and comparison of immediate variables (issues of deep or shallow operations) and with operations combining immediate and reference variables (e.g. assignment from an immediate to a reference variable and vice versa). The effect of this becomes clear by examining the Eiffel description [Meyer 1992]. Eiffel supports reference and immediate variables. In the specification of the language, tables are given listing all possible combinations of reference and immediate variables with separate definitions given for the meaning of the = and := operators for every case.

An interesting trend can be seen concerning this issue in programming languages. Pure reference semantics were first used by research languages that had difficulties being accepted in industry (e.g. Smalltalk). In every industry strength language it was thought necessary to provide immediate variables for efficiency (e.g. C, C++). Eiffel tried to introduce a language aimed at industry with only reference types, but quickly relented. The first version (published in 1988) had a simple and clean design

supporting only references. The second version (1992) introduced immediate types to counter industry criticism of inefficiency. With the Java boom in 1996 it seems that for the first time industry is prepared to accept a language supporting only references. We might have reached a stage where people have enough processing power in their machines that, for many tasks, they can afford to spend some of it on supporting easier, faster and more correct programming.

6.15.7 Multiple inheritance

Multiple inheritance is not supported in Blue. The question as to whether it should was discussed for a long time during the language design phase, and some early users felt very passionately about it, either for or against.

First of all, it has to be emphasised that multiple inheritance is not necessary for object-oriented programming. Every problem that can be solved with multiple inheritance can also be solved with single inheritance or, in fact, without inheritance (although the multiple inheritance solution is more elegant for some problems). Smalltalk, for instance, is an example of a language supporting only single inheritance, which has been used very successfully for implementing a wide range of real life designs.

The decision against multiple inheritance is not based on an assumption that it is not useful. In some cases it is. Some problems are solved more elegantly through multiple inheritance. The decision was made to avoid complicating the language.

Including multiple inheritance makes it necessary to deal with a range of connected issues such as naming conflicts and repeated inheritance. Additional language constructs would have to be provided to support these issues. In addition, multiple inheritance is most often useful when the inheritance relationship is not used for specialisation, but for code reuse. If we were to accept this, we would have to radically change the inheritance semantics described earlier to allow at least hiding of inherited routines. Overall, the resulting complication of the language is not worth the benefit in a first year teaching context. The problems that would significantly benefit from multiple inheritance are too infrequent at that stage to justify the increased complexity.

Java has since presented a fairly elegant approach covering the middle ground. It supports a concept of *interfaces*, a class definition without an implementation and without data attributes. It then allows multiple inheritance of these interfaces, but not of other classes. This approach allows many of the benefits of multiple inheritance while avoiding many of the problems. It was not published at the time we started to design Blue, and a similar approach was not considered. In retrospect, interfaces might offer a solution for the presentation of multiple inheritance in a teaching language.

6.15.8 Iterators

Iterators are used to provide an easy and flexible way to traverse collections of objects. Most of all, they provide a mechanism that allows multiple concurrent or nested iterations to be performed safely and correctly.

Blue does not allow iterators to be specified in any satisfactory way. This is not so much a conscious decision against the use of iterators, but an unfortunate effect of the current definition of the language. Iterators are, in fact, a very useful and elegant construct. We have not yet been able to decide on an easy and elegant way to allow the inclusion of iterators in Blue.

In other languages, two different approaches to iterators exist: user-defined and language-defined iterators.

In the user-defined approach, the system designer defines an iterator class together with the collection class. A user of the collection can then create any number of independent iterator objects. The advantage of this approach is that it needs little explicit language support and thus does not complicate the language definition. The reason it cannot be applied in Blue is that the iterator class, to be implemented in any sensible way, needs access to the internals of the collection class. In Blue, a class cannot give another class access to its internals. The language support needed therefore is selective export (the ability to make some routines accessible by only certain classes - Eiffel uses this technique) or a “friend” mechanism like in C++ (allowing certain classes or routines unrestricted access to class internals). The alternative possible in Blue, moving the necessary internals into the publicly accessible interface, represents such bad design that it should not be considered.

In the second approach, iterators are supported by explicit language constructs. The advantage of this approach is the simplicity and elegance that can be achieved for the user of an iterator, and possibly readability of the code. CLU and Sather both define iterators as part of the language.

We did not define either construct, selective export or explicit iterators, for Blue. The reason was to keep the language small and to concentrate on the use of proven, often-used constructs. (We must admit, though, that we made exceptions to this rule in other parts of the language definition, e.g. the use of multiple return values of routines.) In hindsight, this is one area that might benefit from review in a possible future version of the language. Deciding on and including a way to provide iterators would probably be a valuable addition to the tools for teaching and learning a good programming style.

In the meantime, the Blue collection library defines collections with a traversal state. The collection object itself stores the information about the current element during traversal. The disadvantage of not allowing nested traversals does not, fortunately, often come into effect in first-year programs. But if it does it might produce an error that is hard to debug for beginning students.

6.16 Summary

In this chapter, we have presented the constructs of the Blue language. We have compared them to other languages and alternatives and discussed the issues that influenced the decision for and against particular constructs. We have seen that language design is a complex task with numerous interdependencies of constructs and issues that make it impossible to discuss constructs in isolation of each other.

Now that the reader has seen our demands for a teaching language (in chapter 2) and our solution in this chapter, we hope that our initial claims (that a language can be made more appropriate for first year teaching) have been vindicated. Our earlier claim that all other mainstream object-oriented languages include language constructs detrimental to introductory teaching has now been substantiated through comparisons with examples from those languages and we have shown simpler and more consistent solutions in Blue.

7 The Blue Environment

7.1 Introduction

The main aim of the Blue environment can be summarised in the following two goals:

- to provide an environment which encourages the students to think in terms of objects and classes, and
- to provide an environment that is so easy to use that it does not distract from the task of learning to design and implement a program.

To achieve these goals, details of the underlying operating system are hidden and a point and click world, in which classes and objects are the fundamental concepts of abstraction, is presented. We assume that bit-mapped displays will be used so that we can make extensive use of graphics.

Figure 7.1 illustrates the desktop presented to students when they log on to the system. Blue has a notion of projects. A project is a group of classes which relate to a particular application. On entry to the system the student chooses the project on which they wish to work. This main window is often referred to as the *project window*. Apart from the pull-down menu bar at the top of the window, it has three components: a toolbar, a class structure overview and the *object bench*. The toolbar at the left is a set of push-buttons which activate frequently used operations. As each class is created it is represented in the class structure overview by a box, with the name of the class at the top. The “classes” may be moved around the screen and inheritance and client relationships can be created using the arrow buttons. These relationships are represented graphically using lines and arrows. Different colours, patterns and

symbols are used to mark different kinds and states of classes. This includes information as to whether the class has been compiled and its category, e.g. abstract class, library class, etc. The object bench is discussed in section 7.6.

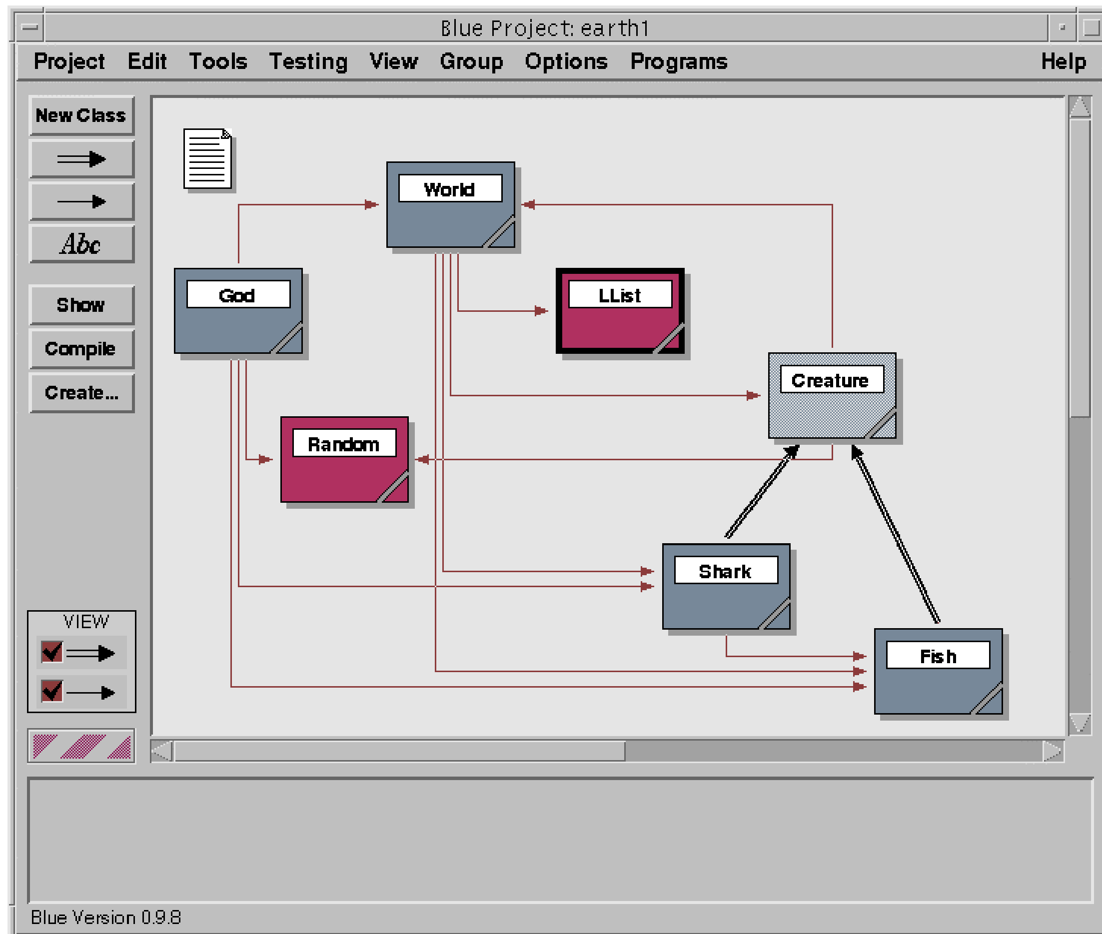

Figure 7.1: The Blue main window

Students are encouraged to begin their design of an application by creating the required classes. They should think about the relationships between these classes and represent them graphically. Only when the overall class structure has been determined should they start to think about the interfaces and the code.

The text associated with a class may be viewed and edited by double clicking on the class (or selecting a class and pressing the "Show" button). The editor is language sensitive and provides specific functionality for convenient insertion of Blue code. Either the interface of the class or its implementation may be viewed. There is only one representation of the text internally, so it is impossible for these to get out of step with each other.

Compilation of classes or the whole project is achieved by a single click on a button in the project window. The environment automatically keeps track of changes to classes and dependencies. Only classes that need to be recompiled will be recompiled. This removes the need for students to become familiar with systems such as "make", allowing them to concentrate on the basic programming concepts and techniques.

A central feature of Blue is the ability to create instances of classes dynamically. When the “Create” button is pressed, a dialogue is shown which prompts for the constructor parameters and an instance of the selected class is created. This instance is displayed in the bottom section of the main window, known as the *object bench.* An interface routine of an instance may be called by selecting it from an associated pop-up menu. Again, Blue prompts for the parameters. Object instances may be composed and passed as parameters to each other. The results of an object invocation are displayed and if a result is an object it can be placed on the object bench. This allows interactive testing of components as they are developed.

A standard console object exists that displays a standard output window and acts as standard input. Alternatively, an application can open its own windows.

The environment also provides a class browser and a debugger. All of these components and the issues associated with their design are discussed in more detail in the remaining sections of this chapter.

7.2 Keeping it easy – the user interface

The Blue system uses a graphical user interface with the usual interface components such as buttons, menus, etc. It provides a graphical, editable display of the class structure of the project currently under development.

It seems that we, in computer science education, have for a long time not made the best use of our own technology. We are training our students in the skill of developing “user friendly” applications. We teach about user interfaces and the use of our technology to ease the tasks at hand. Yet in our own teaching we tend to use systems that do not make good use of these techniques. While we are teaching about how to use a computer for other tasks, we are not using the computer to its best effect for the learning and teaching task itself. This is, admittedly, not true for every institution, but the number of universities, for instance, still relying on text based command line interfaces for first year programming is still significant. And even in conventionally used graphical environments, good support for well known techniques such as experimentation, interaction and visualisation is often poor.

The main argument against the extensive use of graphics has long been the availability problem. There is little benefit in designing systems relying on technology that is not commonly available in the targeted user community. Colour graphics systems, however, are so common now that assuming their availability does not seem to be a serious restriction anymore.

The extensive use of graphics has several benefits. First of all, by providing a graphical user interface (GUI) built from standard components, a system is much easier to use for novices than systems with comparable functionality and a text based interface. Most of the students coming into university today have prior experience with computers and graphical interfaces. They are familiar with menus, buttons, windows and a mouse. GUIs support exploration and self guided learning by making

it easier to find out about the functionality of the system. Secondly, graphics allow us to better support visualisation techniques. One of the central features of the Blue environment is the graphical display of the project structure. Sometimes the old saying "a picture is worth a thousand words" really is true. (Although it must be viewed critically in the context of programming – it is not always clear what a picture really expresses or, as Petre asks [Petre 1995]: "Does a given picture convey the same thousand words to all viewers?")

Thirdly, graphics are fun. It is obvious that students like working with graphical systems much more than working with text-based counterparts. Even though some studies have shown that text based systems can be superior for certain interface tasks, the psychological effect – that things *seem* easier or *seem* faster – should be taken seriously. In a paper comparing graphical and textual representations [Petre 1995], the author states: "The importance of sheer likability should not be underestimated; it can be a compelling motivator. In general, affect may be as important as effectiveness. The *illusion* of accessibility may be more important than the reality."

The Blue interface aims to appear clear and to be easy to use. The challenge lies in the design of a system that has all the functionality of a full software development environment but appears simple and non-threatening to the user. One of the important factors in this respect is the number of visible interface components on the screen at any time. Blue uses relatively few menus and few buttons in its main window. Users can very quickly know or guess the functionality of most of the buttons on screen. This creates an important effect: users can feel relatively confident while working in the environment from a very early stage. Being confronted with a large number of mysterious buttons at the very beginning can have a very strong negative effect on the user; an immediate feeling of resignation and insecurity is often the result.

To support clarity, most of the buttons have text labels rather than icons for identification. Text on the button serves as a much better hint to its functionality than do most icons.

The issues mentioned here – simplicity, clarity, support for exploration – influence the design of each of the system's components and will be discussed further as each of these components is presented in more detail.

7.3 The project

7.3.1 Working with structure

As mentioned above, the project window is the first window users see when starting up the Blue system. The project structure (classes and their relationships) is displayed on the screen. The effect of this is that the first thing users see and interact with are classes. This meets a goal stated earlier in our discussion: that an object-oriented environment should support classes as its fundamental unit of abstraction.

The effect is twofold. Firstly, students are immediately aware of structure and classes as the basic structuring unit. When we start teaching with Blue, the first thing we show to the students is an existing project including a handful of classes. Students see the project representation and are shown how to execute the application. After experimenting for a while with the application itself, they are encouraged to explore its implementation: to look at the source code, to make guesses as to the meaning of certain statements and to make small modifications (such as a change to a string literal in the program). With relatively little guidance they encounter the full edit–compile–execute cycle and very quickly get accustomed to it. All this happens in their very first lab class. An implicit effect of the visualisation of the project structure is that students understand, from the very beginning, the idea that an application is a set of cooperating classes. These classes are distinct entities, they each have their own source code, and they form relationships with each other. All of this is conveyed implicitly by the visualisation and interaction technique; not one word of instruction has to be given about this in the laboratory class.

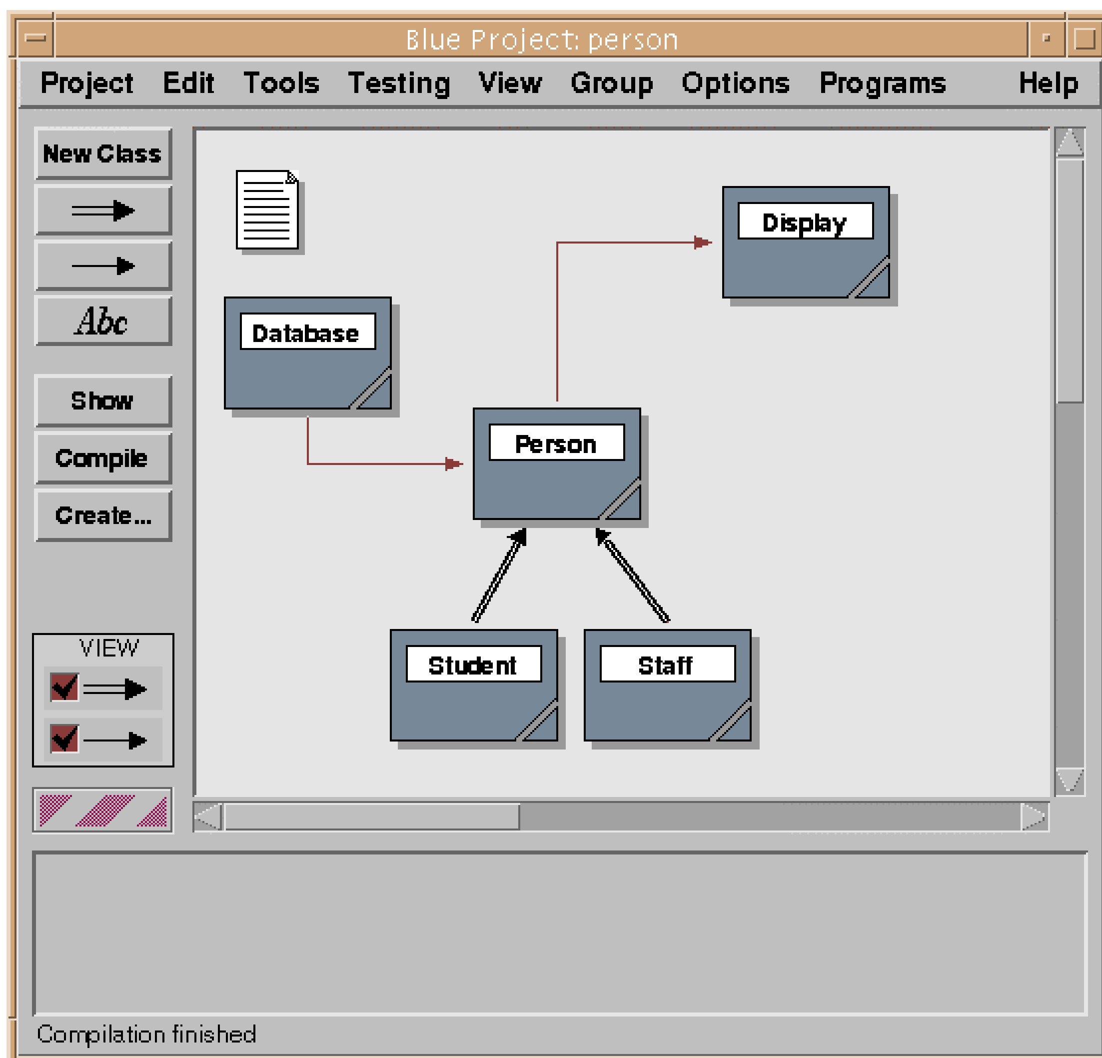

Figure 7.2: Inheritance and uses arrows in the project display

The second aspect of the structure display takes effect in more advanced exercises: when students create a new project from scratch. (Note that we treat this as an advanced exercise. The first things students do is to read and modify existing projects. There they deal with the modification or implementation of single classes. Starting a new project from scratch involves creating a new class structure – an exercise that is done later in the course.) The Blue interface ensures that no code can be written before a class has been created. The environment structure strongly encourages the creation of the project structure before writing lower level code. The classes can be created interactively and are then represented by an icon on the screen. They can be manually arranged and relationships can be defined. Using the arrow buttons from the toolbar allows the insertion of class relations in a similar fashion to the manner in which lines are drawn in a simple drawing program. The arrow can be dragged from one class to another to establish a relationship.

Blue distinguishes two kinds of relationships in its display: inheritance and "uses" relations. Inheritance relationships are represented by a double arrow while uses relationships are represented by a single arrow. The arrows also differ in layout and colour to be clearly distinguishable on screen (Figure 7.2).

This presentation of a project is quite different to the presentation in most existing development environments. All the popular systems evaluated in chapter 4 make use of graphical user interfaces, but rely on text based interface components to give information about the project and the classes. The project is typically represented as a textual list of classes that does not convey much information about structure or relationships between these classes. The list typically serves only as information about the existence of the classes in the project and as a shortcut to open the class's source code. Library classes used in the application are typically not listed. All significant work (reading, understanding, editing the project) is done in a textual view of a single class's source code. Thus, the student is immediately forced to think at the code level of an application and must invest considerable mental effort to deduce the logical application structure.

We have for a long time, in teaching well structured programming, told students to think about program structure first, before starting to write low-level code. This is true for structured programming approaches just as well as for object-oriented ones. Students tend to need a lot of convincing to take this design issue seriously. Since all that is done on the computer itself is the typing in of the code, this is seen as the "real" programming task. Creating a design in advance is frequently seen as a nuisance that they are forced to do *before* the programming work is done (instead of regarding it as *part of* the programming task). Consequently, students often do not take the design phase seriously enough and spend too little effort thinking about it.

The Blue system, by incorporating the class structure into the machine manipulated part of the programming task, has the effect that students take the creation of the structure more seriously. The interface almost forces them to think about class structure before they can start writing code.

The layout of the class diagram on screen is semi-automatic. The position of the classes can be manipulated manually (by dragging the icon) and the arrows are layed

out automatically. The layout algorithm attempts to create an aesthetically pleasing diagram, for example by trying to avoid unnecessary crossing of arrows. This approach has proven to be very successful. Classes can usually easily be arranged in a way that provides a clear and well layed out project overview.

7.3.2 Design notation

As we have seen, Blue uses a very simple design notation. The structure display shows classes, their names, inheritance relationships and uses relationships (Figure 7.2). This diagram is considerably simpler than notations commonly used for professional software engineering, such as Booch Diagrams, OMT or UML Class Diagrams. Those notations record more information about both the classes and their relationships.

In UML, for instance, class representations include a list of operations (interface routines) and data (instance variables) in addition to the class name. Relationships are distinguished as generalisation (inheritance), composition (a special kind of a "uses" relationship) and other uses relations called "associations". Associations are annotated to distinguish, for example, "part of", "contains", "member", "manages" and "communicates with" relations (and many more).

We consider such a detailed notation as inappropriate for introductory teaching. It is undoubtedly valuable for professional software development, but poses too much of an overhead and distraction from more fundamental issues in an introductory course.

Much simpler systems have been developed to support teaching and learning of object-oriented concepts on a more informal level. The best known is the use of *CRC cards* [Beck 1989]. CRC stands for "Class, Responsibility, Collaboration". The CRC card method uses index cards (one per class) to record each class's name, its tasks (responsibilities) and other classes with which it cooperates (collaborations). The class cards can then be arranged on a table to form a class diagram.

The Blue diagram is more closely related to CRC cards than to the professional design notations. It shows (as do CRC cards) the class name and the collaborators. It allows a manual layout that provides additional cues through layout options such as order and grouping (e.g. classes can be shown close together to indicate a close relationship). It does not, however, show responsibilities at the diagram level.

The Blue diagram is not intended to replace CRC cards (or equivalent design tools), but rather to complement them and provide a link between design and the coding stage. We have had very positive experiences with the use of CRC cards for teaching object-orientation to beginners. One of the important aspects of CRC cards is that it is a method that does *not* use a computer. The direct interaction and possibilities of simultaneous manipulation by several people, which results from their real physical existence (as opposed to a representation on a computer screen), is very valuable. The Blue diagram provides a simple tool to record the work done with CRC cards and continue the development process in a seamless way.

7.3.3 System abstraction

Users interact directly with classes by positioning them on screen, opening them (to see or edit their source code) and creating objects from them (see below). Throughout the whole interaction process the underlying operating system is hidden. Files, for example, do not appear as part of the programming task. The source of a class may be edited (see section 7.4, below), but the user does not need to worry about the technology used to store the text.

Equally, when a project needs compiling, all the user has to do is to click the "Compile" button in the toolbar. This compilation operation does a full dependency analysis and compiles all classes that need recompilation. It takes care of a number of subtle, but important details. When a class is modified, for example, the algorithm will recognise correctly whether a change was made to the interface or to the implementation only. It will then, depending on the result, decide whether clients of this class need recompilation. Circular dependencies are also handled correctly. If necessary, an interface analysis is done first to ensure that mutually dependant classes are compiled correctly. (The menu also contains operations to compile selected classes, or to force a recompilation of the whole project. Users are still in control to do what they want. But the "intelligent" compilation is by far the most frequently used compilation operation.)

Both these examples (automatic storage management for classes and compilation management) show that the user ↔ system interaction is at a *logical* level, not a *technical* one. Users interact with logical abstractions (classes, not files) and they perform logical operations while being freed from technical considerations.

When we, in an earlier chapter, listed characteristics that a good object-oriented environment should have, one of the central issues was the support of classes and objects as the fundamental units of abstraction. We have seen how classes are supported in Blue. Support of objects will be discussed in section 7.6. Before we go on to objects, however, we discuss some operations on classes in more detail: editing and compiling.

7.4 Editing

7.4.1 Class views: interface and implementation

The editor is fully integrated into the Blue environment. It does not appear as a separate tool, but rather as a function of each class: the class can be *opened*. It is, in fact, more than an editor in the strict sense of the word, since it provides functionality beyond that of text editing, such as displaying the interface of a class. It can better be thought of as a viewer of class details (with the ability to edit some of the details).

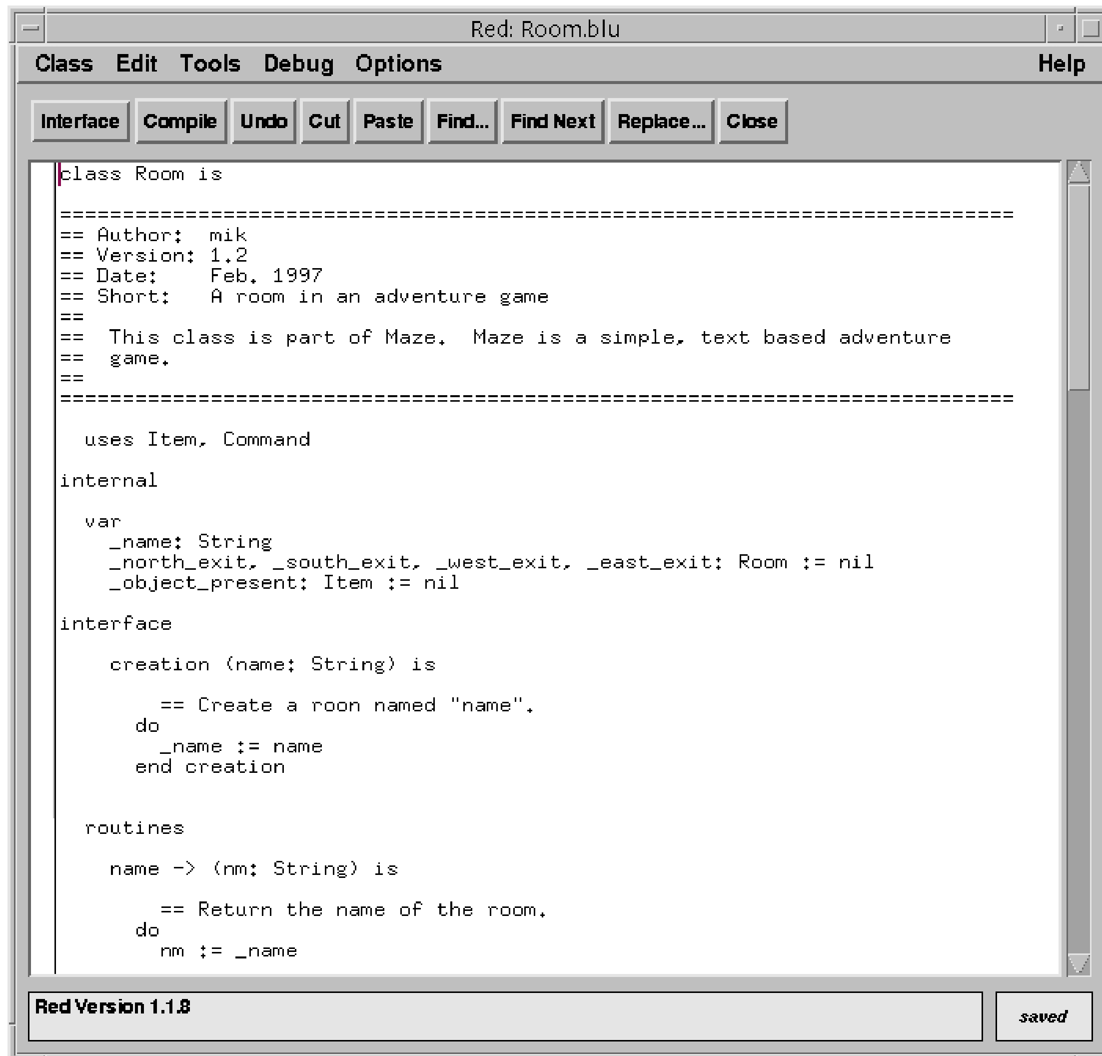

Figure 7.3: Implementation view of a class

From the project window, a *show* operation can be invoked on a class by double-clicking the class icon or clicking the "Show" button. This operation will open a window displaying class details – either the source or the interface of each class. The two views are usually referred to as the *implementation view* and the *interface view* (Figure 7.3 and Figure 7.4). Which view a user wants to see depends on the context: during development or maintenance of a class, the implementation view is used. After the implementation of a class is finished (e.g. while a client class is being developed) only the interface view is needed.

The interface view not only includes the routine headers, but also pre and post conditions and interface comments. (Section 6.7.2 introduced the distinction between interface comments and implementation comments in Blue.) Thus, the interface provided by the interface view does not only provide syntactical information, but serves as full documentation of the class.

Multiple classes can be opened at the same time. The views may be toggled by using the "Interface" toggle button in the toolbar of the editor. When a class is opened, the

view that was last used is initially displayed. If the class has not been opened in this session, the operation depends on the compilation status of the class: if it is compiled, the interface view is shown; if not, the implementation view is used.

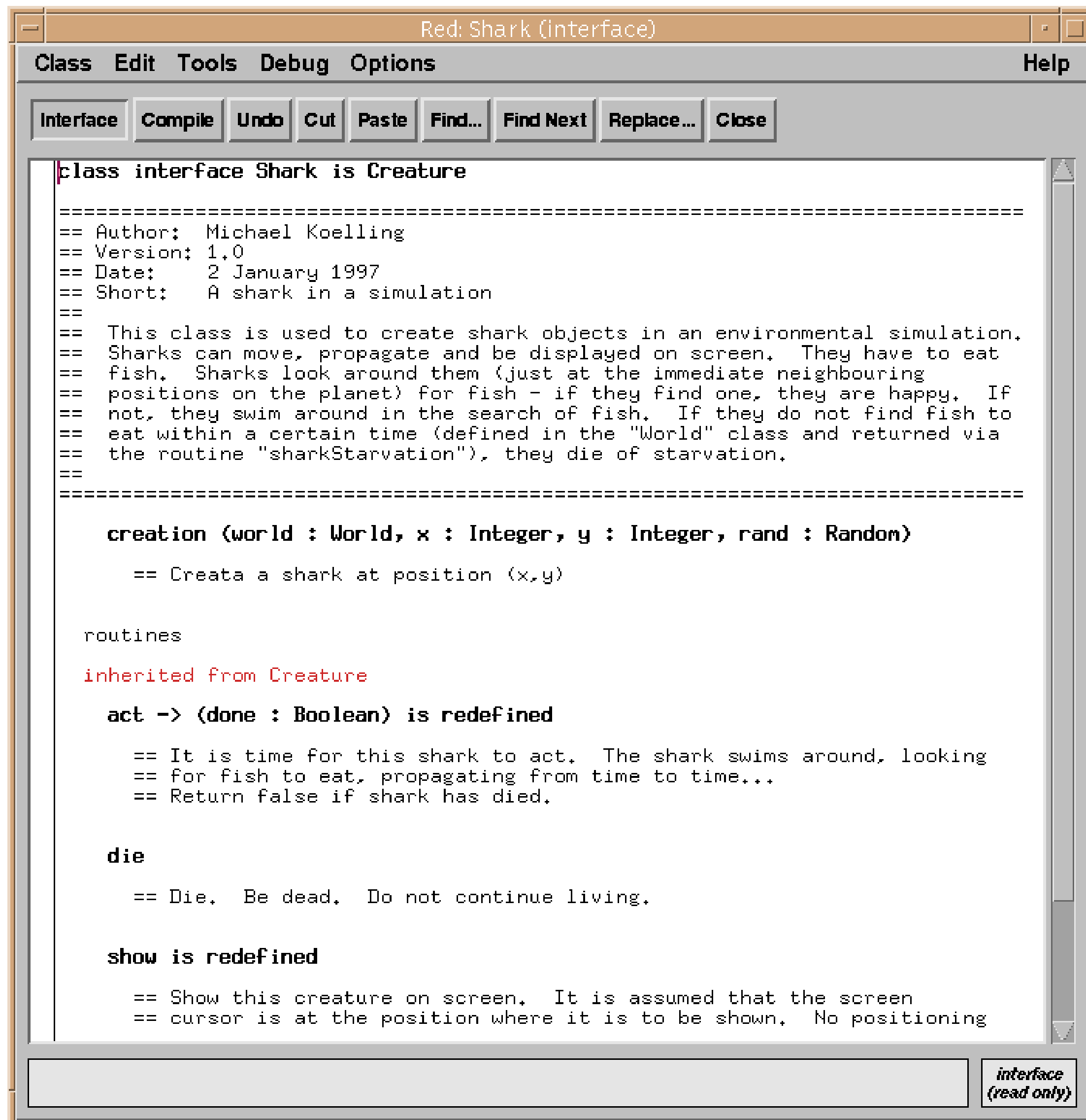

Figure 7.4: Interface view of a class

The interface view can only be shown when the class has been compiled. (For uncompiled classes the "Interface" button is greyed out.) The information shown in the interface view is constructed from the symbol table produced by the compiler. This is possible because of the tight integration of the environment's components. The compiler produces the symbol tables and stores them after compilation for future reference. They are then used by the editor (for presenting the interface view), by the project manager (to check whether an interface of a class was modified) and by the debugger.

Constructing the interface view from the symbol tables rather than the source has two advantages: firstly, the information used to display the interface has been checked by

the compiler and is thus guaranteed to be syntactically correct. The class has been successfully compiled and thus the displayed interface is indeed available. Secondly, using the symbol table makes it easier to include inherited information correctly and makes it possible to display meta information.

Including inherited information in the interface view is important in its typical use. While the implementation view is used during development of the class and contains only the source of that particular class, the interface view is typically displayed when a class is being used by a client class. In this situation the user typically wants to know what operations can be invoked on objects of this class (and what the exact interfaces of those operations are). This clearly includes inherited operations. When using an object it does not make any difference whether an operation was defined locally or inherited. Consequently, the interface of a class consists of all routines, inherited or not. All of these are shown in the interface view.

A maintenance programmer, however, after checking the interface of a class, may need to find a routine's implementation. In that situation it would be a disadvantage to make inherited operations appear the same as locally defined ones – the programmer might have to search through a hierarchy of parent classes to find the implementation of the routine. Here, compiler meta information can be used to help. From the symbol table, the information about the class originally defining the routine (and whether it has been redefined) is read and displayed in the interface view (see Figure 7.4).

Viewing interfaces is important in all object-oriented systems. Eiffel environments also provide tools to produce interface views (one tool called "short" to display the interface of locally defined routines and a tool called "flat" for the inclusion of inherited attributes; the combination of "short" and "flat" provides a complete interface of a class, similar to the Blue interface view.) While early versions of these tools did not include comments, the latest versions do: they recognise some comments placed at special locations as interface comments. C++ systems use another approach. In C++, the programmer must provide separate header files, which serve as interface descriptions. This makes it necessary for the programmer to duplicate information. The routine headers and associated comments have to be typed twice. It also has the effect that inconsistencies can appear very easily when a change is made, for example, in the comment in the implementation file but not in the counterpart in the interface file. Providing tools to automatically generate interface views from a single source removes the necessity to duplicate information and avoids inconsistencies.

7.4.2 Graphical vs. textual editing

The implementation of a class may be edited. The editor provides full standard text editing capabilities and several advanced and specialised features. It may be used by students unfamiliar with editing techniques as a simple GUI editor using mainly the mouse and pull-down menus. However, it also has a powerful and configurable command set allowing more experienced users to perform more complex operations. Several Blue specific functions are supported, such as the insertion of skeletons of frequently used code structures (loops, conditionals, routines, etc.). It is, like the rest of the Blue environment, designed to be easy to use initially to support beginners, but

to be able to be used in a more powerful way as the experience of the user grows. The aim is to support beginners, but not to keep them on a beginners level for a long time. A full editor manual is available [Kölling 1997b].

The editor is tightly integrated with the project manager and the compiler. When a new class is created in the project window, a code skeleton is automatically created. The initial source is a valid (but empty) Blue class. This ensures that a project can immediately be compiled after classes have been added (so that the user can work on one class at a time, without having to open all classes created in a project before the first full compilation).

The automatically created skeleton together with the strict class structure of Blue make it very easy for students to recognise where different aspects of the source, such as variables and routines, have to be inserted.

The project can be edited graphically (by inserting arrows into the class structure diagram, as discussed above) or textually in the editor. Both views are kept consistent at all times. If, for example, an inheritance arrow is inserted graphically, the source of the class is automatically updated. When the class is opened, the inheritance definition is visible in textual form. (If the class was open at the time of the insertion of the arrow, the source code is immediately updated on the screen.) The reverse case also works: when a definition of a uses or inheritance relationship is added to the source of a class, the project diagram will be updated as soon as the class is saved. Closing a class view automatically saves the source, so the standard sequence of opening a class, modifying its definition and then closing it leaves a diagram that always reflects the current relationships. To achieve this effect, the editor cooperates with the compiler. After saving the class's source, it calls the compiler to perform a superficial parsing of the class to extract the information about its superclass and used classes. The compiler then reports the result to the project manager, which updates the graphical project representation. This analysis is, even for large classes, fast enough that it is not noticeable in normal interactive operation.

It is important to allow both graphical and textual editing. In the early stages of program design and class structure definition, we do not want students to have to descend to the lower level of source code definition to specify their application's structure. Relationships should be specified using a higher level tool (the graphical structure editor). Later, however, when the source code is under development, a user must have the opportunity to make changes without being forced to leave one tool and use another one, only because a definition was originally made using that other tool. At the same time it is, of course, essential that the information displayed in both tools is consistent.

7.4.3 Alternatives

Initially, it does not seem very sensible to implement yet another text editor. Many good editors are already available and many users have their preferred editor. In fact, discussions about the relative benefits of particular editors frequently take just as much the form of religious debates as discussions about programming languages.

Users feel strongly about their tool of choice, and it would be a benefit to allow them this choice.

However, there are obvious problems. Much of the benefit of the system described above comes from the tight integration and specialised features of the Blue editor. We have seen examples of its close cooperation with the project manager and the compiler, and in the next sections we will discuss equally tight integration with the debugger and the library browser.

All of this could not be achieved to the same degree with a standard editor. Even if the integration could be partially achieved with a highly customisable editor, such as Emacs, this would not solve the problem. Allowing the use of standard editors only makes sense if there are *several* editors available to be used in conjunction with the Blue environment (otherwise we would not give the user a choice either, but still prescribe one particular editor). We cannot see how functionality even remotely resembling that provided by the present Blue system could be achieved with standard text editors such as “vi”.

There are, however, some aspects which may serve to alleviate the situation. Firstly, the expected users of the environment are largely beginners, not professional software engineers. Beginners often have not yet developed such a strong preference for a particular editor. They are more willing to use a new editor, particularly since the Blue editor provides better functionality than the editors the students are most often accustomed to – those which are typically available on Windows systems. In an environment for software professionals the question of supporting different editors might well be decided differently.

Secondly, the editor provides user-definable key bindings. With this functionality, the key bindings can usually be defined to resemble many other editors. Thirdly, for those users who regularly use Blue and also do other editing tasks and are irritated by the use of different editors, another solution is available: the Blue editor can be used for all other editing tasks as well. A stand-alone version of the editor (named *Red*) has been developed as a general purpose text editor. Many students who start programming with Blue indeed use Red for all other editing tasks during first year and in later years of study.

This makes the situation slightly better than with most other environments. Almost all graphical integrated development environments supply their own text editor which cannot be replaced by another editor of choice. In comparison, the Blue editor offers better functionality then most, and the fact that it is available as a stand alone text editor can serve to avoid having to use multiple editors.

7.5 Compiling

7.5.1 Invoking the compiler

The compiler, just as the other environment tools, does not appear as a separate tool, but is integrated into the project interface. We have already mentioned the “Compile” button in the project window: it performs a sophisticated dependency analysis and compilation sequence, possibly involving several compilation phases to deal with circular dependencies. An operation to compile individual classes is also provided in the project window.

The compiler also cooperates with the editor. It can be invoked from within the editor, and it uses the editor for the display of error messages.

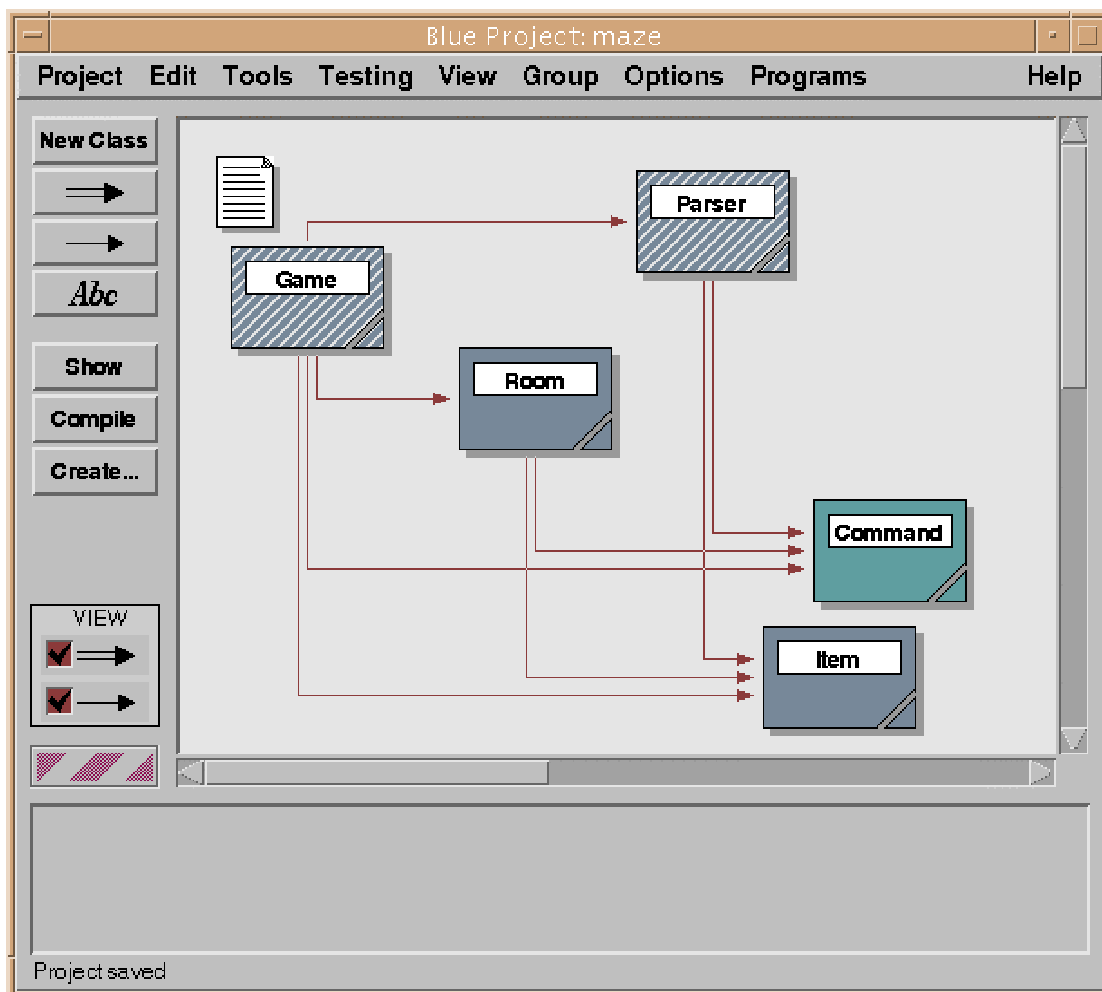

Figure 7.5: Icons indicate whether a class has been compiled

Compilation is initiated from within the editor by clicking on a “Compile” button in the editor toolbar (see Figure 7.3) or by a keyboard shortcut to invoke the same function. The compile button in the editor has a different functionality from the “Compile” button in the project window – it compiles only the class currently being

viewed. This is consistent with the object-oriented view of the world: “Compile” in the project window invokes the compile operation on the project, whereas “Compile” in the class window invokes the class compile operation. The tools themselves can be viewed as objects which provide operations to be performed on them.

Once a class has been compiled, its appearance in the project diagram changes. Compiled classes appear solid, whereas uncompiled classes are striped (Figure 7.5).

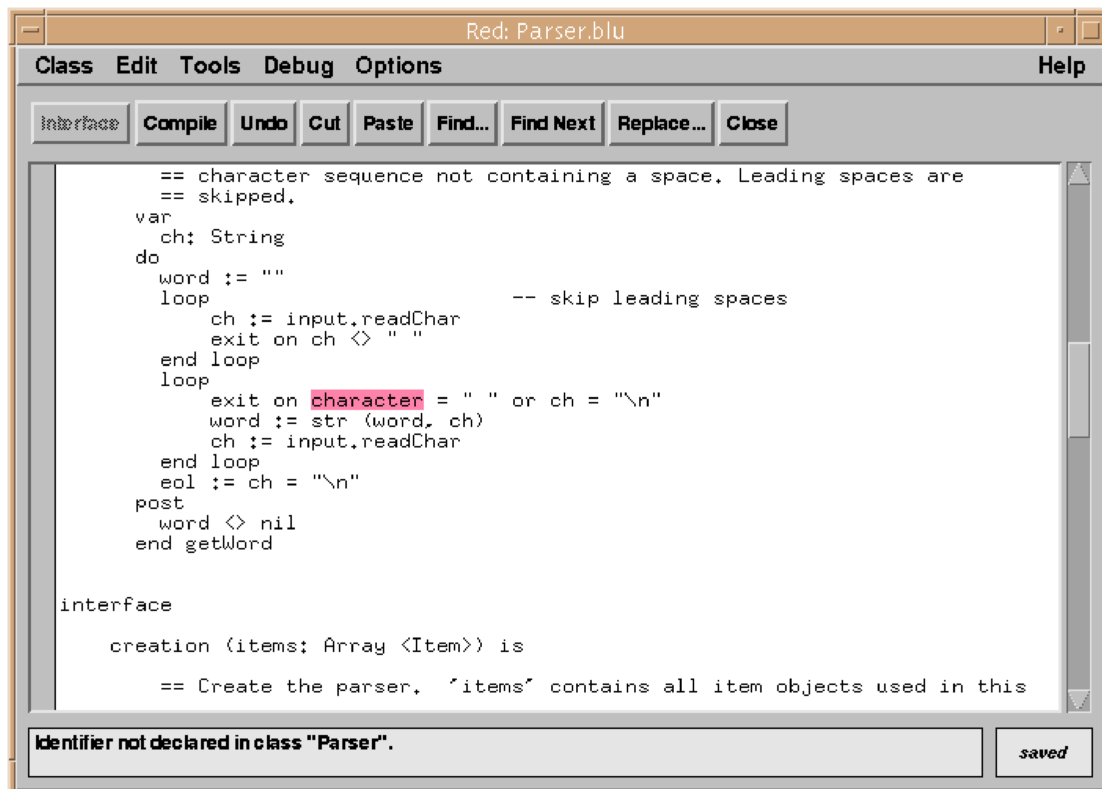

Figure 7.6: An error message reported by the compiler

7.5.2 Display of error messages

When the compiler detects an error in a class, it displays the class source, highlights the region of the error and displays an error message in the information area of the editor (Figure 7.6). Displaying the class source may involve opening the class, bringing its window to the front, de-iconifying its window or switching from interface to implementation view. No file name or line number information needs to be handled by the user – the environment automatically processes this information and finds and displays the relevant code fragment. The user can then immediately edit the class and remove the error. The error messages have been carefully worded to be comprehensible to beginners. Jargon is avoided as much as possible, and the messages attempt to give precise information as to the source of the problem.

Good error messages make a big difference in the useability of a system for beginners. Often the wording of a message alone can make all the difference between

a student being unable to solve a problem without help from someone else and a student being able to quickly understand and remove a small error. The first student might be delayed for hours or days if help is not immediately available (and even in a class with a tutor it may take several minutes for the tutor to be able to provide the needed help). Enabling students to understand errors by themselves avoids a lot of frustration and frees the tutor's time for more important tasks. This can be achieved to a large degree through the use of carefully worded messages and is a goal well worth the investment of significant effort.

The quality of messages that a compiler can produce is significantly influenced by the grammar of the language itself and the compiler technology used. In languages like C++, so many ambiguities exist that it is impossible to avoid producing very general or misleading messages. Blue has been carefully designed to provide a type of grammar and a degree of redundancy within the grammar that enables the generation of good error messages in most cases. The grammar has been designed to be an LL(k) grammar (with k=2 in the case of Blue). This enabled us to use a recursive descent compiler – a compilation technique that lends itself more easily to the production of good error messages than LR compilers.

In case of an error during compilation, the Blue compiler displays only the first error message and aborts – no attempt at error recovery is made. In an earlier design, we discussed an alternative where the compiler attempts to detect all errors in a class. It would somehow mark the errors in the source, but only the first error can be initially displayed (the others might not be on the same screen). The editor would then have a "Next Error" button that takes the user to the location of the next error.

In practice, that design has proved to be unnecessarily complex. Compilation in Blue is so fast (usually under a second for a class of 2000 lines) that the "Compile" button works as a de-facto "Next Error" button. After the first error is fixed, the user just clicks "Compile" again and the next error will be highlighted almost immediately. This has several advantages: the compiler is simpler (because no error recovery has to be attempted) and the error messages are more accurate (because incorrect messages caused by preceding errors are avoided). The same strategy is applied by the Visual Age environments, while most other systems display a list of errors which can then be used to locate all found errors. The problem often is that, after the first error in the list has been fixed, following messages are inaccurate and misleading.

Some more issues concerning the implementation of the compiler are discussed in chapter 8.

7.6 Interacting with objects

7.6.1 Calling interface routines

As soon as a class within a project is compiled, objects of that class may be created. (A brief description of this interaction mechanism has been published in [Rosenberg

1997a].) Interactively creating objects is done by selecting the class and clicking the "Create" button. (Clicking the right mouse button on a class provides a shortcut to the same function. Shortcuts exist for many frequently used functions. They will not generally be mentioned here. For a full description of environment functions and their shortcuts, see [Kölling 1998b]).

This operation is similar to interactively sending a "new" message to a class in a Smalltalk environment. An instance is interactively created and available for operation. No equivalent of this operation is available in common environments for more recently developed, statically typed programming languages.

```
class interface Person is
=====================================================
== Author:   M. Kölling
== Version:     1.0
== Date:        8 June 1998
== Short:       Person class for university management project
==
==  The class Person implements object representing a person in a
==  university management project.  It contains information common
==  to all persons in the university...
==
=====================================================
   creation (firstName : String, lastName : String, age : Integer)
      == Create a new person with given name and age.
      pre
         lastName <> nil and age <> nil

 routines
   changeNames (firstName : String, lastName : String)
      == Change the names for this person
      pre
         lastName <> nil

   changeAge (newAge : Integer)
      == Set a new age for this person
      pre
         newAge <> nil

   getNames -> (firstName : String, lastName : String)
      == Return both names of this person

   getAge -> (age : Integer)
      == Return age of this person

end class
```

Figure 7.7: Interface of class "Person"

Invoking the creation operation on a class results in a normal object creation, including the execution of the creation routine. As an example, we will use a class "Person" which stores some information about a person and provides interface routines to change and access that information. This class is not meant to be complete or really useful in any sense – it is used here only as an example to demonstrate the Blue object interaction facilities. The interface of the class is shown in Figure 7.7.

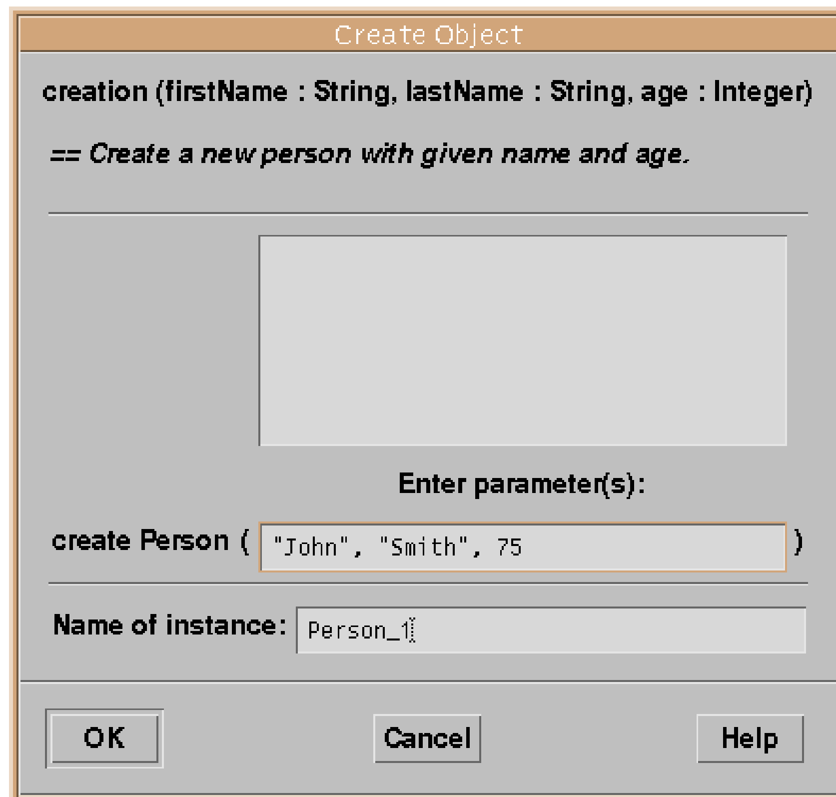

Figure 7.8: Object creation dialogue

When the create operation is invoked a dialogue is displayed to let the user enter routine parameters (Figure 7.8). At the top of this dialogue, the interface of the creation routine is displayed. The interface includes the routine header and the routine comment. Further down is a text field for entering parameter values. Under the parameters is another field to provide a name for the object to be created. A default name is provided and is often adequate. The name will be displayed on the object bench after it has been created. The large area in the middle of the dialogue is a (currently empty) list of previously used parameter lists. It is provided for convenience during testing of a class: previously made calls can be easily repeated by selecting a parameter combination from the list.

Once the dialogue is filled in and the OK button is clicked, the object is created and displayed on the object bench (Figure 7.9). The object is then available to the user for direct interaction. Many different objects of the same or different classes may be created and stored on the object bench at the same time.

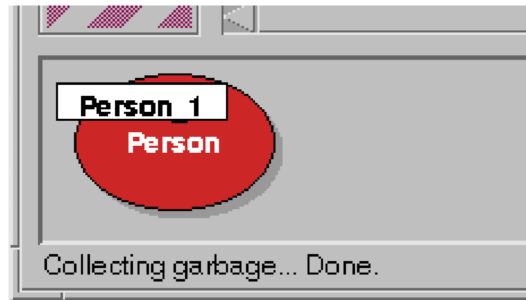

Figure 7.9: An object on the object bench

Clicking on the object with the right mouse button displays a menu that includes all interface routines of that object (Figure 7.10). Also included in the menu are two special operations available for all objects: *inspect* and *remove*. The remove operation removes the object from the bench when it is no longer needed. The inspect operation is discussed below (section 7.6.4).

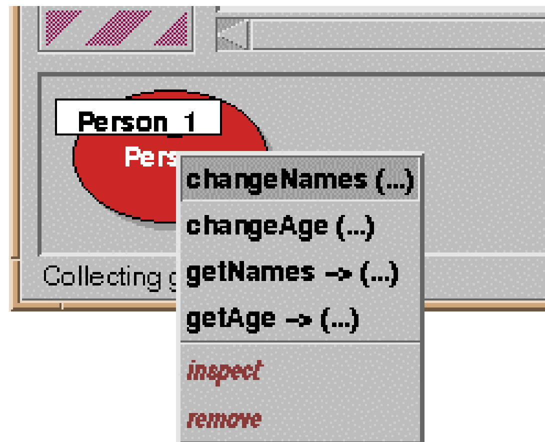

Figure 7.10: Calling a routine on an object

Symbols in the routine menu indicate whether a routine has parameters or return values. When a routine is selected from the menu, a call to that routine is executed. If the routine has parameters, a parameter dialogue similar to the one seen at the creation of the object is displayed (Figure 7.11). On the click of the OK button the routine is executed and, if the routine returns results, the result values are displayed in another dialogue. Figure 7.12 shows a function result dialogue for a call to the routine "getNames". Again, at the top of the dialogue window the interface of the called routine is displayed. Below, the actual call is shown in standard Blue syntax (the name of the called object, the routine name and – if present – actual parameters). This is followed by a list of the result values of the function. For each result its name, type and value are displayed.

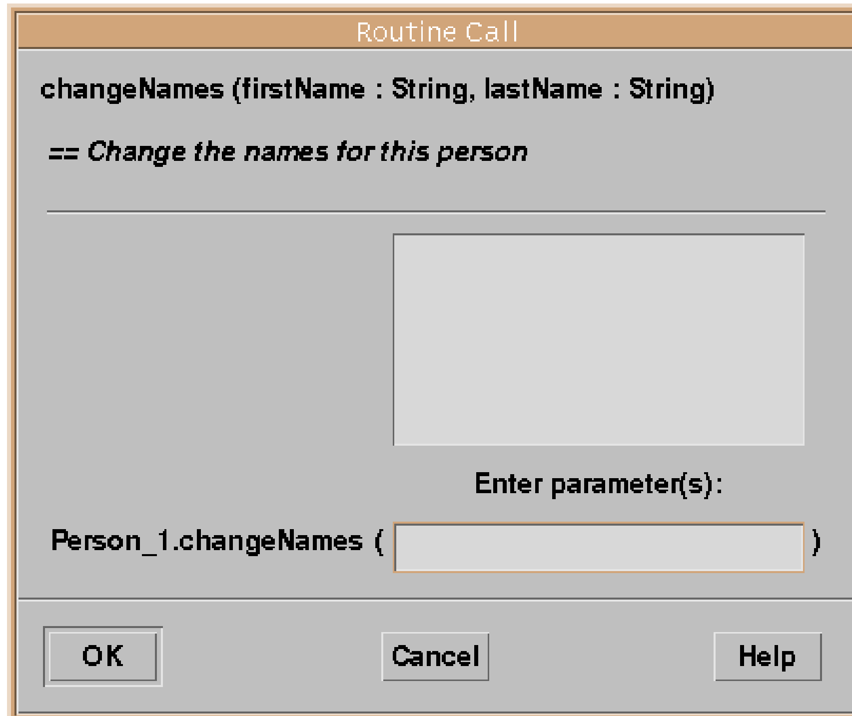

Figure 7.11: Routine call dialogue for "changeNames"

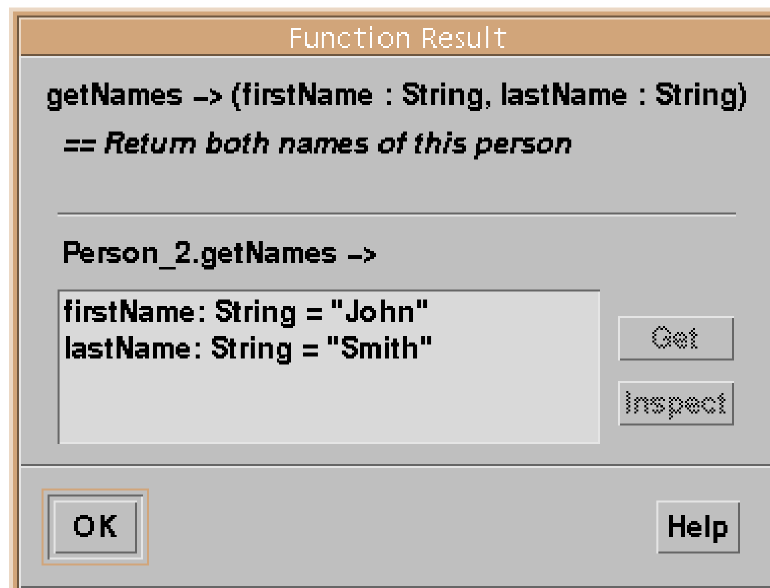

Figure 7.12: Result dialogue for function "getNames"

7.6.2 Linguistic Reflection

To execute an interactive call, the Blue environment uses linguistic reflection. A class is constructed internally that includes the interactive call as the only statement in its creation routine. This class is then passed to the compiler to be translated. An object of the resulting class is instantiated which, as part of the creation, executes its creation

routine and with it the interactive call. Result values are stored in this internal object and can then be extracted for display in the result dialogue. To illustrate this technique let us consider an interactive call to the following routine:

```
extract (line: String, o: Object) -> (word: String, valid: Boolean)
```

The actual call we want to execute is

```
parser.extract ("input line", object1)
```

We assume that `parser` and `object1` are the names of objects on the object bench. To execute this call, the Blue system internally creates the source for another class, usually referred to as the *shell class*. The source code created for our example call would look like this:

```
class __SHELL__ is
  == shell class for interactive call

  uses Parser, Object

  internal var   word: String
                 valid: Boolean

  interface

    creation (parser: Parser, object1: Object) is
        == execute interactive call

      do
        word, valid := parser.extract ("input line", object1)
      end creation

end class
```

The interactive call is then executed by creating an object of the shell class. Creation of the shell object automatically includes the execution of the interactive call as part of its creation routine execution.

The shell class is constructed to have one instance variable for every return value of the interactive call. The return values are stored in those variables and can, after the call, be retrieved from the created object to be displayed to the user. The display of the return values is, in fact, nothing else than an inspection of the shell object (see Inspection of objects, section 7.6.4).

Several advantages are associated with this technique. Firstly, the parameter list does not need to be parsed and evaluated by the project management part of the system. The compiler is used for this purpose, thus avoiding duplication of equivalent code. The project manager sets up only the parameter list for the shell creation routine, which contains only object references. Secondly, error messages for mistakes found in the parameter list are produced by the compiler and are thus guaranteed to be the same messages that would be produced for the same error in a non-interactive call. This increases consistency in the environment. Thirdly, the only call ever to be initiated by the object bench (the call to the shell creation routine) has a simple and known interface. Most importantly, it has only object parameters, no literals. This greatly simplifies the implementation. The interactive call, having an arbitrary parameter list, is turned into an internal call completely handled by the runtime system.

The result of the facilities described so far (interactive creation of objects, interactive routine calls and result display) is that a project can be incrementally developed. There is no need to complete all classes in a project before the first tests can be performed. Instead, each class can be tested as soon as some of its routines have been completed without the need to write special purpose test code. This possibility dramatically changes the style of work available to the developer.

7.6.3 Composition

During the interactive testing of the system, objects accessible on the object bench may be composed, i.e. one object may be passed as a parameter to the routine of another object. If, for instance, a project includes a database class and a person class with the intention of creating a database of persons, then objects of these classes may be combined. Several person objects could be created. Then a database object is created and its “addPerson” routine is invoked. When the routine call dialogue is visible on the screen, a click on one of the person objects on the object bench will enter its name into the parameter field of the routine call dialogue. The object will be passed as a parameter. This can be done repeatedly to add all the persons from the object bench to the database.

7.6.4 Inspection of objects

Sometimes objects contain data which is not directly accessible through interface routines. For this situation the *inspect* operation is provided. Using the inspect operation (by selecting it from the object menu or, as a shortcut, double-clicking the object) opens the object and reveals its internals. Figure 7.13 shows the dialogue displayed as a result of inspecting a person object.

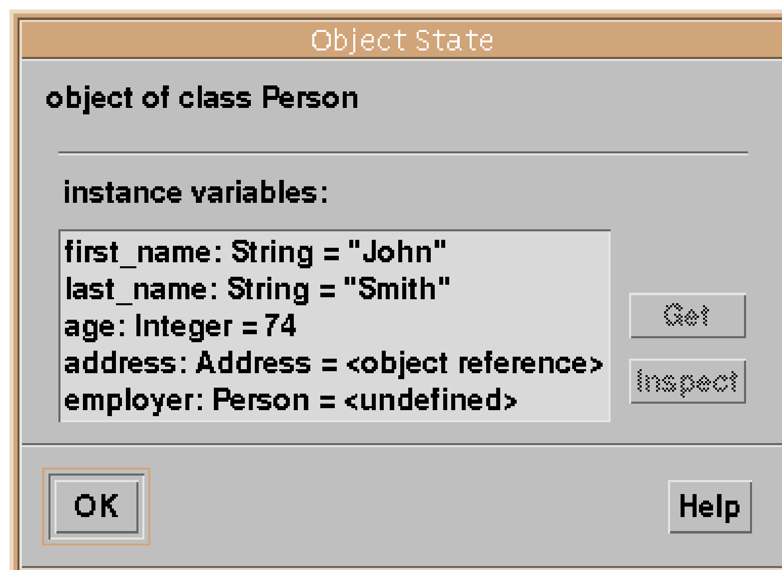

Figure 7.13: Object inspection dialogue

For this example, we have modified the above definition of the class “Person” to include address and employer variables, so that we can show how more complex objects can be inspected. The address variable holds a reference to another user-defined object of class “Address”; the employer variable refers to another person.

The names, types and values of all instance variables of this object are shown. For manifest objects, which have a simple textual representation, values are shown as literals. For variables holding more complex objects only the state of the variable is displayed (whether it is undefined, contains *nil* or an object reference). Those variables may then in turn be inspected by double-clicking on the variable or selecting the variable and then clicking the “Inspect” button. Another window will be opened displaying the internals of that object. An example is shown in Figure 7.14 for the inspection of the address object.

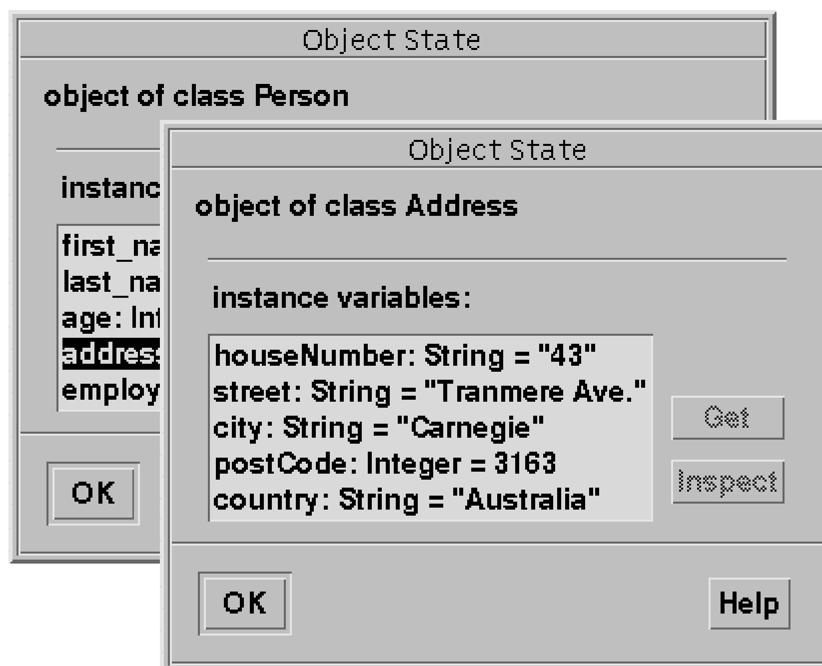

Figure 7.14: Inspection of “Address” object

Note that we are able to examine any object reachable from an object available on the object bench. Sometimes it can become clumsy to repeatedly navigate through object references to reach an object we wish to examine. The “Get” button on the inspection dialogue (Figure 7.13) allows a reference to any existing object to be placed on the object bench so that it can be re-examined at a later time. This also allows the user to interactively call interface routines of objects that were created internally.

Overall, inspection of objects assists users in thoroughly testing objects of any class by allowing users to observe the effect of routine executions on internal data.

7.6.5 A word on testing

The design and implementation of error-free programs is an extremely difficult task, even for experienced computing professionals. It is therefore not surprising that students have great difficulty in producing well debugged programs. This is further compounded by the fact that most students do not undertake rigorous testing of their code. The interaction and inspection mechanisms described above can help to address this problem.

As experienced programmers it is obvious to us that when we develop new programs we must also test them. Why is this not obvious to students? The answer may well lie in the fact that students begin by writing very small programs, so small that in most cases it is difficult to argue the case for any serious testing. The classic example of this is the infamous "Hello World" program.

The effect of the decomposition of the program into classes and the reuse of existing library code in an object-oriented environment means that students can work on larger projects earlier in the course. They need not write all of the code for the project themselves; they can write just a few of the classes and these can then be integrated with those provided.

However, this advantage in terms of structure is itself an obstacle in terms of testing. Typical object-oriented programs will have a number of classes. We would encourage students to test each of these classes individually before combining them to form a solution to the problem. In order to do this, test programs must be developed. There may well be one test program for each class. Since a class typically has several methods, these test programs can become quite lengthy and complex. In fact it would not be unusual for the test program to be longer than the class being tested. The development of such test programs can easily become more of an obstacle for the students than the development of the original classes themselves.

Clearly what is required are better tools for testing. In particular we would like to reduce the amount of code which must be written in order to test classes. Ideally no special testing code should have to be written.

Testing is typically divided into two classes of test cases: *black box* testing and *white box* testing. Black box testing uses the observable behaviour of a program unit (a class or a routine) to judge its correctness. It uses the program as a black box, observing only input and output. White box testing observes the internal workings of the code by monitoring control flow and/or internal state of objects at arbitrary times. Black box testing does not always serve to ensure correct execution of a routine (especially in incomplete classes) since the effect of a routine call may be purely on the internal data of an object that is not directly observable from the outside. Information hiding – a valuable concept for software engineering – becomes a hindrance for testing.

The traditional method of dealing with testing is “test drivers” (programs making calls to the class to be tested) for black box testing and “debug” statements in the code or the use of debuggers for white box testing. All of these cause problems. The problems

of test drivers – their complexity – has been mentioned above. What about white box testing?

First, "debug" print statements could be added into the class code to print out relevant internal data when methods are called. This has several associated problems. The insertion of new test code could well introduce errors in itself. (Most teachers are familiar with the situation where students come with an error in a program, stating that they have not changed anything – just having added a "print" statement.) In addition, if there are several classes, the volume of output can become difficult to interpret.

An alternative solution is to use a symbolic debugger to insert breakpoints and examine the data. This requires the student to become familiar with the debugger at a very early stage. Since we would like students to test the very first class they write, it may be unrealistic to expect them to learn to use the debugger at the same time.

The interaction mechanisms in Blue can help to solve these problems. Interactive routine calls provide facilities for very flexible black box testing. An advantage is not only that test driver programs do not need to be written, but also that a sequence of tests can be adjusted depending on previous test results. The total sequence does not need to be decided before the first test is run.

Object inspection provides a simple, very straightforward mechanism for white box testing. The internal state of objects can be observed to monitor state changes. This does not mean that a more sophisticated debugger is not needed – a debugger can provide additional test facilities and, in fact, Blue also includes a debugger (described later). But it means that sensible testing can be done even before the use of a debugger is introduced to students.

Finally, there is a record facility in Blue which will textually record all interactive object creations, method invocations, return values, text input and text output. This may be used by students as part of an assignment submission to document the testing that was done.

Without good test support a move to an object-oriented language may well have the effect that testing is done in an even less adequate manner by students than in traditional environments because a test program must be written for each class. In addition, the encapsulation of data within classes complicates testing.

Since these tools are a part of the standard environment and are simple enough to use to test the first programs written by a student, testing is actively encouraged and considered to be a part of the normal program development cycle. We expect this to result in an improvement in the reliability of student programs and a better appreciation by the students of the importance of thorough testing.

7.6.6 Execution without I/O

An interesting effect of the direct interaction and inspection mechanisms is that significant code can be written, executed and tested without the need to use conventional input/output constructs.

In traditional systems it is essential to teach output, at least, very early in the course, probably in the first or second week. Output usually is essential in order to see a visible result from a program. Output of results allows a program to be tested and it provides motivation for students because they can see the results of their efforts. Input may be delayed a little longer, but is required as soon as we wish students to write programs which have some controlling parameters.

While these aspects make it necessary to teach about input and output early, there are some arguments against this. By its very nature I/O does not fit in well with programming language design. Every language designer has encountered the problem that the task of including useable I/O facilities invariably seems to make it necessary to break some of the rules or design principles of the language. In many languages with clean and simple concepts I/O is the "odd one out".

The fact that I/O is special is mainly related to the problem of type conversions in a typed language: I/O to and from a terminal or a file is typically represented by a character stream, while the data expected may be an integer, a string or any other type. Another argument is convenience: while routines in many languages have a fixed number of parameters with fixed types it has proven practical to have an output statement that allows a variable number of parameters with different types. The results are constructs like the "write" statement in Pascal, which breaks language rules about parameter lists, "printf" and "scanf" functions in C, which are awkward and error prone, or the use of operator overloading in C++ – a concept which is usually not discussed in the first few weeks.

Avoiding I/O statements for the first few weeks of instruction may well serve to convey a cleaner picture about concepts underlying the programming language [Rosenberg 1997b].

7.6.7 Pedagogical benefits

The facilities described in this section – interactive object creation, interactive routine calls and object inspection – lead to a number of benefits for learning and teaching object-oriented programming:

- *Incremental development*. Projects can be incrementally developed and tested. There is no need to even syntactically complete a whole application. As soon as one class (or even one routine) is completed it can be compiled, objects can be created, executed and tested. This leads to greater motivation (results are visible more quickly) and a better ability to cope with errors (since early errors can be found and removed before more errors are made, avoiding the harder to find cases of multiple interacting errors). This advantage is, in fact, not only relevant

in a learning/teaching situation, but would be beneficial for software development in general.

- *Class/object distinction*. Students often have difficulty understanding the relationship between classes and objects. Allowing the direct creation of and interaction with objects greatly facilitates the understanding of these fundamental issues. The pure act of creating a number of objects from a class demonstrates in a powerful way the respective roles of the concepts. If a student has, for example, a class "Person" and creates three different people with different names, the role of the class and the role of each object becomes much more directly understandable.
- *Programming without I/O*. As we have argued above, it might lead to a clearer understanding of the abstraction concepts if routine calls are taught before language exceptions (like I/O operations) are shown.
- *Testing support*. Good testing, essential to all serious software development, is supported much better than in conventional systems.
- *Interface/implementation distinction*. The distinction between the interface and the implementation of an object – itself an important concept – is clarified. Since only the interface operations are visible to a human user when directly interacting with an object the concept that this is the only part of an object visible to other objects seems a logical conclusion.

Overall, the interaction facility provided by the object bench constitutes one of the most powerful and most valuable learning and teaching mechanisms in the Blue environment.

7.7 Runtime support

7.7.1 Error detection

The Blue system offers sophisticated runtime support for the detection of programming errors. Runtime errors cause the Blue application to gracefully terminate and the user is informed of the cause and location of the error. This is done in a similar fashion to the reporting of compilation errors: the source window is displayed, the region of the error is highlighted and an informative message is displayed in the information area of the source window (Figure 7.15).

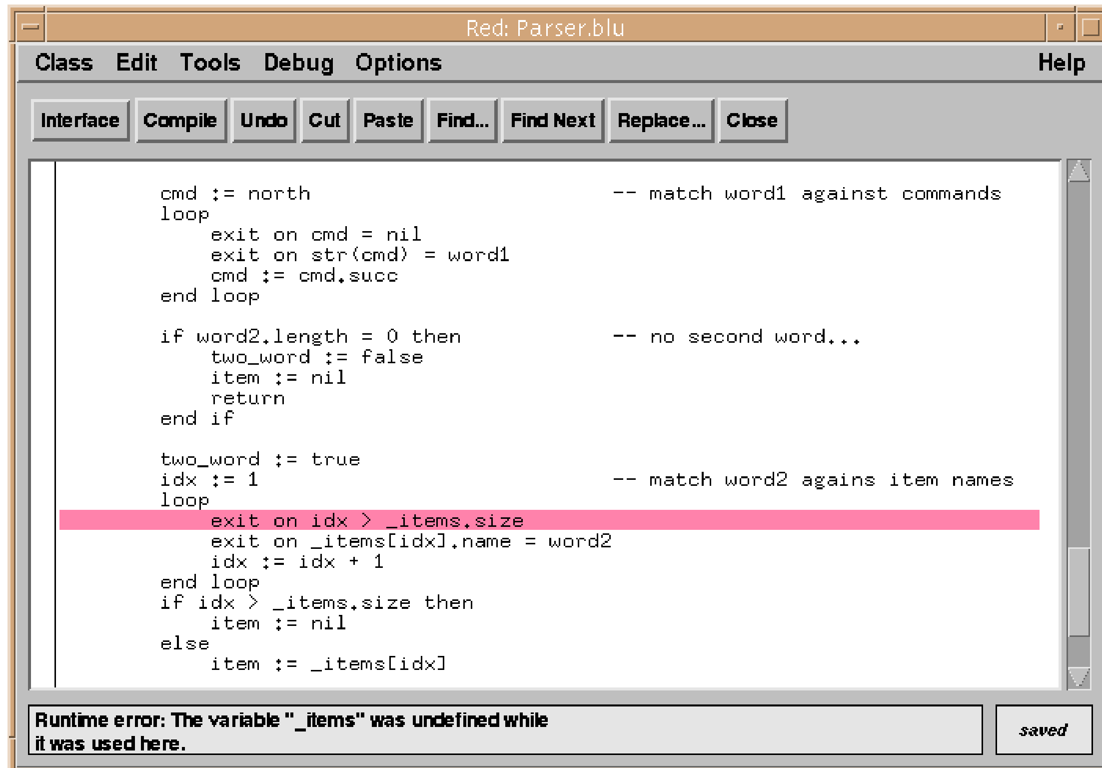

Figure 7.15: Display of a runtime error

Error detection and graceful termination is provided for all possible runtime errors. They include

- *pre condition or post condition violation*: A client fails to meet the pre condition or the server does not meet the post condition.
- *invariant violation*: The class invariant does not hold. This indicates an internal error in a class.
- *undefined variables*: The attempted use of an undefined variable is detected and reported (see example in Figure 7.15).
- *undefined return values*: An error is reported if a function returns without assigning values to all of its return variables. (If the body of the function contains no assignment statement for the return variable at all, the error is detected at compile time.)
- *stack overflow*: Stack overflow, most commonly caused by infinite recursion, is detected and reported.

Some of these examples highlight one of the aspects of the advantage that explicit teaching systems can have over general purpose systems used in a teaching context: better error checking. The use of undefined variables, for example, is a very common error among first year students. Most systems, however, do not check at runtime for this error. The result is most commonly a system crash (which, depending on the operating system and environment, may even terminate the whole environment or operating system). In any case, the student does not get any information about the cause or the location of the error.

The reason that this checking is usually not done is efficiency. Performing these checks costs time (the time to do the check) and space (Blue stores additional information about each variable, including information as to whether it has been initialised). Since this checking is only needed at development time and, in a perfectly debugged program, has no positive effect, most system providers decide against providing these checks (although the existence of a perfectly debugged program may well be doubted!). In an industry project, saving space and increasing efficiency is still seen as much more important than early error detection. This is an example where the needs of professional environments and the needs of a teaching environment directly contradict each other. The decision to concentrate on a teaching context as the intended area of use has given us the ability to provide better support for the student. The result can be less frustration (because a student can find and remove errors that otherwise would have made it necessary to wait until help is available), more correct software (because assertion violations may point to errors that would otherwise have gone unnoticed) and better support for self-directed learning.

7.7.2 Instruction counting

Blue offers an instruction counter to measure approximate relative performance of operations. The counter counts instructions executed by the Blue virtual machine. While this measure cannot be taken as an exact reflection of performance (since some machine instructions execute faster than others) it is good enough to give a correct impression in almost all cases. If, for example, a student were to compare two sort algorithms, the instruction counter can be used to measure their relative performance. Since usually two algorithms to be compared perform similar instructions, the result is of acceptable accuracy.

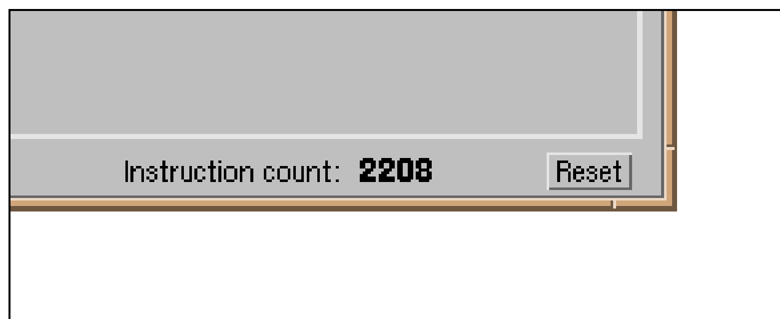

Figure 7.16: The instruction counter

The counter is hidden by default, since it is not typically used very early in a course. Thus, the interface is initially not cluttered with elements for unused features. Display of the counter can be enabled through an option in a “preferences” dialogue. When it is enabled the counter and a reset button appear in the lower right corner of the main window (Figure 7.16). It is updated after every interactive call. The number of instructions in several calls may be accumulated in the counter, or it may be reset after a call.

The instruction counter is a useful tool when learning about algorithm efficiency and complexity.

7.8 Debugging

7.8.1 Integration

In keeping with the spirit of the tools we have discussed so far, the debugger is integrated with the other parts of the system. The goal, from the users' perspective, was to avoid the common student perception that the debugger is an additional, complicated tool. This would have hindered the acceptance of the debugger by users.

Today, in most first year courses, a debugger is not used. The reason usually is lack of time. Considerable time already has to be spent on becoming familiar with other tools (editor, compiler, file system, etc.) and teachers are reluctant to further reduce the time spent on programming concepts. The effect is that, although most teachers agree that a debugger would be a useful tool in the students' effort to understand the concepts of programming, this tool is not being used.

Blue overcomes this problem by reducing the number of concepts and controls used by the debugger to keep it simple and by shifting some of the controls into tools that are already familiar. Breakpoints, for example, can be set directly from within the editor.

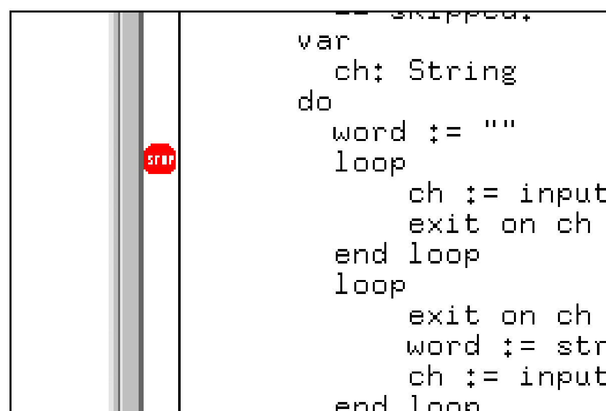

Figure 7.17: Setting a breakpoint

7.8.2 Functionality

We have already seen other parts of the Blue system that support debugging: interactive object creation, direct interaction with and inspection of objects. (This is what we call "debugging without a debugger".) The result is that the debugger functionality can be reduced to three simple concepts to provide all the debugging means necessary for a teaching environment:

- *breakpoints*. The ability to set breakpoints at arbitrary places in the source code.
- *single stepping*. The ability to institute step-by-step execution to observe a program's control flow and state changes.

- *inspection*. The ability to inspect temporary variables (local variables and parameters) and the call sequence.

The first tool, a breakpoint, is easily used. A user can set a breakpoint by just clicking into a side bar next to the source line in the editor. A stop symbol appears to indicate the breakpoint (Figure 7.17). This is an example of using a debugging technique without the need for the introduction of a new tool.

When the breakpoint is reached during execution, it is handled in a similar fashion to runtime errors. The source window is displayed, the current line is highlighted, and a message is shown informing the user that a breakpoint has been reached. At the same time, an "Execution Control" window is opened. This window contains some simple buttons that enable the user to control the continuation of the execution (Figure 7.18).

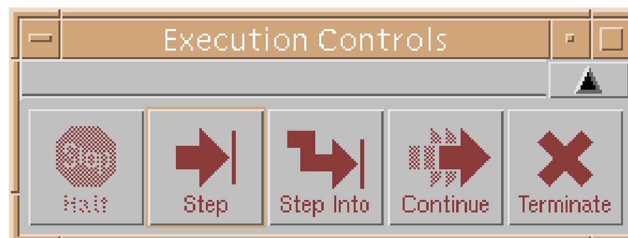

Figure 7.18: The execution controls

The execution controls let the user perform a single step (with the option to step into or step over a function call), continue, stop or terminate the execution. The stop function gives a user the ability to stop an execution without setting a breakpoint. If, for instance, a function runs for an unexpectedly long time and the user suspects that the machine might have entered an infinite loop, he/she could stop the execution (the current source line would be displayed) and examine the state of the application. The execution can then be either resumed or terminated.

To examine the state of variables and the call sequence, the execution window can be extended by clicking on the arrow button in the upper right corner (Figure 7.19).

On the left side of this window the sequence of currently active function calls is shown in a stack-like manner. On the right, the values of instance variables and local variables (including parameters and return variables) are displayed. Double-clicking an object variable opens an "inspect" window to examine the internals of that object (just as double-clicking objects on the object bench opens their inspect windows). Other active routines may be selected in the call sequence on the left to display their instance and local variables.

Overall, these three mechanisms, breakpoints, single stepping and variable inspection, provide, in conjunction with the object bench interactions, all the necessary debugging facilities in a manner that is easy to use, easy to learn and easy to remember. Students typically have no problems, after being told or having read about these features, making effective use of them almost immediately.

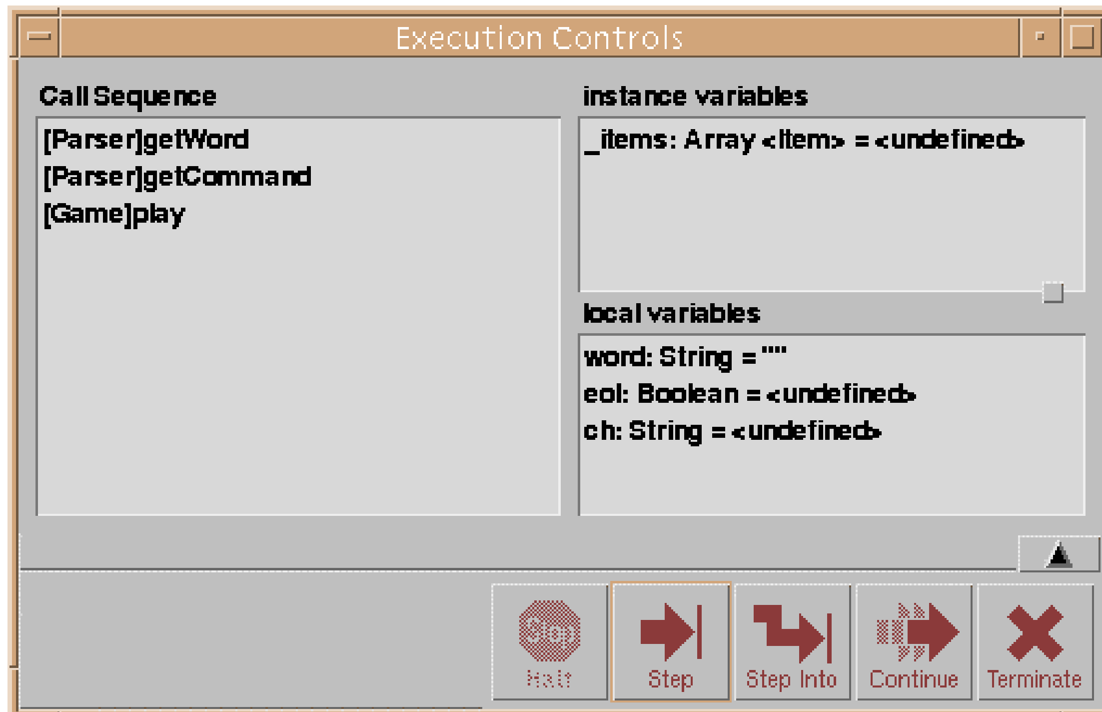

Figure 7.19: Display of call sequence and variables

7.9 Browsing class libraries

Using class libraries is an important part of modern programming, and reuse of library classes is a fundamental software development technique. To be effective, a course based on an object-oriented language must integrate the use of libraries at an early stage in the curriculum [Tewari 1994]. It has frequently been observed that introducing the practice of reuse into existing organisations is very difficult. Many programmers rather reimplement classes than reuse existing ones. Three characteristics have been identified as prerequisites for the acceptance of reuse:

- the programming language must support a mechanism for reuse of existing code,
- the environment must ensure that the effort of finding an appropriate class to be reused is smaller than writing it again, and
- a culture must be established in which the programmer views the reuse of class libraries as an intrinsic part of normal programming.

The first point has been addressed by object-oriented languages. A substantial part of the hope that object-oriented languages may improve the quality of software being developed has to do with the possibility of greater reuse. The other two points, however, have initially been neglected.

The second point – finding classes – can be addressed with better tools. A good class browser is needed that allows a programmer to search and browse libraries of existing

classes easily. The third point – reuse culture – is best addressed by teachers. If reuse is conveyed as normal part of programming from the very beginning, the difficulties associated with requiring an attitude shift later on are avoided.

For a teaching context this means that a good, easy to use browser is an important part of the environment.

The Blue browser can be opened by selecting the "Library Browser" command from the "Tools" menu. The main browser window is shown in Figure 7.20.

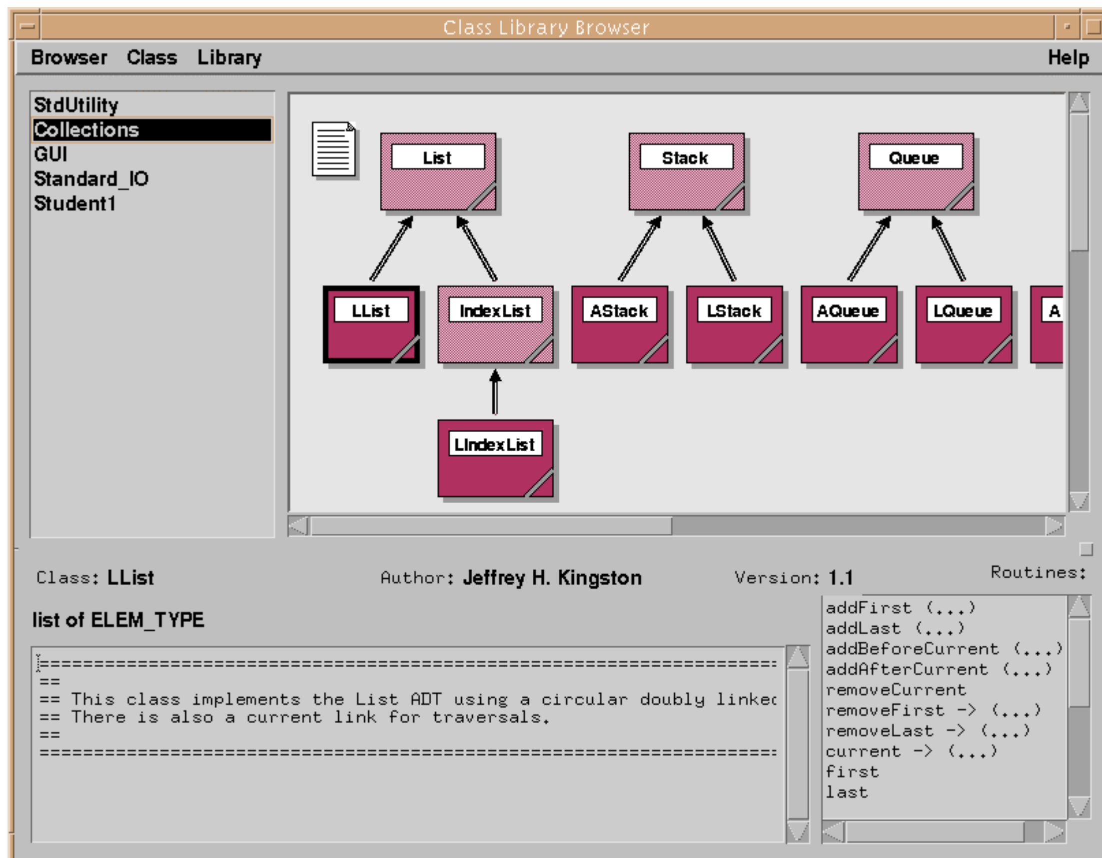

Figure 7.20: The Blue library browser

The browser uses a pane based design with three panes. Pane based designs have successfully been used in several other systems, e.g. in Smalltalk-80 [Goldberg 1984] and the Dylan environment [Dumas 1995]. At the top left a list of library sections is shown. Each section groups related classes together. On the top right a graphical representation of the currently selected section is shown in the same manner in which Blue projects are presented. In the bottom pane detailed information about the currently selected class is displayed. This includes the class comment and a list of interface functions. The bottom pane is updated each time the user clicks on a class icon to display the details of the selected class. The class icon may also be opened (with a double-click) in the same manner as classes in user projects are opened to display its full interface or (depending on permission) its implementation.

7.9.1 Browsing

When browsing the library, a user first examines the list of sections in the library and selects a section which most likely includes classes for the task at hand. The section is then presented using the same graphical display used for a project.

It has often been argued that a browser should enable the user to find related classes. Relevant relationships are super- and subclasses, used classes and clients, and other classes with related functionality. Some existing browsers provide functionality to find some of those related classes.

The graphical display used in Blue provides all of this information in an easily understandable manner. Inheritance and uses relationships are immediately obvious, and even semantically related classes which have no technical relation can be visually associated by placing them close to another class in the same section. If, for example, a programmer looks for a list class, the classes "Stack" and "Queue" (and other collection classes) will also be visible on screen, encouraging the programmer to also examine and consider those classes for use.

When a suitable class is found, the programmer can select the class and invoke the "Use class in project" command from the "Class" menu. The class will then appear in the current project, identified as a library class by its colour. Library classes in a project can be opened, read, called and used like other classes in the project. They cannot be edited, though.

The concept of browsing classes is significantly different from browsing in a Smalltalk system. The first noticeable difference is the graphical presentation. While the Smalltalk browser displays a single class tree as a nested list, the Blue browser uses several independent library sections and graphics for representation. Dividing the library into sections makes it easier to find relevant classes and to switch between related classes. The graphical display visualises relationships. In Smalltalk, inheritance relations are displayed through the list nesting structure (by indenting subclasses under their superclass). Client relationships, however, are not displayed. It is, in fact, not possible in Smalltalk to extract client information from a class definition, since Smalltalk code is not statically typed. Call relationships between classes are only known at runtime.

The main difference, however, between browsing in Smalltalk and in Blue is the clear distinction between the current project and the libraries in Blue. In Smalltalk the browser presents a unified view of the "universe" which includes all user-defined and library classes. This unification, while consistent and powerful, has been named as one of the most serious problems for beginners in dealing with Smalltalk environments (see chapter 4). The distinction between the current project and the libraries in Blue, and the process of explicitly importing library classes into the project, greatly simplify the appearance of the project under development and ease students' understanding of the environment.

7.9.2 Searching

Another important browser function is *searching*. The browser provides a "Find class" command that displays a find dialogue. In this dialogue details about the search can be specified. This includes the search string, the search context and the set of classes to be searched.

The search context gives the user the option to search in the whole class source, in the interface of a class, in comments only, in routine names only, by author name or any combination of these. The set of classes to be searched can be restricted to the current section, the whole library or all classes in all known libraries.

Search results are then displayed in a list. Selecting a result in that list displays the class by opening its section and highlighting the class.

7.9.3 Integration

The browser closely cooperates with the compiler. This integration provides the basis for better functionality than would be achievable with a stand-alone browser.

The information used and displayed in the browser is not taken from the classes' source text, but from the symbol table produced during compilation. The compiler stores the symbol tables and makes them available to the browser. This ensures that the information shown in the browser is syntactically correct. It also provides a simple mechanism to allow different levels of access to library classes. Creators of libraries can, for example, decide whether they want to allow read access to the source text of the library or not. This can then be implemented by using the access restrictions of the underlying operating system. Because the source is not otherwise used for browsing, a library can be searched, browsed and used with access to only the code and symbol tables. An author can easily deny access to the source and still provide a fully functional library. One example where this may be used is in the context of student assignments. A teacher can provide a sample solution in a library which students can execute to examine the behaviour of the program, but without having the ability to view the source code.

On the other hand, the author of a library may wish to allow access to the source. In the library of collection classes in the current Blue system, for instance, all source code is readable by users. They serve as useful examples of well written Blue code that students might find interesting to look at and study.

7.9.4 Documentation

A class library should not only provide the code of useful classes, but also its documentation. Users need to know the signatures and the semantics of library routines.

Many systems separate these issues and use two different tools to provide these two sides of the same coin. Borland C++, for instance, uses the standard C++ include

mechanism for the inclusion of library classes and provides documentation for those libraries through a separate help system. This mechanism is error prone, duplicates effort and is not easily extensible. It is error prone because it can easily happen (and does regularly happen) that the implementation and the documentation get out of sync. Library classes might be changed or added, while the documentation is not updated. It duplicates effort because the creator of a library class is forced to write explanations twice: once in the class interface and once in the help system. (In the case of C++ it is, in fact, often written three times: in the implementation file as well.) And it is not extensible because it is not always equally easy to add library classes and help files. In the Borland system, for instance, user classes may be added into the library collection for general inclusion, but help information cannot be added, effectively leaving the user-defined library classes without documentation.

Java tries to overcome some of these problems with a similar construct as Blue: interface comments. Certain comments in the Java classes are especially marked as interface documentation and can be extracted by a separate tool (called “Javadoc”). This creates documentation for a class. This mechanism avoids the duplication of effort: documentation is only written once (in the class source) and automatically extracted for outside documentation. It does not, however, solve the problem of outdated or missing documentation. It may still happen that a class’s code is updated while its documentation is not, leaving the user with incorrect information. It also does not guarantee the availability of documentation.

In Blue, the documentation is written only once in the class’s source code. It is then stored together with the class in its symbol table. The class’s code and its documentation cannot be separated. This guarantees the accuracy and availability of the documentation (as far as it has been written by the author). The class browser doubles as a help system that can display detailed information about interfaces and semantics of classes and routines.

7.9.5 Additional functionality

The description so far reflects the current implementation that derives its design from an early prototype. Currently work is under way to implement an improved library browser which provides additional functionality to improve the usefulness of class libraries. The new design includes two new areas of functionality: improved *library management* and *library creation*.

The library creation facilities allow users to create their own class libraries easily. A student then can, over time, create a private collection of useful classes and store them in a personal library that may be searched and browsed with the standard browser. Libraries can also be created by work groups for current projects. The creator of a library can specify access restrictions. This allows individual libraries to be kept private or to be published for the whole world to use. The access model used is similar to the three level Unix model that distinguishes the user, a work group and the rest of the world.

The library management facilities let every user configure the browser to offer access to different libraries. In effect, they make libraries known to the browser. By default, each browser will offer access to the Blue standard library and a personal, private user library. Other libraries can then be added to the browser. These are included in a menu for easy access to all known class libraries.

7.9.6 The libraries

As with other parts of the programming environment, the requirements for libraries used for teaching differ from those used in professional software development. Tewari and Gitlin [Tewari 1994] state that "most discussions about the design of object-oriented libraries revolve around the need of professional programmers. However, we should not assume that the needs of students are identical with the needs of experienced software engineers. In fact, there are a number of areas in which the two differ." The primary requirements for a teaching context are ease of use and clarity. This means that, as with the environment as a whole, the libraries used should be specifically designed for a teaching system. If a library is complicated and the student needs to study it for a long time before it can be effectively used, its value is greatly reduced. It will then detract from other concepts the student is investigating. Tewari and Gitlin [Tewari 1994] observe that many libraries are unsuitable for teaching because "many commercial vendors seem fixated on supplying every possible option imaginable. The extensive features add to the difficulty of using the library without necessarily adding to its pedagogical value."

Blue currently offers four library sections: collection classes, graphical user interface (GUI) classes, input/output classes (such as files), and "utility" classes (such as date and time classes). All of these were designed to offer an interface that is easy to understand and to use.

A discussion of the details of the library design is beyond the scope of this work. The interested reader can download the Blue system from one of the locations listed in Appendix B. The system distribution includes the libraries.

7.10 Group support

Teaching object-orientation in a first year course represents a well known paradigm shift in the language area. A second, much less discussed paradigm shift is slowly being introduced in the teaching area: a shift from algorithm-based courses to software engineering oriented courses.

Traditionally, the introductory programming course had a strong focus on the algorithmic aspect of programming. With the introduction of object-orientation and the recognised importance of a solid software engineering foundation for all programmers, many first year courses are slowly shifting this focus: software engineering topics, such as readability, maintenance, commenting, testing, correctness assurance and modularisation, have found their way into introductory courses. This shift of

focus is not often discussed in the literature. Many courses make this shift quietly and gradually, but most introductory courses make it. One important part of this new software engineering focus is group work.

Working in teams poses a whole range of additional problems to those faced while working alone. While few first year course designers would consider the introduction of a formal software process into the introductory course, many use group work to give the students exposure to the problems faced when working in a team, and to teach some of the technical and organisational solutions to group work problems. Some of the advantages of object-orientation are, in fact, much more clearly visible when working in a group (e.g. information hiding and the interface/implementation separation). Teachers increasingly regard teaching about group work as being more important than teaching individual wizardry [Goldberg 1995a]. All this leads to the conclusion that a teaching environment should provide facilities to support group work.

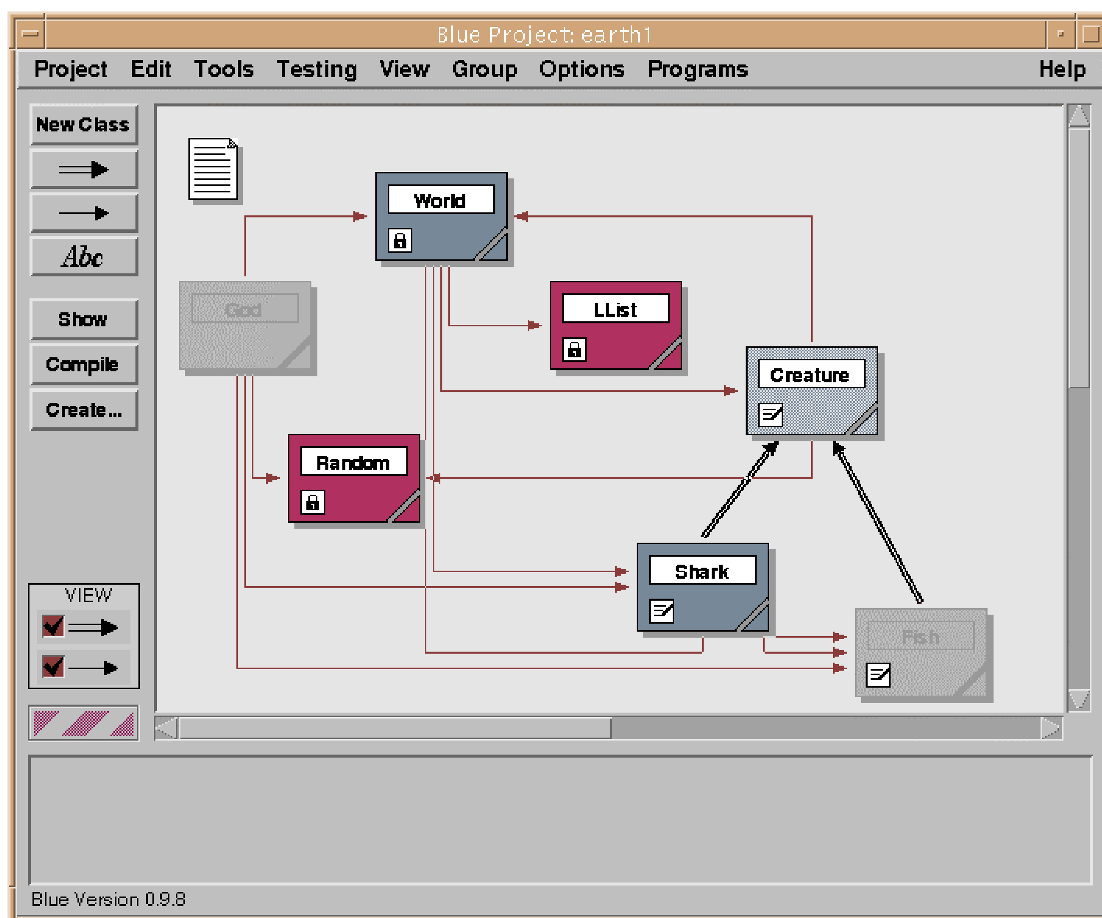

Figure 7.21: Class display in a group project

The Blue environment offers file locking mechanisms to ensure correct group support. A "Group" menu contains the relevant commands.

One of the aims was to keep the group work mechanism unobtrusive while it was not needed. It should not be necessary for users to perform explicit lock or check

out/check in operations while working alone on a project. To achieve this, each project can be declared a "group project" or a "single user project". A new project is a single user project by default.

When single user projects are opened by a user, a lock is automatically created that prevents other users from editing or executing the same project. When a second user opens the project, a message is displayed informing him/her that it is already in use by another user. This message also contains the name of the user who owns the lock, so that he/she may be contacted if necessary.

If several users want to work on one project simultaneously, the project can be declared a group project. This activates a more sophisticated locking system on a class basis.

The locking system used is explicit. Users have to declare their intention to edit or execute a class before they can do so. In group projects, the locking state of each class is visually displayed (Figure 7.21).

The state of each class is displayed using colour and symbols in the lower left corner of a class.

When a user opens a group project, all classes initially appear grey. This indicates that the user cannot currently edit or execute these classes. To do so, the user must execute one of the following operations (available in the "Group" menu):

- *Take Class for Editing*. This command places the class under the user's control. He/she may edit and execute the class, other users cannot edit or execute it. They can, however, open the class for reading. A class that is "taken" is marked with an edit symbol. Automatic communication exists between all instances of the environment currently displaying the same project. The edit symbol appears on the class in every user's display to indicate that the class is currently being edited by someone. For the user who has taken the class, it is also presented in normal colour. Thus, if a user sees a normal coloured class with the edit symbol, the class can be edited. If a grey class has the edit symbol it is being edited by another user (and cannot be taken by anyone else).
- *Lock Class*. This command locks the class to prevent all editing. The class is then shown in normal colour with a lock symbol and the user can execute the class. This is useful because a user, in order to execute a class, must have taken or locked the class to be called *and all classes it depends on* (we cannot allow a supplier class to be edited or recompiled by one user while another user tries to execute it). In a situation where one user edits and tests a class *A*, a second user edits and tests a class *B* and both classes use a third class *C*, the class *C* can be locked to ensure that both users can execute it, but nobody can change it at the same time. This implies that more than one user can lock a class at any time. A class locked by another user appears grey and with the lock symbol. It may then be locked by other users as well. For each user who holds a lock the class appears in normal colour, indicating that it can be used. (As a general rule, the normal coloured – non-grey – subset of the project is available for use.) The lock symbol will be displayed on the class until the last user releases the lock.

This mechanism allows easy-to-use, secure access to projects by multiple users at the same time.

Another approach, even easier to use, has proven unworkable: implicit locking. It would have been nice to avoid the necessity to explicitly "take" and "lock" classes, and to let the environment handle the synchronisation automatically (as it does on the project level for single user projects). As long as we only consider editing, this is possible. Opening an editor could lock the class and prevent other users from editing it at the same time (although it might lead to annoying situations: if a users edits a class, closes the editor in order to test the changes and finds a problem, another user might in the meantime have opened the class so that the first user cannot continue the coding task). If we also consider execution, the automatic model breaks down. Since a client class is not being edited, it is not locked, and nothing would prevent another user from editing and recompiling a class while another user executes it. The idea of making the code files private (so that every user has an own copy of the code to allow other users to recompile while the first user can still execute) was discarded. This method would have lead to inconsistencies: if one user edits a class and another user still has an old copy of code available, then the source does not match the code being executed. Trying to debug a program in that situation becomes impossible.

The mechanism chosen – explicit locking with implicit communication – seeks an acceptable compromise: the system is still reasonably easy to use and, at the same time, provides enough flexibility to allow simultaneous execution as well as editing.

7.11 Summary

In this chapter we have discussed the Blue environment. While many different aspects were discussed, three ideas emerged as the important principles underlying the Blue design: simplicity, visualisation and interaction.

Simplicity refers to the ease-of-use aspect of the environment: while it provides sophisticated functionality in many respects, it still presents an interface that is easy to use and understand for beginners. It avoids the cognitive overload often present in advanced programming environments. The interface is consistent so that users can, after having understood some basic principles, explore the environment on their own by experimentation and guessing. It has been shown that users can guess possible actions in a software system which are close to previous experiences, while it is difficult to suggest new possibilities through the interface [Dumas 1995]. For Blue, this means that beginners need some amount of initial instruction, but that the consistent interface then enables them to expand their knowledge through self directed learning very quickly. The combination of sophisticated functionality, clarity and simplicity presents a unique opportunity for use in a teaching situation and is not found in any other environment.

Visualisation is used to represent the application structure. This represents a shift from a pure coding environment to a software development environment that includes the design process. Visualising class structures and objects provide an invaluable help

in teaching and learning about object-oriented principles. It helps students to understand classes, class relationships and class design. Making the structure visible encourages more conscious creation and discussion of designs.

Interaction refers to the ability to create and execute objects interactively. It is also represented in debugging tools that allow the interactive examination of object or machine state. Direct interaction is a powerful tool that helps in understanding of programming principles (e.g. the class–object relationship), debugging, testing, and many other aspects of the software development process. It helps students to more quickly obtain a deeper insight into the principles behind the software development task.

While the Blue environment was developed explicitly as a teaching environment, and the techniques employed are beneficial in this situation, the use of these techniques is not restricted to teaching. A similar combination of tools – visualisation and interaction – together with other sophisticated development tools may well be valuable in professional software development environments. While the visualisation aspect has been used in various CASE tools, the direct interaction facility is typically not supported. The possibilities of integrating these techniques into professional environments seem worth further investigation.

8 Implementation

In this chapter we present a very brief overview of the Blue implementation architecture and briefly discuss selected topics of interest. Space does not permit a more extensive discussion of implementation concerns.

8.1 Implementation environment

Platform

Blue was originally implemented on Solaris 2.5 in C++, using the GNU compiler g++, X Windows and Motif. It was ported to Linux and currently runs stably on numerous different Linux systems. A port to Microsoft Windows is almost complete, although it has not been fully debugged and is not yet suitable for distribution. The source currently consists of about 85,000 lines of code (including comments and white space).

Programmers

The implementation was completed over the last three years by the author and a second programmer, employed by the Basser Department of Computer Science at the University of Sydney. The programmer, John Bignucolo, was responsible for the implementation of the compiler; the author was responsible for the remaining parts of the system.

Language and programming guidelines

Blue is implemented in a strictly object-oriented style in C++. All code is part of a class (with the exception of a *main* function and one-line Motif callback routines) with strict adherence to information hiding. A programming standard was used that excludes many of C++'s more questionable features. Public variables, for instance,

are treated as read-only throughout, and only reference (pointer) variables are used to refer to objects. Immediate variables are used for scalar types only.

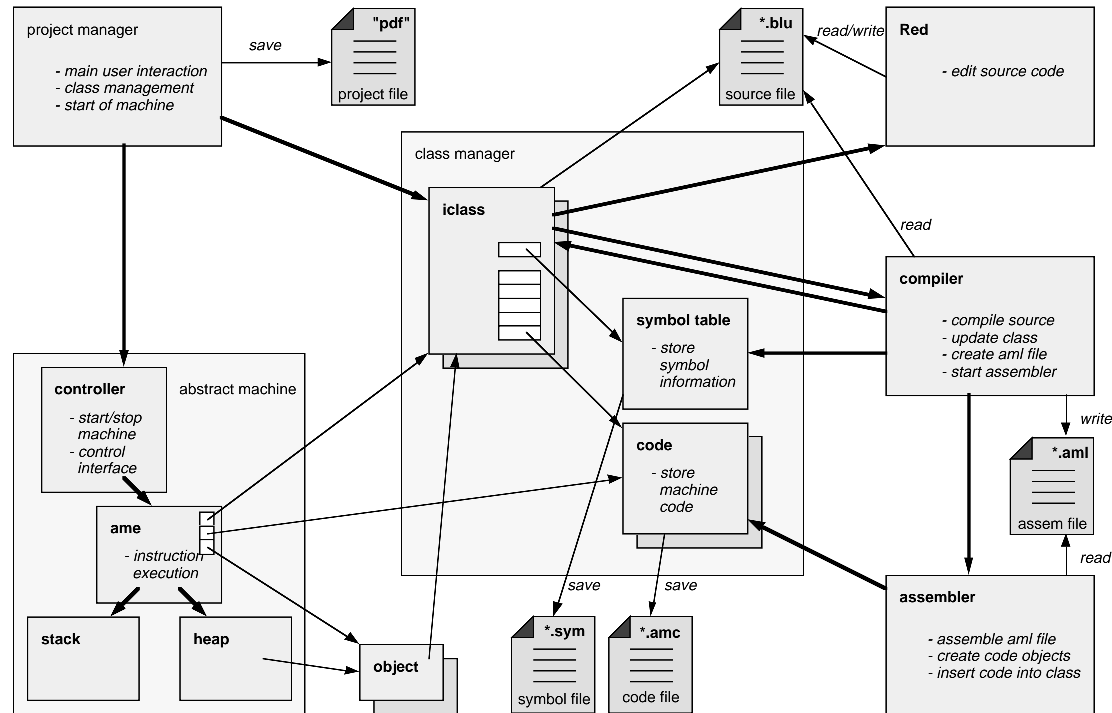

Figure 8.1: Application structure overview

8.2 Software architecture

Figure 8.1 shows a high level overview of most of the system. The central element is a *class manager* which manages all classes in the current project. Each class held in the class manager potentially refers to its *symbol table* and a set of *code objects*. (These only exist when the class has been compiled.) One code object exists for each routine in the class. Each class also has an associated source file (marked with a *blu* suffix) which is created and initialised to a default class skeleton at class creation time.

At the top left of the diagram is the *project manager*. It is responsible for the display of the main window and the handling of the main user interaction. It also maintains the *project description file* (pdf) which stores high level information about a project.

When a class is opened for editing, the class invokes the editor (named *Red*) and passes to it the relevant source file. Each time the source is saved the editor signals back to the class, which in turn notifies the project manager to update the class's dependency arrows.

When a user performs a *compile* operation on a class, the class sends its source file to the compiler. The compiler produces a symbol table, which is handed back to the class for later reference, and an assembly language file (marked with an *aml*-suffix, indicating an *abstract machine language* file). It then invokes the assembler, which creates code objects for each routine in the assembly language file and enters them into the class. The assembly file can then be removed.

The symbol table and the code objects are saved to disk for permanent storage, but are also held in memory for immediate use. When a project is opened, the code and symbol tables for all compiled classes are loaded into main memory. The symbol table and code files are marked with *sym* and *amc* (*abstract machine code*) suffixes, respectively.

The abstract machine consists of four major parts: the *abstract machine engine* (*AME*), which performs the instruction executions, a stack, a heap and a controller. The controller provides the machine's interface to the project manager (and thus, indirectly, to the user). Its interface includes operations such as starting, stopping or single stepping the machine.

The heap holds a set of objects, each of which has a reference to its class. The machine engine has several registers, such as the *current object register* (*CO*), the *current class register* (*CC*) and the *current routine register* (*CR*). The current instruction pointer (often referred to as *program counter*) is stored in the code object itself. When an interface routine of an object is called, the machine pushes the current register set onto the stack, retrieves the reference to the new class from the object to be called and uses the routine number as an index to retrieve a reference to the required code object from the class. (The reference to the object and the routine number are provided as parameters to the call instruction). The reference to the class and the code object are stored in registers to optimise future references. On return from the routine the previous state is restored from the stack.

The Blue system executes two threads concurrently. The *main thread* handles all user interaction, class management, editing and compilation. The *machine thread* executes the virtual machine and thus, indirectly, Blue user code. The threads communicate via semaphores. The dual thread architecture allows the user interface to remain responsive while user programs execute. It enables the user, for instance, to stop the virtual machine via a user command if it seems to be in an infinite loop.

The implementation overview in Figure 8.1 omits the class browser and the debugger. The class browser is a modified copy of the project manager and a class manager. It has a slightly different interface and different functionality, but shares much of the internal structures and code with the original. The debugger has links to most other major components of the system.

8.3 The compiler

The Blue compiler is a recursive descent compiler, implemented using ANTLR [Parr 1995]. ANTLR is a parser generator developed at Purdue University which generates well structured and readable C++ code from a *LL(k)* grammar (with arbitrary *k*; in case of Blue *k*=2). We chose ANTLR over other parser generators because it generates a recursive descent parser, rather than an LR parser. Recursive descent parsing allows better control over the production of error messages. Since the quality of error messages is an extremely important issue for the overall useability of beginners' systems, this was the determining factor in the decision for ANTLR.

The compiler is used for various levels of analysis of the source code in Blue. When the editor window for a class is closed, the class passes its source to the compiler to perform a *dependency analysis*. In this operation, only information about the class's name, its superclass and supplier classes is extracted. This information is reported back to the class to be represented in the project view. When a class is compiled, a *full compilation* is started initially. This includes a check whether supplier classes have been compiled. If not, those are compiled first. If, during this process, a circular dependency is detected, the compilation is aborted, and a *header compilation* and an *interface compilation* are performed first. The header compilation collects information about generic parameters of a class. The interface compilation analyses the full class interface and builds its symbol table. Both operations are applied to all uncompiled classes in the project. Finally, a full compilation is initiated again as requested in the initial operation. This time circular dependencies are not a problem, since all interfaces are known and supplier classes do not need to be compiled first.

This algorithm ensures that the most common case where no circular dependencies exist executes efficiently (since only one compilation pass is needed), while the more complex case is still handled correctly. (Note that even in projects with circular dependencies the most common case is the one that does not involve interface compilation, since rarely is the whole project compiled. Due to incremental compilation each run of the compiler typically involves only a few classes.)

When the source of a class is changed, the class is marked as *uncompiled*. Whether its clients also need recompilation depends on the interface of the changed class: if the change affected only the internals, clients are still correct. If, however, the interface was changed, all clients have to be recompiled as well. To deal with this situation, each class retains its previous symbol table before a compilation is started. After a successful compilation, it compares the old symbol table to the new one to determine whether the interface was changed. If it was, all clients are notified and are themselves marked as uncompiled. The comparison of the interfaces stored in the two symbol tables is executed at a logical level: names, number and types of routines and parameters are compared, not the source text. As a result an interface will correctly be recognised as unchanged even if, for example, the layout of a routine header was changed.

8.4 The abstract machine

The Blue abstract machine is a very simple register/stack machine executing a custom-built machine language. The machine has eight registers. Machine instructions are designed specifically to support Blue language operations. While many instructions are simple (such as register load instructions) some perform complex operations. An example are common string operations. String concatenation, for instance, is supported by a machine instruction. This avoids duplicating tasks in the Blue compiler that have already been solved in the underlying implementation language.

The stack is used to save the machine state when routines are executed and for passing and returning parameters. All objects are stored in the heap. The machine includes a straightforward *mark-and-sweep* garbage collector. A detailed description of the machine and its instruction set can be found in [Fetzer 1998].

9 Experience

The Blue system was used for first year teaching by the Basser Department of Computer Science, University of Sydney, for the first time in 1997. It was used with beginning students for two semesters, after which the students switched to C++. Blue was used again for the introductory course in 1998 and, at the time of writing, is still being used.

9.1 Runtime environment

The first year course at the Basser Department has about 750 students. The system was run on a dual processor Sun Sparc Ultra machine (running Solaris 2.5) with about 120 X-terminals connected. Roughly 60 of these terminals were used to run Blue at any time (the other terminals were used by second and third year students for other tasks).

The Blue system used about 1.2 Megabytes of main memory for its first instantiation and an additional 150 Kilobytes for each additional copy running simultaneously. (The memory increase for additional copies is small since much of the code used is in shared libraries.) 60 concurrent copies were comfortably supported by the existing hardware in both memory requirements and processor speed.

9.2 Feedback

Unfortunately, no formal evaluation of the effect and effectiveness of the Blue system on students and their learning experience has yet been undertaken. It was recognised very early on that such an evaluation is highly desirable, but the required work is outside the scope of this PhD thesis. A formal evaluation is planned for the near future (see *Future Work*, chapter 10).

We have, however, collected feedback from students in the form end-of-semester surveys and from teaching staff (lecturers and tutors) through informal interviews. The feedback shows that students felt generally comfortable in working with the Blue environment after a very short time and that they felt in control most of the time. Teaching staff uniformly reported clear advantages of the environment and the language when compared to the system used previously at the Department (Pascal and a Unix command line interface). They reported a very high degree of satisfaction with both the language and environment design.

Because of the informality of the methods used for gathering feedback, no clear conclusions can be drawn. But the anecdotal evidence points to a positive effect of the system on students and teachers and gives an encouraging starting point for a more formal analysis in the future.

9.3 Stability

One of the criticisms often brought forward against the use of new, little-tested systems is the probable presence of bugs in the application. This problem was also present in the Blue system.

The first semester of use ran unexpectedly smoothly with only few problems which could be fixed very quickly. Students were developing relatively simple applications, and the implementation seemed well tested and debugged for this level of use. In the second semester students started developing more complex applications and uncovered more bugs. They wrote many programs using combinations of constructs that had not been fully tested (such as complex combinations of circular dependencies and genericity). While there were an increased number of bugs, we could usually fix problems quickly or at least offer a work-around, so that no student was held up very long by system problems. There was, however, a significant list of known bugs for several months. The problems lasted until the middle of the first semester 1998. By that time a version of the system was produced that runs very reliably in the present circumstances.

9.4 Problems

In the Department of Computer Science at the University of Sydney, the Blue system was generally considered clearly superior, compared to competing systems, in terms of its design and functionality. After a careful comparison of different options Blue was chosen as the first year teaching language. There are, however, problems inherent in the use of Blue that need to be recognised and addressed.

Stability

Stability, discussed above, is an important factor in all software and often a potential problem in new and locally developed systems. The fact that Blue had not previously

been used in a real-world teaching situation was a matter of concern. We tried to address this problem by providing a bug-reporting system and technical help line (via email and newsgroups) through which we dealt with arising questions and tried to remove problems quickly.

Textbook

One of the major problems was the lack of a textbook for Blue. A good textbook is a great help in running a course, both for the students and instructors. We instead recommended a small set of books to students and provided extensive course notes. The recommended books included language independent books on object-oriented programming as well as books using other languages for examples. For one of those books we reimplemented all examples, which were originally written in C++, in Blue and made them available to the students. The course notes included selected parts from other books, chapters written locally on specific issues and a Blue language and environment manual.

For the second year in which Blue was used at Sydney University, the teaching staff has written an extensive introductory text book for use with Blue. This book covers general introductory topics as well as Blue language issues.

Popularity

One argument against the use of Blue was that it was not widely known. Since students have not heard of it before, it was argued, it will be less popular and less motivating than, say, C++ or Java. Our experience does not confirm this fear. We told students very clearly and very honestly why we chose Blue as the teaching language, namely that we believe that the use of Blue can teach them principles which are later applicable to many languages, and that the understanding of those principles is more important at this stage than the knowledge of specific details of a particular language. Students generally seemed to understand or at least accept that argument, and we were not confronted with demands to use a more “hyped” language.

Availability

One of the aims was to make Blue available to students for working at home on personal computers. Unfortunately, the port to Microsoft Windows, the most commonly used system on home PCs, was not available on time. To somewhat alleviate the problem we provided a CD to students that contained a modified Linux system and the Linux version of Blue.[17] The Linux system could be installed alongside Windows in the same file system. To run Blue, the machine had to be booted in Linux and Blue could then be executed.

[17] The CD and the Linux system on the CD were prepared by Michael Cahill, at the time an honours student at the Basser Department of Computer Science.

Surveys show that about half of the students installed and used this system at home. A considerable number, however, had problems with the installation and some time and extensive help was needed to get the Linux system working on their machines.

Overall, this solution is clearly only second-best, since installing a Linux system invariably causes regular difficulties and requires some skills that not all students have at that stage. Providing an easy-to-install Windows system still remains the goal, and work on the Windows port continues.

10 Future Work

The Blue project is an ongoing project that is still under development and will be for some time. The work still to be done falls into five different categories:

- Completion and enhancements
- Evaluation
- Support materials
- Porting to other platforms
- Follow-on projects

10.1 Completion and enhancements

Work listed in this section includes functionality that was included in the original Blue design, but was so far not implemented (completion), and additions or modifications to the originally specified design, typically a result of experience with the current system (enhancements).

10.1.1 Completion

Group work

The group work mechanism described in section 7.10 has not been fully implemented. It represents an important part of the environment for teaching a course with a software engineering focus, since group work is an essential skill to be learned by programmers. The implementation of this part is well under way and will be completed shortly.

Library browser

As described in section 7.9.5, an improved design for the class library browser has been developed. The design extends the browser's functionality to include the creation, management and use of multiple libraries. It allows personal libraries to be created and libraries to be shared between individuals or groups. We expect this extension to have a positive effect on the students' experience, because it takes reuse beyond the idea of reusing standard libraries. It makes it easy for students to build their own collection of reusable classes and thus does not only encourage reuse, but also writing for reuse.

At the time of writing the design of the new library browser has been completed and implementation has started.

GUI library

A graphical user interface library has been developed. This library will allow the creation of Blue programs with graphical interfaces. The library is not essential for the content of the introductory course – graphical interfaces are not necessary for learning the basics of good program development. (In fact, if used too early or too extensively, graphical user interfaces can convey a misleading impression of object-orientation to the student. It is important, when teaching about classes and objects, to demonstrate their applicability to real world problems outside the computing domain.) GUIs can, however, have a very positive effect on student motivation. Students like to work with graphical interfaces, and the fact that they can produce programs with professional looking front ends creates a sense of excitement. This factor should be exploited by the teacher, and a good class library should include classes for graphical interfaces.

The GUI library has been defined and implemented. It is currently in the debugging stage.

10.1.2 Enhancements of functionality

Additional libraries

The class libraries currently supplied cover only the most essential functionality (collections, input/output, GUIs and some utility classes, such as *time*, *date* and *random*). Blue's functionality and flexibility could be significantly enhanced by adding additional class libraries. Libraries for some form of persistence (e.g. with operations for saving objects to disk) and libraries for database connectivity would be especially useful. Currently no such libraries exist.

Heap display

A display of objects currently existing on the heap would extend the visualisation techniques used from the static (class) model to the dynamic (object) model. Displaying those objects and their relationships would help in debugging and could increase the understanding of object-oriented concepts.

Inspection of objects currently provides some of the functionality necessary for visualising the object structure. Objects can be examined and the references between objects can be uncovered. This technique is, however, only aimed at examining single objects or a small number of them. As the number of objects increases, the view provided by object inspection (which consists of a number of seemingly unrelated dialogue boxes) fails more and more to present an easily understandable picture of the relationships between objects.

A heap display would show all objects currently in existence, or a subset of them. References between objects would be displayed, and the object state would be indicated (either permanently on screen or made available on demand). It would not be feasible to always display all objects, since thousands might exist at any one moment. The display of several thousand objects would, in most cases, not be helpful. A heap display tool would need to offer mechanisms for convenient management of selective views. This might begin by initially visualising only one or more of the objects available on the object bench. From there, the tool could offer commands to extend single references (i.e. to display the object pointed to by a reference in the already visible object) or to extend all references of a selected object. This could be done with selectable depth: a command might be offered to extend all references from a given object to the level of, say, three. This would recursively extend references until all objects which are reachable from the starting point through three or less indirections are visible. If this extension touches the same object twice (because it is referenced by two other objects) it must, of course, be shown as a single object with two references to it.

One of the problems of such a display is layout. It is not obvious how an appropriate object layout could be managed automatically. The viewer would probably have to offer a combination of manual and automatic layout which tries to guess a sensible position for objects on display, but lets the user rearrange the graph. (This is similar to the technique used for the class display.) It could offer several options for default layout schemes, such as layouts for trees or lists, which the user could easily enable and switch between. This would help if the user expects a certain data structure and wants to give hints to the system as to a potentially good layout scheme.

Another important factor would be the connection of different representations of data in different display formats. If, for instance, the debugger window is open, displaying the stack, the heap display is visible displaying some objects, and some object inspection windows are on screen, the same object (or references to it) might well be visible on screen several times in these different system views. McDonald and Stuetzle [McDonald 1990] have proposed a technique to help understand the connections between these different representations, termed "painting" of multiple views. This approach provides commands to highlight one or more objects in one view (with the ability to provide different simultaneous highlights in different colour) and the highlight would be visible in all different views presenting the same object. This technique might prove helpful for advanced debugging or experimentation exercises.

Heap display should also be integrated with the debugger. Heap visualisation in conjunction with single stepping can create the effect of a simple algorithm animation. The user could, for example, step through the creation code for a linked

list and, if the heap display is updated after each instruction (which would probably not be feasible during normal execution, but is possible in single stepping mode) could watch the list being created in the heap.

Preliminary studies, including the design and implementation of a prototype have been undertaken, but currently heap display does not exist in Blue and no work is presently being done towards its development.

GUI interface builder

An interesting extension, once the GUI library is completed and released, is the addition of an interface builder using the library. Several such builders exist for other languages, and they are typically useful in creating graphical user interfaces quickly and easily. The interface builder should be interactive and combine graphical and textual editing capabilities. When interfaces are graphically built, the Blue code needed to construct that interface should be created (and optionally displayed) on the fly, and the user should have the option of editing the code directly and see the resulting changes to the interface.

10.1.3 Enhancements of implementation

Much of the implementation could be improved in hindsight. As is always the case, some of the problems became apparent only after a large part of relevant implementation had already been completed, and it was not always feasible to rewrite all of the affected code to arrive at the optimal implementation. Undoubtedly a substantial improvement could be made through a complete rewrite of the system. The Smalltalk system developed at Xerox PARC, for example, was completely reimplemented several times, and Krasner [Krasner 1984] reports a clear improvement in system quality through this approach. The problem is workload: we do not have the time and people needed for such a task.

We can, however, identify many separate areas in the system that would especially benefit from reimplementation. While it is not helpful to discuss all of them in detail, one of them – our experience with memory management and the lack of garbage collection – is worth noting.

Garbage collection

Since Blue is implemented in C++, and C++ does not provide automatic garbage collection, considerable effort had to be spent dealing with memory management. The code needed to keep track of use of some implementation objects and to deallocate their memory is some of the most complex code in the system. Type information for Blue class and variable types, for instance, is held for some time, and typically regenerated at each recompilation. The information generated by previous compilations can typically be discarded at some stage, but the time when deletion of this information is safe is sometimes hard to determine. (This is, by the way, one of the disadvantages of tight integration: since the same information might be used by different tools, it is not always easy to determine when the last possible access to an implementation object has taken place.) Several bugs were found in early versions of

the system, and it took a significant amount of time of testing and maintenance to arrive at a stable implementation.

Several garbage collectors are now available for C++ systems. While we have not evaluated any of them in detail, reports indicate that they work reliably and with acceptable efficiency. Using one of those garbage collectors for the Blue implementation has the potential to increase the reliability of the system while removing some of the most complex code.

10.2 Evaluation

One of the most important and most interesting projects following on from the development of the Blue system is a formal evaluation of its effects and effectiveness. We have claimed numerous times that one or the other aspect of the Blue system has a positive effect on the learning process for students. We have also claimed that the Blue system in total provides advantages compared to other available systems. So far nothing more than guessing (but at least: educated guessing based on the experience of several people) and, at best, anecdotal evidence exists to substantiate our claims.

An evaluation is needed on at least two levels. A system useability study could uncover useability problems and suggest improvements to the user interface. An example of such a study for a software development environment, the Dylan environment, is described in [Dumas 1995].

The second level involves an evaluation of teaching and learning aspects with the Blue system. We need to find out whether our goals, that students gain a better understanding of object-oriented programming principles, were really achieved. It would be necessary to analyse student knowledge and perceptions and to compare the use of Blue to the use of other languages and environments. It would also be interesting to investigate how students cope with the next language after having learnt Blue, and to compare these results with students who started with a different language.

The work needed to undertake such a study requires careful planning and preparation and should include input from education experts.

10.3 Support materials

Support materials play an important role in the acceptance of programming languages. They can make the difference between a useful system and an unusable system.

While a set of support documents exists for Blue, others are still missing. So far, a language specification, an environment manual, an editor manual, a web site and a set of example programs exist. The web site contains references to these documents, software and example programs for downloading, technical help documents and some other information.

Among the resources still missing are a complete textbook that uses Blue and the Blue teaching approach, a library manual and a better introductory tutorial to using Blue. Each of these documents can improve the usefulness of the Blue system as a whole.

10.4 Porting to other platforms

As mentioned above, Blue currently runs only on Solaris and Linux systems. Among the possible ports to other platforms, Microsoft Windows is the most urgent. Windows is used in many teaching institutions and by the majority of students at home. The port to Windows (which includes Windows 95, 98 and NT) has been almost completed and the system is currently in the debugging phase.

A port to Macintosh systems is also desirable, although it targets a far smaller user group. Porting to Mac OS has not yet been started, and it is not yet clear whether we will have the resources to attempt such a port.

10.5 Follow-on projects

Two new projects have been started as a result of this work. Both draw heavily on experience gained with the Blue project and try to make some of Blue's benefits available to a wider user group.

BlueJ

We have started to develop a Blue-like system for Java. The Blue language was replaced with Java as the supported language, while the environment remains to a large extent unchanged. The result will be an easy-to-use teaching system for Java.

For this project, we are reimplementing the complete system in Java. Using Java as the implementation language will hopefully, because of its platform independence, remove the problems with porting the system to various platforms.

This project is supported by a local Australian company, Softway Pty Ltd, and Sun Microsystems.

A professional distributed environment

Many characteristics of Blue related to its integration, visualisation and interactivity can be useful for professional software developers as well as students. We are planning to develop a professional software development environment that incorporates many of Blue's tools and techniques. This environment will also explicitly support the development of distributed applications, and execute itself as a distributed system.

10.6 Summary

While a large amount of work has been completed, a never-ending stream of future work continues to evolve. With a software system of this type, one cannot expect the work to be completely finished at any stage. Only two types of software exist: software that is not used and software that is continuously maintained. Since the world around us changes, all programs that continue to be used will have to adapt to different situations and expectations.

The Blue language definition is stable and will not be changed in the near future. The environment is operational and has been used with great success. It remains to be seen whether the Blue system or aspects of it will survive in the real world, on their own or as part of a future system.

11 Conclusion

Teaching object-orientation has become an increasingly important topic over the last few years. Object-orientation has been accepted as an important computing paradigm, and almost universal agreement has been reached that it should be taught at universities in the undergraduate curriculum. Increasingly, the trend is towards teaching an object-oriented language in the first year.

Experience, however, shows that teaching object-oriented programming to first year students remains difficult. Although object-oriented principles are now reasonably well understood, and several important programming language designers more or less agree in their definition and interpretation of object-oriented concepts, their application and their importance, the *teaching* of those principles still causes teachers more problems than did previous programming paradigms.

In this thesis, we have identified the most significant problems in this area and proposed solutions. The major contributions of this work are listed below.

- We have identified and discussed requirements for an object-oriented teaching language. These requirements are formulated in general terms and may be used to evaluate any given language in terms of its usefulness for introductory teaching. They can also be used as design guidelines in the development of a new language for this purpose.
- We have identified requirements for an object-oriented teaching environment. The importance of environments was discussed and a basis established on which to evaluate environments for their suitability in teaching situations.
- The most important object-oriented languages were analysed and evaluated against the requirements. We have clearly identified strong and weak aspects of each particular language and assessed their suitability for our purpose. A similar analysis was done for some environments.
- We have designed a language that meets the requirements identified earlier. The language Blue has the potential to significantly ease the teaching of object-orientation to beginners through a unique mix of constructs which emphasise

clarity, simplicity and conceptual understandability. Many constructs which are overly complex or redundant in existing languages were simplified while still being similar enough to other imperative object-oriented languages to serve as an ideal stepping stone to more professionally oriented systems in later years. The language design was driven strongly by a focus on its use as a dedicated teaching language, avoiding many problems encountered with languages currently used in teaching situations. The language is fully defined. Most interesting language aspects and design decisions are discussed here; a more formal language reference manual is available (see Appendix B for download locations).

- An integrated environment for teaching was designed which meets the stated requirements. This environment has many features which make it more suitable for teaching than other environments currently available. Most notable is its sophisticated support for program development coupled with a simple appearance and ease-of-use. The environment includes most of the important software development tools, including a project manager, an editor, a compiler, a debugger and a class browser. The interface is highly visual and interactive.
- For the integrated environment, we have designed an object interaction technique that allows direct manipulation of objects. The user can interactively create object instances, interact with these instances (e.g. invoke operations on those objects) and interactively test individual classes. This mechanism allows incremental development of applications. Object interaction is not usually available in environments for strongly typed languages, and this facility is particularly useful in a teaching situation.
- The complete system, including the integrated environment and the compiler, has been implemented and extensively tested. It has been used in a large computer science department for first year teaching. This practical application has confirmed our claims about advantages of our design and proves the feasibility of the techniques discussed.

The Blue system as a whole demonstrates how the integrated development of a language and a carefully chosen set of support tools can lead to an environment that offers much better teaching support than existing systems on the market. The tight integration of available tools and the close relationship between the environment and the language enable the provision of functionality not previously available to teachers and students.

A large number of students (about 1500 students over two years) have learnt to program using the Blue system for their first year. Although no formal evaluation has been undertaken it has become evident through feedback from students and teachers that many of our goals were achieved.

The design and implementation of Blue has lead to the identification and development of numerous techniques and constructs generally useful for programming environments, independent from its application as a teaching system. Those ideas will flow into new projects, some of which are already under development.

Appendix A: Related documents

The following documents were produced as part of this PhD thesis:

Papers

Requirements for a first Year Object-Oriented Teaching Language, M. Kölling, B. Koch and J. Rosenberg, in ACM SIGCSE Bulletin, ACM, Nashville, 173-177, March 1995.

Blue - A Language for Teaching Object-Oriented Programming, M. Kölling and J. Rosenberg, in Proceedings of 27th SIGCSE Technical Symposium on Computer Science Education, ACM, Philadelphia, Pennsylvania, 190-194, March 1996.

An Object-Oriented Program Development Environment for the first Programming Course, M. Kölling and J. Rosenberg, in Proceedings of 27th SIGCSE Technical Symposium on Computer Science Education, ACM, Philadelphia, Pennsylvania, 83-87, March 1996.

Testing Object-Oriented Programs: Making it Simple, J. Rosenberg and M. Kölling, in Proceedings of 28th SIGCSE Technical Symposium on Computer Science Education, ACM, San Jose, Calif., 77-81, February 1997a.

I/O Considered Harmful (At least for the first few weeks), J. Rosenberg and M. Kölling, in Proceedings of the Second Australasian Conference on Computer Science Education, ACM, Melbourne, 216-223, July 1997b.

On Creation, Equality and the Object Model, M. Kölling and J. Rosenberg, School of Computer Science and Software Engineering, Monash University, Technical Report 98/16, July 1998. Conference publication in preparation.

Support for Object-Oriented Testing, J. Rosenberg and M. Kölling, publication in preparation.

Manuals

All manuals are available in PostScript format from the Blue web site (address below).

Blue - Language Specification, Version 1.0, M. Kölling and J. Rosenberg, School of Computer Science and Software Engineering, Monash University, Technical Report TR97-13, 80 pages, November 1997.

The Blue Programming Environment – Reference Manual, M. Kölling, School of Computer Science and Software Engineering, Monash University, to be published as Technical Report, 27 pages, 1998.

Blue – Abstract Machine Language Manual, S. Fetzer, M. Kölling and J. Rosenberg, School of Computer Science and Software Engineering, Monash University, to be published as Technical Report, 54 pages, 1998.

The Red User Manual, M. Kölling, Monash University 1997, http://www.csse.monash.edu.au/~mik/red-manual.

Web Pages

The Blue Page - Teaching Object Oriented Programming, M. Kölling, http://www.sd.monash.edu.au/blue/

Blue Resources, M. Kölling, http://www.sd.monash.edu.au/blue/resources/

The Red User Manual, M. Kölling, http://www.csse.monash.edu.au/~mik/red-manual

Software

Blue - An integrated software development environment for the Blue language, written in C++ for X Windows/Motif, approx. 90,000 lines. Available for Solaris 2.5 and Linux. Port to Windows 95 and Windows NT is well advanced and expected before the end of 1998. Available free of charge. See Appendix B for download locations.

Red - A stand-alone version of the text editor used in the Blue system. Available for Solaris 2.5 and Linux free of charge. See Appendix B for download locations.

packbp - A Unix utility to archive and compress Blue projects. Used to transfer projects from one system to another. Included in the full Blue distribution package.

Appendix B: Download Locations

The software and manuals developed as part of this work can be downloaded over the internet. The following list gives WWW and ftp addresses. All text files are in PostScript format.

package	address and directory	notes
The Blue system	ftp.sd.monash.edu.au *directory:* /pub/mik/blue/	file names include operating system and version number; please list directory to see current version
Red (binaries)	ftp.sd.monash.edu.au *directory:* /pub/mik/red/binaries/	file names include operating system and version number; please list directory to see current version
Red (source)	ftp.sd.monash.edu.au *directory:* /pub/mik/red/	file names include operating system and version number; please list directory to see current version
Language Specification	ftp.sd.monash.edu.au *directory:* /pub/mik/blue/doc/	file name is *spec-XXX.ps*, where XXX is a version number
Environment Reference Manual	ftp.sd.monash.edu.au *directory:* /pub/mik/blue/doc/	file name is env-man-*XX.ps*, where XX is a version number
The Red User Manual	http://www.sd.monash.edu.au/~mik/red-manual/	

Appendix C: CD contents

The CD attached to this thesis contains copies of the Blue system, version 0.9.7. The top level directory structure is as follows:

README.txt	A text (ASCII) file explaining the CD contents
Linux	A directory containing Blue executables for Linux
Solaris	A directory containing Blue executables for Solaris 2.5/2.6
Windows	A system to be installed on a Microsoft Windows system. Note that this is **not** a native Windows version. Instead, installing this archive will install a modified Linux system within the Windows file system and the Blue version for Linux. The Linux system is modified to use the Windows file system format, so the disk does not need to be partitioned.

The Linux and Solaris versions also contain copies of the stand-alone version of the *Red* text editor.

All directories contain README files with detailed installation instructions. To install any of the versions on your system, read the README file in the sub-directory for your operating system and follow the instructions.

References

[Abelson 1995] Hal Abelson, Kim Bruce, Andy van Dam, Brian Harvey, Allen Tucker and Peter Wegner, *The First-Course Conundrum,* Communications of the ACM, Vol. 38 No. 6, 116-118, June 1995.

[Allen 1998] Robert K. Allen, Kevin Bluff and Annette B. Oppenheim, *Jumping Into Java: Object-Oriented Software Development for the Masses*, in Proceedings of the Third Autralasian Conference on Computer Science Education, ACM, Brisbane, Australia, 165-172, July 1998.

[America 1990] Pierre America and Frank van der Linden, *A Parallel Object-Oriented Language with Inheritance and Subtyping,* SIGPLAN Notices (ECOOP/OOPSLA '90 Proceedings), Vol. 25 No. 10, 161-168, October 1990.

[Apple 1994] Apple, *Dylan™ Interim Reference Manual,* Apple Computer, Inc., June 1994.

[Arnold 1996] Ken Arnold and James Gosling, *The Java™ Programming Language,* Addison Wesley, 1996.

[Baecker 1997] Ron Baecker, Chris DiGiano and Aaron Marcus, *Software Visualization for Debugging,* Communications of the ACM, Vol. 40 No. 4, 45-54, April 1997.

[Beck 1989] Kent Beck and Ward Cunningham, *A Laboratory For Object-Oriented Thinking*, in OOPSLA '89 Conference Proceedings, ACM, New Orleans, Louisiana, 1-6, 1989.

[Berman 1994] Michael Berman, Rick Decker, Dung X. Nguyen, Richard Reid and Eugene Wallingford, *Using C++ in CS1/CS2*, in Proceedings of SIGCSE '94, ACM, 383-384, 1994.

[Biddle 1994] Robert Biddle and Ewan Tempero, *Teaching C++ - Experience at Victoria University of Wellington,* University of Wellington, Technical Report CS-TR-94/18, 1994.

[Blaschek 1989] Günther Blaschek, Gustav Pomberger and Alois Tritzinger, *A Comparison of Object-Oriented Programming Languages,* Structured Programming, Vol. 10 No. 4, 1989.

[Böhm 1997] Michael Böhm, Jürgen Freytag, Bernd Owsnicki-Klewe, Guido Pfeiffer and Jörg Raasch, *Objektorientierung in der Informatikausbildung auf der Basis von Smalltalk,* Informatik Spektrum, Vol. 20 No. 6, 335-343, December 1997.

[Booch 1986] Grady Booch, *Object-Oriented Development,* IEEE Transactions on Software Engineering, Vol. SE-12 No. 2, 211-221, February 1986.

[Cardelli 1985] Luca Cardelli and Peter Wegner, *On Understanding Types, Data Abstraction, and Polymorphism,* ACM Computing Surveys, Vol. 17 No. 4, 471-522, December 1985.

[Cashman 1991] Mark Cashman, *The Benefits of Enumerated Types in Modula-2,* SIGPLAN Notices, Vol. 26 No. 2, 35-39, February 1991.

[Clark 1998] David Clark, Cara MacNish and Gordon F. Royle, *Java as a teaching language – opportunities, pitfalls and solutions*, in Proceedings of the Third Autralasian Conference on Computer Science Education, ACM, Brisbane, Australia, July 1998.

[Clarke 1988] Lori A. Clarke, Debra J. Richardson and Steven J. Zeil, *TEAM: A Support Environment for Testing, Evaluation, and Analysis*, in Proceedings of the ACM SIGSOFT/SIGPLAN SE Symposium, SIGSOFT SE Notes, 153-162, 1988.

[D'Souza 1995] Desmond D'Souza, *Effective C++ learning and teaching,* Journal of Object-Oriented Programming, Vol. 8 No. 6, 73-76, 1995.

[Decker 1993] Rick Decker and Stuart Hirshfield, *Top-Down Teaching: Object-Oriented Programming in CS 1*, in Proceedings of SIGCSE '93, ACM, 270-273, 1993.

[Decker 1994] Rick Decker and Stuart Hirshfield, *The Top 10 Reasons Why Object-Oriented Programming Can't Be Taught in CS 1*, in Proceedings of SIGCSE '94, ACM, 51-55, 1994.

[Delft 1989] A.J.E. van Delft, *Comments on Oberon,* SIGPLAN Notices, Vol. 24 No. 3, 23-30, March 1989.

[DoD 1983] DoD, *Reference Manual for the Ada Programming Language,* United States Department of Defence, ANSI standard Ada, 1983.

[Dumas 1995] Joseph Dumas and Paige Parsons, *Discovering the Way Programmers think about New Programming Environments,* Communications of the ACM, Vol. 38 No. 6, 45-56, June 1995.

[Ellis 1990] Margaret A. Ellis and Bjarne Stroustrup, *The Annotated C++ Reference Manual,* Addison-Wesley, 1990.

[Evered 1991] Mark Evered, J. L. Keedy, Gisela Menger and Axel Schmolitzky, *How Well Do Inheritance Mechanisms Support Inheritance Concepts?*, in Lecture Notes in Computer Science, 1204, Springer-Verlag, 252-266, 1991.

[Evered 1995] Mark Evered, Michael Kölling and Axel Schmolitzky, *A Flexible Object Invocation Language based on Object-Oriented Language Definition,* The Computer Journal, Vol. 38 No. 3, 181-191, 1995.

[Fetzer 1998] Stefanie Fetzer, Michael Kölling and John Rosenberg, *Blue – Abstract Machine Language Manual,* School of Computer Science and Software Engineering, Monash University, Technical Report TR 98/20, August 1998.

[Fry 1997] Christopher Fry, *Programming on an Already Full Brain,* Communications of the ACM, Vol. 40 No. 4, 55-64, April 1997.

[GI 1997] Gesellschaft für Informatik (GI), *Themenheft zur Ausbildung in objektorientierter Programmierung,* Informatik Spektrum, Vol. 20 No. 6, December 1997.

[Goedicke 1997] Michael Goedicke, *Java in der Programmierausbildung: Konzept und erste Erfahrungen,* Informatik Spektrum, Vol. 20 No. 6, 357-363, December 1997.

[Gold 1991] Eric Gold and Mary Beth Rosson, *Portia: An Instance-Centered Environment for Smalltalk*, in OOPSLA 91 Conference Proceedings, ACM, 62-74, 1991.

[Goldberg 1984] Adele Goldberg, *Smalltalk-80: The Interactive Programming Environment,* Addison-Wesley, 1984.

[Goldberg 1995a] Adele Goldberg, *What Should We Teach?*, in Proceedings of OOPSLA '95, ACM, 30-37, 1995.

[Goldberg 1995b] Adele Goldberg, *Why Smalltalk?,* Communications of the ACM, Vol. 38 No. 10, 105-107, October 1995.

[Goldberg 1989] Adele Goldberg and David Robson, *Smalltalk-80: The Language,* Addison-Wesley, 1989.

[Grogono 1991] Peter Grogono, *Issues in the Design of an Object-Oriented Programming Language,* Structured Programming, Vol. 12 No. 1, 1-15, 1991.

[Grogono 1993] Peter Grogono, *Equality and assignment in object oriented languages*, in EastEurOOPe'93, November 1993.

[Grogono 1994a] Peter Grogono, *PC-Dee: Syntax and Semantics,* Dept. of Computer Science, Concordia University, Report, February 1994.

[Grogono 1994b] Peter Grogono and Patrice Chalin, *Copying, Sharing, and Aliasing,* in *Object-Oriented Technology for Database and Software Systems, Proceedings of the Colloquium on Object Orientation in Databases and Software Engineering (COODBSE'94),* V.S. Alagar and R. Missaoui, Editor. 1994, World Scientific: River Edge (NJ).

[Haarslev 1990] Volker Haarslev and Ralf Möller, *A Framework for Visualizing Object-Oriented Systems*, in ECOOP/OOPSLA 90 Conference Proceedings, ACM, 237-244, 1990.

[Henderson 1993] Robert Henderson and Benjamin Zorn, *A Comparison of Object-Oriented Programming in Four Modern Languages,* University of Colorado at Boulder, Technical Report CU-CS-641-93, February 1993.

[Holt 1994] Richard C. Holt, *Introducing Undergraduates to Object Orientation Using the Turing Language,* SIGCSE Bulletin, Vol. 25 No. 3, 324-328, 1994.

[Holt 1988] Richard C. Holt and J. R. Cordy, *The Turing Programming Language,* Communications of the ACM, Vol. 31 No. 12, 1410-1423, December 1988.

[Joyner 1996] Ian Joyner, *C++?? - A Critique of C++ and Programming Language Trends of the 1990s, 3rd edition,* Unisys - ACUS, Report, 1996.

[Knudsen 1993] J. Lindskov Knudsen, M. Lofgren, O. Lehrmann Madsen, B. Magnusson and (Eds.), *Object-Oriented Environments - The Mjølner Approach,* Prentice Hall, 1993.

[Knudsen 1988] Jørgen Lindskov Knudsen and Ole Lehrmann Madsen, *Teaching Object-Oriented Programming is more than teaching Object-Oriented Programming Languages*, in Proceedings of ECOOP '88, Springer-Verlag, Oslo, Norway, 21-40, August 1988.

[Kölling 1997b] Michael Kölling, *The Red User Manual,* Monash University, 1997, http://www.sd.monash.edu.au/~mik/red-manual.

[Kölling 1998b] Michael Kölling, *The Blue Programming Environment – Reference Manual,* School of Computer Science and Software Engineering, Monash University, Technical Report TR 98/19, August 1998.

[Kölling 1995] Michael Kölling, Bett Koch and John Rosenberg, *Requirements for a First Year Object-Oriented Teaching Language*, in ACM SIGCSE Bulletin, ACM, Nashville, 173-177, March 1995.

[Kölling 1996a] Michael Kölling and John Rosenberg, *Blue - A Language for Teaching Object-Oriented Programming*, in Proceedings of 27th SIGCSE Technical Symposium on Computer Science Education, ACM, Philadelphia, Pennsylvania, 190-194, March 1996.

[Kölling 1996b] Michael Kölling and John Rosenberg, *An Object-Oriented Program Development Environment for the First Programming Course*, in Proceedings of 27th SIGCSE Technical Symposium on Computer Science Education, ACM, Philadelphia, Pennsylvania, 83-87, March 1996.

[Kölling 1997a] Michael Kölling and John Rosenberg, *Blue - Language Specification, Version 1.0,* School of Computer Science and Software Engineering, Monash University, Technical Report TR97-13, November 1997.

[Kölling 1998a] Michael Kölling and John Rosenberg, *On Creation, Equality and the Object Model,* School of Computer Science and Software Engineering, Monash University, Technical Report 98/16, July 1998.

[Krasner 1984] Glenn Krasner, *Smalltalk-80: bits of history, words of advice,* Addison-Wesley, 1984.

[Kristensen 1996] Bent Bruun Kristensen and Kasper Østerbye, *A Conceptual Perspective on the Comparison of Object-Oriented Programming Languages,* SIGPLAN Notices, Vol. 31 No. 2, 42-54, 1996.

[LaLonde 1990] Wilf LaLonde and John Pugh, *Smalltalk as the first programming language: the Carleton experience,* Journal of Object-Oriented Programming, Vol. 3 No. 4, 60-65, November/December 1990.

[LaLonde 1991] Wilf LaLonde and John Pugh, *Subclassing ≠ subtyping ≠ Is-a,* Journal of Object-Oriented Programming, Vol. 3 No. 5, 57-62, January 1991.

[Lins 1990] C. Lins, *Programming Without Enumerations in Oberon,* SIGPLAN Notices, Vol. 25 No. 7, 19-27, July 1990.

[Liskov 1992] Barabara Liskov, *A History of CLU,* Laboratory for Computer Science, MIT, Technical Report, April 1992.

[Liskov 1995] B. Liskov, *Theta Reference Manual,* MIT Laboratory for Computer Science, Cambridge, MA, Programming Methodology Group Memo 88, 1995.

[Liskov 1981] Barbara Liskov, Russel Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Roby Scheifler and Alan Snyder, *CLU Reference Manual,* Lecture Notes in Computer Science Vol. 114, Springer Verlag, 1981.

[MacLennan 1982] B. J. MacLennan, *Values and Objects in Programming Languages,* SIGPLAN Notices, Vol. 17 No. 12, 70-79, December 1982.

[Madsen 1993] Ole Lehrmann Madsen, Birger Møller-Pedersen and Kristen Nygaard, *Object-Oriented Programming in the Beta Programming Language,* Addison-Wesley, 1993.

[Mazaitis 1993] Dorothy Mazaitis, *The Object-Oriented Paradigm in the Undergraduate Curriculum: A Survey of Implementations and Issues,* SIGCSE Bulletin, Vol. 25 No. 3, 58-64, September 1993.

[McDonald 1990] John A. McDonald and Werner Stuetzle, *Painting multiple views of complex objects*, in ECOOP/OOPSLA 90 Conference Proceedings, ACM, 245-257, 1990.

[Meyer 1988] Bertrand Meyer, *Object-oriented Software Construction,* Prentice Hall, 1988.

[Meyer 1992] Bertrand Meyer, *Eiffel: The Language,* Prentice Hall, 1992.

[Meyer 1993] Bertrand Meyer, *What is an object-oriented environment?,* Journal of Object-Oriented Programming, Vol. 6 No. 4, 75-81, July/August 1993.

[Nørmark 1995] Kurt Nørmark, *An Evaluation of Eiffel as the first Object-oriented Programming Language in the CS Curriculum,* Aalborg University, Denmark, May 1995, ftp://ftp.iesd.auc.dk/pub/projects/normark/eiffel-eval.ps.

[Notkin 1988] David Notkin, *The Relationship Between Software Development Environments and the Software Process*, in Proc. of the ACM SIGSOFT/SIGPLAN SE Symposium, SIGSOFT SE Notes, ACM, 107-109, 1988.

[Omohundro 1991] Stephen M. Omohundro, *The Sather Language,* ICSI, Berkeley, Part of the Sather system distribution, 1991.

[Pancake 1995] Cherri M. Pancake, *The Promise and the Cost of Object Technology: A Five-Year Forecast,* Communications of the ACM, Vol. 38 No. 10, 33-49, October 1995.

[Parr 1995] T.J. Parr and R.W. Quong, *ANTLR: A Predicated-LL(k) Parser Generator,* Software - Practice and Experience, Vol. 25 No. 7, 789-810, July 1995.

[Petre 1995] Marian Petre, *Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming,* Communications of the ACM, Vol. 38 No. 6, 33-44, June 1995.

[Pohl 1988] Ira Pohl and Daniel Edelson, *A to Z: C Language Shortcomings,* Computer Languages, Vol. 13 No. 2, 51-64, July 1988.

[Roberts 1995] Eric S. Roberts, *Loop Exits and Structured Programming: Reopening the Debate*, in ACM SIGCSE Bulletin, ACM, Nashville, Tennessee, 268-272, March 1995.

[Rosenberg 1997b] John Rosenberg and Michael Kölling, *I/O Considered Harmful (At least for the first few weeks)*, in Proceedings of the Second Australasian Conference on Computer Science Education, ACM, Melbourne, 216-223, July 1997.

[Rosenberg 1997a] John Rosenberg and Michael Kölling, *Testing Object-Oriented Programs: Making it Simple*, in Proceedings of 28th SIGCSE Technical Symposium on Computer Science Education, ACM, San Jose, Calif., 77-81, February 1997.

[Ryba 1997] Michael Ryba and Stefan Leboch, *Eiffel in Lehre und Forschung - Erfahrungen und Perspektiven,* Informatik Spektrum, Vol. 20 No. 6, 344-349, December 1997.

[Sakkinen 1991] Markku Sakkinen, *Another defence of enumerated types,* SIGPLAN Notices, Vol. 26 No. 8, 37-41, August 1991.

[Sakkinen 1992] Markku Sakkinen, *The darker side of C++ revisited,* Structured Programming, Vol. 13 No. 4, 155-177, 1992.

[Schmidt 1991] Heinz W. Schmidt and Stephen M. Omohundro, *CLOS, Eiffel and Sather - A Comparison,* ICSI, Berkeley, Technical Report TR-91-047, 1991.

[Skublics 1991] Suzanne Skublics and Paul White, *Teaching Smalltalk as a First Programming Language*, in Proceedings of SIGCSE '91, ACM, 231-234, 1991.

[Smith 1997] Randall B. Smith, Mario Wolczko and David Ungar, *From Kansas into Oz - Collaborative Debugging When a Shared World Breaks,* Communications of the ACM, Vol. 40 No. 4, 72-78, April 1997.

[Stroustrup 1991] Bjarne Stroustrup, *The C++ Programming Language,* second edition, Addison Wesley, 1991.

[Stroustrup 1994] Bjarne Stroustrup, *The Design and Evolution of C++,* Addison-Wesley, 1994.

[Temte 1991] M.C. Temte, *Let's Begin Introducing the Object-Oriented Paradigm*, in Proceedings of SIGCSE '91, ACM, 73-77, 1991.

[Tewari 1994] Rajiv Tewari and David Gitlin, *On Object-Oriented Libraries in the Undergraduate Curriculum: Importance and Effectiveness*, in Proceedings of the 25th ACM SIGCSE Symposium, ACM, 319-323, 1994.

[Ungar 1997] David Ungar, Henry Liebermann and Christopher Fry, *Debugging and the Experience of Immediacy,* Communications of the ACM, Vol. 40 No. 4, 39-43, April 1997.

[Ungar 1987] David Ungar and Randall B. Smith, *Self: The Power of Simplicity,* SIGPLAN Notices (OOPSLA '87 Proceedings), Vol. 22 No. 12, 227-242, December 1987.

[Winograd 1995] Terry Winograd, *From Programming Environments to Environments for Designing,* Communications of the ACM, Vol. 38 No. 6, 65-74, June 1995.

[Wirth 1988a] N. Wirth, *The Programming Language Oberon,* Software - Practice and Experience, Vol. 18 No. 7, 671-690, July 1988.