http://researchspace.auckland.ac.nz

ResearchSpace@Auckland

Copyright Statement

The digital copy of this thesis is protected by the Copyright Act 1994 (New Zealand).

This thesis may be consulted by you, provided you comply with the provisions of the Act and the following conditions of use:

- Any use you make of these documents or images must be for research or private study purposes only, and you may not make them available to any other person.
- Authors control the copyright of their thesis. You will recognise the author’s right to be identified as the author of this thesis, and due acknowledgement will be made to the author where appropriate.
- You will obtain the author's permission before publishing any material from their thesis.

To request permissions please use the Feedback form on our webpage. http://researchspace.auckland.ac.nz/feedback

General copyright and disclaimer

In addition to the above conditions, authors give their consent for the digital copy of their work to be used subject to the conditions specified on the Library Thesis Consent Form.
Enhancements to an
Object-Oriented Programming Language

Warwick Bruce Mugridge

Submitted in fulfilment of the requirements for the degree of

Doctor of Philosophy

University of Auckland    January 1990
Abstract

The objective of this thesis has been to explore the value and limitations of Class, an object-oriented programming language, in order to further the development of the language.

The pivot for this thesis is WallBrace, a code-checking system. The development of the WallBrace system is the basis of a critique of Class, and leads to a number of language extensions being proposed. An important aim in this work has been the careful integration of these enhancements with the rest of the language, avoiding unnecessary additions.

A number of functional and object-oriented extensions to the language are proposed. Discrimination functions, which may be higher-order and polymorphic, add considerable functional power. Generic classes allow for abstract data types, such as sets and lists, to be defined within the language.

The forms interface proposed will greatly enhance the quality of user interfaces to Class programs. An external interface will allow Class programs to communicate with files, databases, and specialist user-interface programs, such as for plan entry.
Acknowledgements

I would like to thank John Hosking and John Hamer for many enjoyable discussions over the last five years, in which we have strived for a practical and elegant programming language. Special thanks go to John Hosking, as my supervisor, who has always been most encouraging and supportive.

Thanks are also due to: the staff of the Department of Computer Science for a friendly working environment; Haris Dechapunya at BRANZ ("It all depends on WallBrace!") for his enthusiasm and support; Joe Ten Broeke at BRANZ for providing his knowledge of wall bracing; BRANZ, the University Grants Committee, and the University of Auckland for financial support.

Finally, I would like to thank my parents for their encouragement, and Jackie and Anna for their encouragement, support, and patience while I have been distracted from my life with them.
## Contents

1. **Introduction**  
   1.1 Introduction ................................................................. 1  
   1.2 Programming Language “Paradigms” ........................................ 2  
      1.2.1 Functional Languages .................................................. 2  
      1.2.2 Object-Oriented Languages .......................................... 3  
   1.3 Chapter Organisation of the Thesis ....................................... 5  

2. **Introduction to Class**  
   2.1 Introduction ......................................................................... 7  
   2.2 Class as a Single-Assignment Language .................................. 7  
   2.3 Procedural Aspects of Class .................................................. 8  
   2.4 Change and Dependency Management ..................................... 9  
   2.5 Class as an Object-oriented Language ................................... 10  
      2.5.1 Classes ........................................................................... 11  
      2.5.2 Generalisation Lattice and Inheritance ............................. 15  
      2.5.3 Dynamic Classification .................................................. 16  
   2.6 Summary .............................................................................. 18
3. **WallBrace**

3.1 Introduction........................................................................................................21
3.2 The Code.............................................................................................................21
3.3 The Research Project.........................................................................................24
   3.3.1 The Problem and the Need for an Application System............................24
   3.3.2 The Research Project................................................................................25
3.4 Stage One: Development of a Simple Checker.............................................26
   3.4.1 Object-Centred Representation.................................................................26
   3.4.2 A Goal-Directed Approach........................................................................28
   3.4.3 User Interface Issues..................................................................................29
3.5 Stage Two: A General Checker.......................................................................30
   3.5.1 Wind Loadings on Non-Trivial Buildings................................................31
   3.5.2 Plan Entry....................................................................................................31
   3.5.3 Roof Structures.........................................................................................32
   3.5.4 Calculating Loadings..................................................................................33
   3.5.5 Laying Out Bracing Lines & Positioning Diaphragms.............................34
3.6 Class Organisation of WallBrace.....................................................................34
   3.6.1 An Overview...............................................................................................34
   3.6.2 Plan Entry....................................................................................................35
   3.6.3 Calculate Loadings......................................................................................36
   3.6.4 Bracing........................................................................................................36
3.7 Summary............................................................................................................37
4. A Critique of Class

4.1 Introduction ..............................................39

4.2 Single-Assignment Aspects ................................39

4.2.1 Sets and Bags ........................................39

4.2.2 Need for Functions ....................................41

4.3 The User Interface and Procedures ......................43

4.3.1 Limitations of a Textual Interface ....................44

4.3.2 Single Service: Undue Emphasis on the Report ....44

4.3.3 Coherence of Questions ................................45

4.4 Object-Oriented Aspects ................................46

4.4.1 Modularity .............................................47

4.4.2 The “Type Loss” Problem ..............................48

4.5 Classification .............................................49

4.5.1 Multiple Classification .................................49

4.5.2 Classification and Inheritance .........................50

4.5.3 Sequence: Recursive Classes .........................51

4.5.5 Inconsistent Clusters .................................52

4.6 Conclusions ...............................................54

5. Functional Abstraction

5.1 Introduction ..............................................55

5.2 Functional Programming Languages .....................55
5.2.1 Pattern Matching........................................55
5.2.2 Higher-Order Functions ................................56
5.2.3 Polymorphic Functions................................57
5.2.4 Lazy Constructors........................................57

5.3 Object-Oriented Languages.................................58
5.3.1 The "Type Loss" Problem................................58
5.3.2 The "Type Loss" Problem for Functions..............60
5.3.3 The "Contravariance" Problem.........................62

5.4 Adding Functions to Class................................64
5.4.1 Simple Functions.........................................64
5.4.2 Functions within Classes.................................65
5.4.3 Higher-Order and Polymorphic Functions...........67
5.4.4 Lazy Functions............................................68

5.5 Discrimination Functions..................................69
5.5.1 Simple Discrimination Functions.......................69
5.5.2 Discrimination Functions in Classes...................70
5.5.3 Narrowing the Result Type.............................72
5.5.4 A Solution to the Contravariance Problem...........73
5.5.5 A (Partial) Type-Loss Solution.........................74
5.5.6 Pattern Classes...........................................75

5.6 Further Type Loss........................................75
5.6.1 Type Loss................................................76
5.6.2 Type Inference............................................78
6. **Generic Classes**

6.1 Introduction .................................................. 87
6.2 Functional Languages ......................................... 87
6.2.1 Polymorphic Data Types .................................... 87
6.2.2 Abstract Data Types ......................................... 88
6.3 Object-Oriented Languages .................................... 89
6.3.1 Parametric Polymorphism in Classes ....................... 89
6.3.2 Bounded Parametric Polymorphism ......................... 90
6.3.3 Contravariance and Generic Classes ....................... 91
6.4 Adding Generic Classes to Class ............................. 91
6.4.1 A Simple Example ........................................... 91
6.4.2 Subclasses of a Generic Class ............................. 92
6.5 An Extended Example: Lazy Lists ............................ 93
6.5.1 Set/Bag Functions Replaced by List Functions .......... 95
6.6 Bounded Parametric Polymorphism ........................... 96
6.6.1 Sets and Bounded Polymorphism ........................... 96
6.6.2 Inheritance with Bounded Polymorphism .................. 98
6.7 Generators ....................................................... 99
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.8</td>
<td>Generic Classification</td>
<td>100</td>
</tr>
<tr>
<td>6.9</td>
<td>Summary</td>
<td>102</td>
</tr>
<tr>
<td>7.1</td>
<td>Introduction</td>
<td>103</td>
</tr>
<tr>
<td>7.2</td>
<td>Functional Languages and I/O</td>
<td>103</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Streams: Miranda</td>
<td>104</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Procedures Added: Standard ML</td>
<td>104</td>
</tr>
<tr>
<td>7.2.3</td>
<td>Result Continuations: Hope+C</td>
<td>105</td>
</tr>
<tr>
<td>7.3</td>
<td>Providing a Set of Services</td>
<td>105</td>
</tr>
<tr>
<td>7.3.1</td>
<td>A Program as a Set of Services</td>
<td>105</td>
</tr>
<tr>
<td>7.3.2</td>
<td>Syntel</td>
<td>106</td>
</tr>
<tr>
<td>7.3.3</td>
<td>Clausen’s Proposal</td>
<td>107</td>
</tr>
<tr>
<td>7.4</td>
<td>An Extended Forms Interface</td>
<td>108</td>
</tr>
<tr>
<td>7.4.1</td>
<td>Forms and Widgets</td>
<td>108</td>
</tr>
<tr>
<td>7.4.2</td>
<td>Property Values: Inputs and Outputs</td>
<td>110</td>
</tr>
<tr>
<td>7.4.3</td>
<td>Procedures and User Control: Buttons</td>
<td>112</td>
</tr>
<tr>
<td>7.4.4</td>
<td>Classification: Form Elaboration</td>
<td>113</td>
</tr>
<tr>
<td>7.4.5</td>
<td>Components: Subforms</td>
<td>115</td>
</tr>
<tr>
<td>7.4.6</td>
<td>Components and Classification: Tables</td>
<td>116</td>
</tr>
<tr>
<td>7.5</td>
<td>Applications of the Forms Interface</td>
<td>118</td>
</tr>
<tr>
<td>7.5.1</td>
<td>WallThermalResistance</td>
<td>118</td>
</tr>
<tr>
<td>7.5.2</td>
<td>Hypertext</td>
<td>118</td>
</tr>
</tbody>
</table>
7.6 Conclusions ..................................................................................120

7.6.1 Needs of an External Interface .............................................121

7.6.2 Other Needs of an External Interface ....................................121

8. External Interface to Class

8.1 Introduction ...............................................................................123

8.2 External Classes ........................................................................124

8.2.1 Dependency Issues ..............................................................124

8.2.2 Properties ..............................................................................125

8.2.3 Procedures ............................................................................125

8.2.4 Using External Classes ........................................................126

8.2.5 An Example ...........................................................................127

8.2.6 Classification .........................................................................127

8.2.7 Components .........................................................................129

8.3 Variable-length Tables ..............................................................134

8.4 Multiple Inheritance and Reference of External Classes ..........138

8.5 Implementation of the External Interface .................................139

8.5.1 Concurrency and Laziness ....................................................139

8.5.2 Connecting to the External Code .........................................139

8.5.3 Transferring Data and Control ............................................140

8.5.4 Processes in Class .................................................................140

8.5.5 Change ..................................................................................141

8.5.6 Protocol of Interaction ........................................................142
8.5.7 Adding Strictness ................................................................. 142
8.6 Summary .................................................................................. 143

9. Conclusions and Future Work

9.1 Conclusions ............................................................................... 145
9.2 Future Work .............................................................................. 146
  9.2.1 Types .................................................................................. 146
  9.2.2 I/O ...................................................................................... 147
  9.2.3 Encapsulation ..................................................................... 148
  9.2.4 Catalysts for the Further Development of Class ............... 151
Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>A Simple Example</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>A Simple Example Continued</td>
<td>9</td>
</tr>
<tr>
<td>2.3</td>
<td>Running a Program</td>
<td>10</td>
</tr>
<tr>
<td>2.4</td>
<td>Outline of a Program with Classes</td>
<td>12</td>
</tr>
<tr>
<td>2.5</td>
<td>Class Structure Diagram for Components</td>
<td>14</td>
</tr>
<tr>
<td>2.6</td>
<td>Inheritance</td>
<td>14</td>
</tr>
<tr>
<td>2.7</td>
<td>Class Structure Diagram for Generalisation</td>
<td>15</td>
</tr>
<tr>
<td>2.8</td>
<td>Conformance</td>
<td>15</td>
</tr>
<tr>
<td>2.9</td>
<td>Multiple Inheritance</td>
<td>16</td>
</tr>
<tr>
<td>2.10</td>
<td>Classification: BracingElement Extended</td>
<td>17</td>
</tr>
<tr>
<td>2.11</td>
<td>Classification</td>
<td>17</td>
</tr>
<tr>
<td>2.12</td>
<td>Boolean Classification</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.1</td>
<td>Wind Loads on a Building</td>
<td>22</td>
</tr>
<tr>
<td>3.2</td>
<td>Clause 6.3.2.1 of NZS 3604</td>
<td>27</td>
</tr>
<tr>
<td>3.3</td>
<td>An Example Building Plan</td>
<td>31</td>
</tr>
<tr>
<td>3.4</td>
<td>Roof Overlaps</td>
<td>33</td>
</tr>
<tr>
<td>3.5</td>
<td>Clause 6.3.5.2 of NZS 3604</td>
<td>33</td>
</tr>
<tr>
<td>3.6</td>
<td>Clause 6.9.6.3 of NZS 3604</td>
<td>33</td>
</tr>
</tbody>
</table>
Figure 3.7 Overview of Classes in WallBrace .............................................. 35
Figure 3.8 Varied-Storey Building Plan ....................................................... 36

Figure 4.1 Use of the Function any ............................................................... 40
Figure 4.2 The Class Rectangle ................................................................. 40
Figure 4.3 Other Use of Sets ....................................................................... 41
Figure 4.4 Repetitious Code ...................................................................... 42
Figure 4.5 Class Range .............................................................................. 42
Figure 4.6 Using Classes as Functions ....................................................... 43
Figure 4.7 An Example of Interface Control .............................................. 46
Figure 4.8 Wall Class Family .................................................................... 48
Figure 4.9 Classification Type Information is “Lost” ................................ 49
Figure 4.10 Classification in Roof ............................................................... 49
Figure 4.11 Multiple Classification ............................................................. 50
Figure 4.12 Without Classification .............................................................. 50
Figure 4.13 Generalisation in the Roof Class Family ............................. 51
Figure 4.14 Recursive Wall Class Family .................................................. 52
Figure 4.15 A Recursively-Defined Sequence ........................................ 52
Figure 4.16 Inconsistency with Classification ........................................... 53
Figure 4.17 Invalid Multiple Inheritance .................................................. 53
Figure 4.18 Mutually-Exclusive Classes ..................................................... 54
Figure 5.1 Pattern Matching ......................................................... 56
Figure 5.2 Higher-Order Function .................................................. 56
Figure 5.3 Fold: A Polymorphic Function ...................................... 57
Figure 5.4 Using Fold ................................................................. 57
Figure 5.5 Infinite Structures through Laziness .............................. 58
Figure 5.6 Incorrect Code ............................................................ 59
Figure 5.7 Type_case in Trellis/Owl .............................................. 59
Figure 5.8 Multi-Methods in CLOS ............................................... 60
Figure 5.9 Type Loss Problem ...................................................... 60
Figure 5.10 Avoiding “Type Loss” in Eiffel ................................. 61
Figure 5.11 Avoiding “Parameter Loss” ........................................ 62
Figure 5.12 Contravariance Problem in Eiffel ............................... 63
Figure 5.13 Separate Interfaces ..................................................... 64
Figure 5.14 An Example Function ............................................... 65
Figure 5.15 Public Functions ......................................................... 66
Figure 5.16 Using a Public Function .............................................. 66
Figure 5.17 IntList Class Family ................................................... 67
Figure 5.18 Using filter, map, and fold ......................................... 68
Figure 5.19 Use of Lazy Lists ....................................................... 68
Figure 5.20 Strict Form of Function fromTo .................................. 69
Figure 5.21 Classification Type Information Regained .................. 70
Figure 5.22 Discrimination Function for Function last .................. 71
Figure 5.23 A Discrimination Function ........................................ 71
Figure 5.24 Using addList

Figure 5.25 A Discrimination Function

Figure 5.26 Contravariance and Parameter-Loss Solution

Figure 5.27 Consistency of result from equal

Figure 5.28 Defining firmEqual Outside the Classes

Figure 5.29 Using Function move

Figure 5.30 Pattern Matching and Classification

Figure 5.31 Revised IntList Class Family

Figure 5.32 Inheriting filter in the ExtIntList Classes

Figure 5.33 The Relationship between the Classes

Figure 5.34 Mixing a Type with a Super Type

Figure 5.35 Introducing another Family

Figure 5.36 Mixing Types

Figure 5.37 A Problem with "like Current"

Figure 6.1 User-Defined Data Types

Figure 6.2 An Abstract Data Type (Standard ML)

Figure 6.3 Generic Class SET

Figure 6.4 Using the Generic Class SET

Figure 6.5 Bounded Type Variables

Figure 6.6 Constraining the Type

Figure 6.7 Generic Class Example

Figure 6.8 Generic Parameters in a Subclass
Figure 6.9 Generic Class Stack and its Subclass...............................92
Figure 6.10 An Integer Stack....................................................93
Figure 6.11 Lazy Lists .............................................................94
Figure 6.12 Collect and Select....................................................95
Figure 6.13 Collect with Map and Select with Filter......................95
Figure 6.14 Filtering Subclasses................................................95
Figure 6.15 Create from Map.....................................................96
Figure 6.16 Set Family..............................................................97
Figure 6.17 Class Comparable....................................................97
Figure 6.18 Invalid Inheritance ................................................98
Figure 6.19 Valid Inheritance with a Discrimination Function.........98
Figure 6.20 The AskGenerator Class Family.................................99
Figure 6.21 An Example of the use of AskGenerator......................100
Figure 6.22 The Class Relationship............................................100
Figure 6.23 The Generator Family..............................................101
Figure 6.24 AskGenerator2 With Generator................................101
Figure 6.25 Class Tree Written in a Functional Style....................102

Figure 7.1 Synchronising Streams in Miranda...............................104
Figure 7.2 I/O in Standard ML..................................................104
Figure 7.3 An Example Form.....................................................109
Figure 7.4 Multiple Forms on the Screen....................................111
Figure 7.5 An “Error” Property................................................111
Figure 7.6  Form Elaboration .................................................. 113
Figure 7.7  The Roof Class Family (Repeated from Chapter 4) ........ 114
Figure 7.8  Constraints .......................................................... 115
Figure 7.9  Constraints Incorporated ........................................ 116
Figure 7.10  A Table in a Form ............................................... 117
Figure 7.11  WallThermalResistance in the Forms Interface ........... 119

Figure 8.1  An Interface to a Forms-Based User-Interface ............... 126
Figure 8.2  An External Class ................................................. 127
Figure 8.3 (a) Classification: External Classes ............................ 128
Figure 8.3 (b) Classification: Classes ........................................ 128
Figure 8.4  Relationships Between the Classes ............................. 129
Figure 8.5  A Triangle Object which has been Classified ................ 129
Figure 8.6 (a) Component Forms ............................................. 130
Figure 8.6 (b) Component Forms ............................................. 131
Figure 8.7  Component Classes ............................................... 132
Figure 8.8  Relationship between Rows in the Table and Objects in Class 134
Figure 8.9  Class/Interface Structure for the Table ....................... 135
Figure 8.10 (a) Variable-Length Table: Table .............................. 136
Figure 8.10 (b) Variable-Length Table: External Classes ............... 136
Figure 8.10 (c) Variable-Length Table: Row ................................ 137
Figure 8.11  Some Interesting Possibilities ................................. 138
Figure 9.1  Module for Set ....................................................... 149
Figure 9.2 Module for ListFamily ....................................................149
Figure 9.3 Sub-Families of List..........................................................150
Figure 9.4 The GeneratorFamily and a Sub-Family .............................151
Chapter One

Introduction

1.1 Introduction

The main objective of the research described in this thesis has been to explore the value and limitations of *Class*, an object-oriented programming language, and to extend the language. The pivot for this thesis is *WallBrace*, a code-checking system. The development of the *WallBrace* system is the basis of a critique of *Class*, and leads to a number of language extensions being proposed. An important aim in this work has been the careful integration of these enhancements with the rest of the language, avoiding unnecessary additions.

This work is one part of a larger research programme. A series of research projects have been carried out in collaboration with the Building Research Association of New Zealand (BRANZ). These have focused on developing systems which provide assistance in checking buildings for conformance with codes of practice (Hosking et al, 1987a, b, c; Mugridge et al, 1988; Mugridge and Hosking, 1988; Hosking et al, 1989a, b; Hosking, 1989; Mugridge et al, 1989). This work has been carried out with the programming language *Class* (Hamer et al, 1988; Hamer, 1989; Hamer et al, 1989a,b; Hamer, 1990; Mugridge et al, 1987).

The forms interface proposed in Chapter Seven and external interface in Chapter Eight are being developed as a part of this research programme. The work in these chapters will have a major impact on the quality of the user interfaces that can be provided with *Class* programs.

For example, the external interface will allow *Class* programs to communicate with specialist user-interface programs, such as for plan entry (Hosking, 1989). It was also intended to be used with *WallThermalResistance* (Mugridge et al, 1989), another specialist user-interface program; however, it is now clear that the forms interface is general enough that the specialist *WallThermalResistance* program can be discarded.
1.2 Programming Language “Paradigms”

*Class* is based on three programming “paradigms”: procedural, functional and object-oriented. The functional and object-oriented paradigms are introduced briefly here; further details are provided in Chapters Five and Six.

1.2.1 Functional Languages

Hope is representative of the statically-typed pure functional languages (Field and Harrison, 1988; Perry, 1989). It is used in this thesis to illustrate features of functional programming languages which are relevant to the development of *Class*.

Referential Transparency

The distinguishing feature of pure functional languages is that they provide for “referential transparency”. This means that an expression always evaluates to the same value in the same context. Hence functions can have no side effects within a program. Such languages are also called “single-assignment” languages.

One of the aims of the functional approach is to simplify the semantics of programming languages in order to ease the task of automatic and manual manipulation of programs, such as in the derivation of correctness proofs. It has also been hoped that such languages will be particularly suitable for exploiting the potential of parallel hardware. However, a continuing issue for functional languages has been how to deal with change, such as in updating files and interacting with users.

Hughes (1989) argues that modularity is the main advantage of functional languages, allowing program code can be developed in independent pieces, tested in isolation, and reused easily. This requires suitable abstractions, such as higher-order and polymorphic functions; these are introduced into *Class* in Chapter 5.

Laziness

Functional languages may be “strict” (“eager”) or non-strict (“lazy”). A strict functional language, like Standard ML (Wikstrom, 1987), is one in which all the arguments to a function or constructor\(^1\) are completely evaluated before being used. A lazy language, like

\(^1\) A constructor is used to create a data item, such as a tree with two sub-trees.
Lazy ML (Augustsson and Johnsson, 1989), is one in which arguments to functions are only evaluated as they are needed (i.e. on demand). Hope eagerly evaluates function arguments but lazily evaluates constructor arguments. *Class* is a lazy language in which a class is a form of constructor.

**I/O**

Functional languages have some difficulties in resolving the incompatability between the notion of referential transparency within the language, and the program having effects through input and output (I/O) and otherwise dealing with a changing world (Wadler, 1989). The matter of I/O in functional languages is taken up briefly in Chapter Seven.

### 1.2.2 Object-Oriented Languages

The benefits of an object-oriented approach are well argued in Meyer (1989). Object-oriented programming originated in Simula (Dahl and Nygaard, 1966) and has been popularised by Smalltalk (Goldberg and Robson, 1983).

There are three important aspects to object-oriented programming languages:

- the organisation is based primarily on "objects" rather than procedures, and the objects are defined in terms of classes
- encapsulation means that access to an object is limited so that details of the implementation are unavailable
- inheritance provides for code sharing

Typed object-oriented languages add a fourth aspect: classes are types, and the generalisation relationships between them explicitly define type relationships.

**Objects and Classes**

Halbert and O'Brien (1987) point out that the object-oriented approach tends to "... concentrate on the data to be manipulated rather than the code that does the manipulation".

For example, consider an object which represents a wall within a building. The object contains information, such as the length, the height, and the building materials used in its construction. It also contains functions which provide additional information about the wall. For example, on request the wall object will calculate its thermal resistance, based on the thermal resistance and configuration of its component materials.
Encapsulation

The notion of "encapsulation" or "information hiding" comes from abstract data types as provided in Ada (DOD, 1981). An abstract data type consists of a signature, which defines the interface to an object of the type, and an implementation which is unavailable to be manipulated by other parts of a program which use the data type. The advantages of this are two-fold: the implementation can be changed without affecting the use of the data type, and a programmer using a data type can avoid the details of the implementation, reducing the (cognitive) complexity of the software.

Classes as Types

Some languages consider that classes are types which are arranged in an explicit type hierarchy (or lattice when multiple superclasses are permitted). This is the approach in Class, as discussed in Chapter Two. In Trellis/Owl (Schaffert et al, 1986; Halbert and O'Brien, 1987), a class inherits the signature of its superclasses. It also inherits that part of the implementation which has been made visible in the superclass.

On the other hand, Canning et al (1989) argue that signatures should be defined separately from classes and that classes only be used for the inheritance of the implementation.

Inheritance

Inheritance is an important notion in object-oriented languages. In languages based on classes, inheritance is defined within the classes, while in prototype-based languages, objects may "delegate" to other objects (LaLonde, 1986; 1989). As statically-typed languages are the focus of this thesis, prototype languages are not considered further.

Inheritance allows for some common attributes to be made explicit. For example, generalisation makes explicit both those aspects that are common to a group of classes and those aspects that are distinct. Common elements are defined in classes higher in the generalisation lattice, while distinctions are defined in classes lower in the lattice.

There are several points of view about inheritance in class-based programming languages. In Simula, for example, the typing and implementation inheritance relationships are integrated (Dahl and Nygaard, 1966). Similarly, in Trellis/Owl the inheritance of the implementation is secondary to the type relationship (Schaffert et al, 1986); this means that code-sharing from a class cannot occur unless that class is a super-type.

In some languages, the emphasis is on the sharing of code (Johnson and Foote, 1988). This occurs in untyped languages like Smalltalk (Goldberg and Robson, 1983) and CLOS.
It is also the emphasis of languages in which there are separate hierarchies for the inheritance of the signature and the implementation (Canning et al, 1989; LaLonde, 1989).

There has been much discussion about the value of inheritance and the lack of encapsulation with inheritance (Black et al, 1986; LaLonde, 1986; Snyder, 1986; America, 1987; Liskov, 1987; Canning et al, 1989; Lunau, 1989; Sakkinen, 1989; Snyder, 1986). Some object-oriented languages reject inheritance altogether and provide for code sharing in another manner (Lunau, 1989; Raj and Levy, 1989).

1.3 Chapter Organisation of the Thesis

Chapter Two introduces Class, the programming language that has been used in this work. This language blends aspects of three programming paradigms: functional, procedural, and object-oriented.

Chapter Three introduces WallBrace, an application system for helping check a building against the wall bracing requirements of a code of practice. The experience of building this system leads to an assessment of the strengths and weaknesses of Class in the following chapter.

Chapter Four provides a critique of Class in the context of the implementation of WallBrace. While the object-oriented aspect of the language is well developed, the functional aspect is shown to be lacking. The most serious problem with the language is that it is not possible to develop a high-quality user interface to a Class program.

Chapter Five introduces functional abstraction into the language. The most significant addition in this chapter is the discrimination function, a typed form of the multi-methods of CLOS (Moon, 1989). An informal outline of a multi-level type inference system is also introduced.

Chapter Six introduces generic classes. These provide for further abstraction in classes and allow for general-purpose data structures like lists to be defined in the language. Generic classes and discrimination functions are integrated to provide much of the power of other functional and object-oriented languages.

Chapter Seven proposes a forms interface that can be used as a general-purpose user interface for Class programs. This interface allows a user to enter information in fields on a form, make selections between options, press buttons to open or close other forms, and
enter tabular information. Surprisingly sophisticated user interfaces can be developed using the forms system, due to the way in which the basic elements can be combined.

Chapter Eight presents the design of a powerful external interface for Class. This interface allows for the transfer of data and control between a Class program and external code. It is also general enough to handle the transfer of file and database information.

The final chapter concludes the thesis and suggests a number of directions for future work.
Chapter Two

Introduction to Class

2.1 Introduction

*Class Language* is a typed, object-oriented programming language with some functional and procedural features. The language was developed by Hamer, beginning in 1985, and is still evolving (Hamer, 1990). It was originally intended as an "expert systems" language, with an object-oriented approach to representation. The language was originally modelled on backward-chaining systems like Mycin (Buchanan and Shortliffe, 1984) and was influenced by the object-oriented form of Smalltalk (Goldberg and Robson, 1983) and the data base ideas of Smith and Smith (1977).

The motivation for developing *Class* was the development of application for checking conformance of buildings with codes of practice (Mugridge et al, 1987; Hamer et al, 1988; Hamer et al, 1989a, b).

*Class* is introduced here to provide necessary background for later chapters in which the language is used and various extensions are proposed. The language is described in this chapter by example, within the context of the three programming "paradigms": single-assignment (or functional), procedural, and object-oriented. The functional and object-oriented paradigms, offering different approaches to abstraction, are integrated in a single language.

2.2 *Class* as a Single-Assignment Language

*Class* is a single-assignment language. It is intended to model a consistent state of something like a building and its components, such as in programs which check for code conformance.

A simple program consists of a set of typed *instances* (or single-valued variables) with associated expressions (or "rules"). The expression associated with an instance is evaluated when a value is required, such as in the evaluation of another expression.
Part of an example program for calculating loadings on a wall is shown in Fig. 2.1 (B.U. is Bracing Unit, a measure of load and bracing). The predefined types integer, float, boolean, and text (like "string" in Pascal) are used. A conditional expression is used in the expression for the instance minBUREquired (the "i" is read as "else").

```plaintext
instance
   minBUREquired : integer
       := max(70, diaphragmLoad) if internalWall
       | max(round(10*length), diaphragmLoad) if length > 3
       | diaphragmLoad.
   diaphragmLoad : integer
       := max(100, diaLoad) if diaLoad > 0
       | 0.
   diaLoad : integer
       := ask('What is the diaphragm load on this wall? ').
   internalWall : boolean
       := ask('Is ', label, ' an internal wall? ').
   length : float
       := ask('What is the length of the ', label, ' wall? ').
   label : text
       := ask('What is the label on this wall? ').
```

Figure 2.1 A Simple Example

The function ask collects information from the user; i.e. the user provides the function evaluation, based on the prompt supplied as arguments to the function.

The single-assignment part of the language allows for expressions to be defined, but does not provide any need for evaluation to occur. Evaluation is initiated by the procedural component of Class, as introduced in the next section.

2.3 Procedural Aspects of Class

Procedures direct the flow of control, causing the evaluation of instances as needed. There is no procedural assignment; procedures do not specify how to evaluate anything, only what is to be evaluated. Procedural constructs are provided for conditional and iterative execution. The procedures display and summary provide output to the screen and to a summary (or report) file respectively. Thus the single-assignment aspect of the language remains pure; the procedural code remains an outer-layer over the functional part.

The execution of a Class program begins with the procedure main. For example, the first statement of the program in Fig. 2.2 causes the evaluation of the instances minBUREquired and label before displaying the result. The evaluation of minBUREquired in turn leads to the evaluation of internalWall and other instances.
instance
minBUREquired : integer
  := max(70, diaphragmLoad) if internalWall
    | max(round(10*length), diaphragmLoad) if length > 3
    | diaphragmLoad.
diaphragmLoad : integer
  := max(100, diaLoad) if diaLoad > 0
    | 0.
diaLoad : integer
  := ask(’What is the diaphragm load on this wall? ’).
internalWall : boolean
  := ask(’Is ’, label, ’ an internal wall? ’).
length : float
  := ask(’What is the length of the ’, label, ’ wall? ’).
label : text
  := ask(’What is the label for this wall? ’).
when length <= 0.0 do
  display(’Please enter a positive length.’).
  change(length).
done.

procedure main.
  display(minBUREquired, ’ B.U. required on wall: ’,
    label, ’.’).
end main.

Figure 2.2 A Simple Example Continued

When procedures allow for procedural code to be executed under specified circumstances. For example, a when procedure is used in the example shown in Fig. 2.2 for checking the length input. This when procedure checks the condition only once length has a value; the statements are executed if the condition is satisfied, causing the display of an error message and a change of the input instance, forcing the user to be reprompted for a new value.

2.4 Change and Dependency Management

While Class is a single-assignment language, it was found to be useful to allow for inputs to be changed in a running Class program. The provision of change in the system is discussed here because it is an aspect of the system that is important to retain while extending the user interface.

The user may change the value of any previous reply in order to correct input errors or to experiment with different inputs. For example, in a code-checking system the user may find that a building does not meet the requirements of a code of practice. The user may then change the value supplied to an earlier question in order to rectify the conformance problem.
When a change is made, all values which depend on the changed input are retracted in order to ensure consistency. A dependency management system records the dependency relationships between values.

For example, the value of the instance minBUrequired in Fig. 2.2 depends on the values of diaphragmLoad and internalWall if internalWall is true. If the user changes the value of internalWall to false, the value of the instance minBUrequired is recomputed. This may lead to other instances being evaluated, such as length.

It is assumed that with functions, such as ask, the result of the function is dependent on all the arguments of the function. If any argument of a function is affected by a change, the function is re-evaluated. In the case of ask, the user is asked the question again. For example, if the instance label is changed, the user is asked again for values for the properties internalWall and length.

A change of a user-input also affects procedures and when procedures. Any affected procedures and when procedures are re-executed. This is illustrated in Fig. 2.3, in which the program of Fig. 2.2 is executed.

```
CLASS> run
-1- What is the label for this wall? A
-2- Is A an internal wall? y
-3- What is the diaphragm load on this wall? change 2
-2- Is A an internal wall? n
-3- What is the length of the A wall? 12
-4- What is the diaphragm load on this wall? 100
119 B.U. required on wall: A.
CLASS>
```

Figure 2.3 Running a Program

2.5 Class as an Object-oriented Language

Class inherits from the object-oriented paradigm the notions of information-hiding, abstract data types, and multiple inheritance within a class generalisation structure. In addition, it introduces the novel notions of object parameters and multiple, dynamic classification. However, Class is also a single-assignment language and so does not allow state-change, unlike most object-oriented languages.
2.5.1 Classes

A complete Class program consists of the definition of classes, plus the declaration of a number of instances (references to objects) and procedures (methods). Typically, the outermost procedural code will call procedures within (class) instances.

A class consists of a signature and an implementation. The signature consists of the parameters and publics of the class. Parameters are used to pass information to a newly created object of a class from the context in which it was created. Publics are those parts of a class that can be accessed from outside an object of the class. The implementation consists of expressions for public properties, bodies of procedures, and private (non-public) properties and their expressions. A property of a class, like an instance, is typed and has an expression which can be evaluated to provide a value or an object.

An outline of an example program is shown in Fig. 2.4. The class Section \(^2\) has a signature consisting of two parameters (the Environment of class Environment and connectivity of class SectionConnectivity), a public property (the Roof of class Roof), and a public procedure (report).

**Signature**

Parameters of a class supply to an object of the class all the external information that is needed by an object of that class.\(^3\) Parameter passing is expressed by explicitly assigning values to each of the formal parameters. As discussed in Chapter 4, explicit parameters provide for good modularity as no assumptions need to be made about the context in which an object is created.

A public property of a component object is referenced using the "^" operator, as illustrated in the expression for the property name in Fig. 2.4. Private properties are hidden, providing for encapsulation of a class. All procedures are public. A procedure of a component is called as illustrated in the procedure main in Fig. 2.4.

---

1 Attributes of the class.

2 By convention, class names begin with a capital letter.

3 As Class is single-assignment.
type windAreaType : [lowWind, mediumWind, highWind].
    metres : float.

class Section.
    parameters
        theEnvironment : Environment.
        connectivity : SectionConnectivity.
    public
        theRoof : Roof
            := new(sectionDetail := connectivity,
                   windArea := theEnvironment 'windArea,
                   heightToEaves := sum(collect(s in storeys,
                                                 s 'height))).
    private
        name : text
            := connectivity 'name.
        storeys : bag Storey
            := create(aStorey in selectedStoreys with parameters( 
                        theStorey := aStorey,
                        numberOfStoreys := numberOfStoreys,
                        theRoof := theRoof,
                        effectiveHeight := maxStoreyHeight,
                        allStoreys := storeys)).
    private
        selectedStoreys : set StoreyLevel := ...
        numberOfStoreys : integer
            := ask('How many storeys are there in ', name, ' ? ').

    procedure report.
        call report of theRoof.
        foreach storey in storeys do
            call report of storey.
        done.
    end Section.

%---------------------------------------------------------------

class Roof.
    parameters
        sectionDetail : SectionConnectivity.
        windArea : windAreaType.
        heightToEaves : metres.

    procedure report.
        ...
    end report.

    ...
end Roof.

%---------------------------------------------------------------

instance
    section : Section
        := new(theEnvironment := theEnvironment,
                connectivity := connectivity).
    theEnvironment : Environment := new(...).
    connectivity : SectionConnectivity := new(...).

procedure main.
    ...
    call report of section.
    ...
end main.

Figure 2.4 Outline of a Program with Classes
Objects

Classes extend the available types. A property whose type is a class may be assigned a new object or an existing object. A new object is created with the pseudo-function \textit{new} as illustrated by the property \textit{theRoof} in class \textit{Section} in Fig. 2.4. A collection of objects is created with the pseudo-function \textit{create} as illustrated by the property \textit{storeys}.

An object is created on demand. For example, the instance \textit{section} is assigned an object of type \textit{Section} with the first access to a public component (in this case when the procedure \textit{report} is called in the procedure \textit{main}). The reference to \textit{section} causes the expression associated with it to be evaluated, leading to an object being created. The new object is passed two parameters, although these are not evaluated until they are needed. Note in class \textit{Section} that the collection of \textit{storeys} is passed to each of the \textit{storeys}; this capacity for self-reference is made possible by the lazy evaluation of parameters.

A property or parameter may be assigned an existing object of an appropriate type, which allows for arbitrary connections between objects. This is illustrated in Fig. 2.4 where the object referred to by the property \textit{theRoof} is passed as a parameter to each of the new \textit{storeys}.

Collections

Collections (sets and bags) of objects or values are available, as illustrated by the property \textit{storeys} in Fig. 2.4. A number of operators and functions are provided for manipulating collections of values. For example, the function \textit{sum} adds up the elements of a set or bag; the function \textit{collect} creates a set or bag from each of the components of a set or bag (i.e. like a relational project); and the function \textit{select} selects a subset of a set or bag based on a predicate (i.e. like a relational select).

Procedure calls may be made to each object within a collection. For example, the procedure \textit{report} in class \textit{Section} shown in Fig. 2.4 calls the procedure \textit{report} in each of the \textit{storeys}.

Class Structure Diagrams

It is convenient to have a diagrammatical form for showing the various relationships between classes. Diagrams will help with understanding of the complex class relationships that arise in later chapters.

Some of the component relationships between classes from the example of Fig. 2.4 are shown in the class structure diagram in Fig. 2.5. This shows that an object of class \textit{Section} has a component \textit{Roof} and a bag of component \textit{Storey} objects. It does not show other
connections between objects, such as the reference to the *Roof* supplied to each of the *Storey* objects.

![Class Structure Diagram for Components](image)

Figure 2.5  Class Structure Diagram for Components

```java
class BracingElement {
  parameter wallHeight : metres.
  public bracingProvided : integer := round(bracingPerMetre * length).
  private bracingPerMetre : bu := ...
  private length : float.
  private minimumLength : float := ...
}

end BracingElement.

class DiagonalBrace {
  generalisation BracingElement.
  rules
    length := min(planLength, 1.5*sheetLength).
    private planLength : float := ask('What is the plan length (in metres)? ')
    private sheetLength : float := ask('What is the length of wall covered entirely, 'by sheet material (in metres)? ')
  ...
}

end DiagonalBrace.

class DiagonalBoarding {
  generalisation BracingElement.
  rules
    length := ask('What is the distance measured center-to-','centre between studs covered by diagonal','boarding (in metres)? ')
  ...
}

end DiagonalBoarding.
```

Figure 2.6  Inheritance
2. Introduction to Class

2.5.2 Generalisation Lattice and Inheritance

A class may be declared as a subclass of another class. For example, the class DiagonalBrace shown in Fig. 2.6 has the class BracingElement as a generalisation.

A subclass inherits the signature and implementation of the generalisation class (superclass). It may add to either, as well as supply rules for inherited properties. For example, the property length is declared in class BracingElement in Fig. 2.6 and is assigned a value in both the subclasses. A subclass may provide an expression which overrides an inherited expression for a property; it may also override procedure bodies.

Class generalisation is shown in class structure diagrams as illustrated in Fig. 2.7.

![Class Structure Diagram for Generalisation](image)

Figure 2.7 Class Structure Diagram for Generalisation

Type Conformance and Polymorphism

The generalisation relationships between classes defines a (partial) type ordering. For example, the type DiagonalBrace is a subtype of BracingElement. Hence a property of type BracingElement may be assigned an object of type DiagonalBrace, as shown in Fig. 2.8.

```plaintext
brace : BracingElement := diagonal.
diagonal : DiagonalBrace := new(wallHeight := 2.4).

procedure report.
  display('The bracing provided is ',
            brace^braceProvided, ' B.U.').
end report.
```

Figure 2.8 Conformance
This is appropriate because signatures that are inherited cannot be overridden. Hence an object of type *DiagonalBrace* provides (at least) the interface of an object of type *BracingElement*.

In general, an object of type *C* may be assigned to a property of type *P* (i.e. *C* conforms to *P*), when *P* is a super*type of *C*. The relation super*type is the transitive closure of the supertype relation. There is no notion of inheritance without a type relationship, as compared to Smalltalk and other non-statically typed object-oriented languages.

**Multiple Inheritance**

A class may have several generalisation classes (or superclasses), providing for multiple inheritance. The signature of the class is the union of the signatures of the superclasses. Name clashes between several independent generalisation classes are not permitted. However, a class inherits a single property from a superclass even when there are several inheritance paths. For example, the class *D* in Fig. 2.9 inherits each of the properties of class *A* once.

![Diagram of Multiple Inheritance](image)

**Figure 2.9 Multiple Inheritance**

The order of classes is significant when subclasses override an inherited property expression or procedure body; the code in the lowest class in the generalisation lattice is used before the code in classes further up the type lattice. However, multiple inheritance leads to a partial order of classes, and so the order of execution is not defined completely in *Class*. Snyder (1986) discusses various approaches to turning this partial order into a total order.

**2.5.3 Dynamic Classification**

Dynamic classification of objects is a powerful feature, introduced in *Class*, which allows for the type of an object to be elaborated at runtime. As information is gathered about an
object of some general class, it can be explicitly classified as also belonging to a subclass. In this way, the type of an object is refined as necessary. For example, the classification property `elementType` in Fig. 2.10 specifies a number of possible classes and an expression which selects the appropriate class.

```
class BracingElement.
classification elementType :
  [DiagonalBrace, DiagonalBoarding, SheetBracing,
   SheetMaterial2, ReinforcedConcrete, OtherBracing]
  := DiagonalBrace if standardBraceUsed or
     selectedType = 1
  | DiagonalBoarding if selectedType = 7
  | ...
end BracingElement.
```

**Figure 2.10 Classification: BracingElement Extended**

The class structure diagram for `BracingElement` and all its subclasses is shown in Fig. 2.11.

```
bracingElement
```

```
\[\begin{array}{c}
  \text{DiagonalBrace} \\
  \text{DiagonalBoarding} \\
  \text{SheetBracing} \\
  \text{SheetMaterial2} \\
  \text{OtherBracing} \\
  \text{ReinforcedConcrete}
\end{array}\]
```

**Figure 2.11 Classification**

**Classification is lazy**

The process of classification is done on demand, whenever a possible classification may lead to code which can affect the current evaluation (expressions of properties) or execution.

---

4 Classification is shown by the downward arrows coming from a single point, corresponding to a single classification property. In this case all the subclasses also inherit from `bracingElement` and hence the upward arrow for generalisation.
(procedure bodies). For example, a reference to the property length within the class BracingElement, as extended in Fig. 2.10, will force classification to occur because the subclasses DiagonalBrace and DiagonalBoarding may be classified to and they have rules for length. Classification is lazy in that it is only carried out to the extent that is necessary for the current processing.

**Multiple Classification**

Multiple classification is permitted, through independent classification properties. Hence an object may be classified to several independent subclasses. This provides considerable representational power, because the various cases can be separated out cleanly, improving the information hiding and generalisation aspects of the language.

Hamer et al (1989b) discuss the application of classification to codes of practice, and demonstrate the value of laziness and multiple classification. For example, classification means that the rules for deciding on the specific class of an object can be encoded within the class, rather than outside. This provides for improved encapsulation as compared to a language without classification. Multiple classification also avoids the need to allow for all combinations of separate subclasses to be defined and selected at the time that an object is created.

**Boolean Classification**

Optional classification to a class is also permitted. In this case, the classification property is declared as shown in Fig. 2.12, in which OptionalClass, the class that may be classified to, is written as a boolean property.

```
class Example.
classification OptionalClass : boolean
   := condition.
...
end Example.
```

**Figure 2.12  Boolean Classification**

**2.6  Summary**

*Class* is unusual in that it is a combination of three programming paradigms, of which the object-oriented aspect is most developed. It is a typed, single-assignment language, with functions for creating and accessing collections of objects.
The procedural part of the language provides for sequence control and the production of output. The Class system provides for dependency management. A user may change the value of a previous input and the system will ensure that any values that depend on the changed input are recomputed.

Classes in class-subclass relationships extend the types available in a program. Object creation is lazy, permitting self-referential objects. Multiple inheritance is provided, in which a class may have a number of classes as generalisations. Dynamic classification is a notable and novel aspect of the language, which allows for an object of a class to be refined to multiple subclasses in a lazy fashion. Details of the language, the motivation for its design, and the semantics of classification are provided in Hamer (1990).

Class has been used principally for code-of-practice conformance systems, including the WallBrace system that is introduced in the next chapter. The experience of building WallBrace leads to a critique of the language in Chapter 4 and to subsequent extensions that are described in later chapters.
Chapter Three

WallBrace

3.1 Introduction

This chapter introduces WallBrace, the pivotal work of this thesis. WallBrace is a system to aid a building designer or building inspector to check conformance of a building with the wall bracing requirements of a code of practice for timber frame houses.

WallBrace was intended to go further than previous code checking systems written in Class by developing beyond the prototype stage and incorporating design assistance. It was also intended to act as a catalyst for the further development of Class, the main topic of this thesis.

Designers and building inspectors have difficulties in interpreting the wall bracing requirements of NZS 3604: 1984 Code of Practice for Light Timber Frame Buildings (Sanz, 1984a). Many local authorities use this code and require a wall bracing schedule with a building permit application. It was decided by BRANZ that a computer system could save time for designers, building inspectors, and BRANZ advisory staff who are often asked to help with the wall bracing requirements.

This chapter provides the background to the written code, and describes the overall project development and the resulting WallBrace system. The experience of building the system leads to a critique of Class in Chapter 4. Hence one aim of the current chapter is to provide enough context to allow for detailed discussion of Class in the next chapter.

3.2 The Code

The majority of buildings constructed in New Zealand are of light timber frame construction and hence are required to meet the wall bracing requirements of NZS 3604. Buildings require bracing in order to withstand horizontal loads from wind and earthquakes. Such loads are illustrated in Fig. 3.1.
Figure 3.1 Wind Loads on a Building
**Loadings**

The code specifies the use of building information, such as the height and type of roof, to calculate the loading on a building in two orthogonal directions (in the horizontal plane). The calculations represent a rough mapping from some attributes of the building (and its environment) to a measure of loading which can be used to ensure that adequate bracing has been provided to resist horizontal loads in the building.

The loadings are calculated for both earthquake and wind loadings (as given in Table 11 of the NZS 3604). The earthquake loading is based roughly on the storey height, the surface area and weight of the roof (categorised as light or heavy), and the earthquake zone of the building. The wind load is based on the wind exposure zone, the exposed roof and wall surfaces, and takes account of the number of storeys, the storey height, and the slope and height of the roof.

All loadings are expressed in *bracing units* (B.U.’s). A 2.4 m long section of 2.4 m high wall which is supported by a bracing element (consisting of a diagonal brace with gib-board on one side) supports 100 B.U. (or 5 kN) applied horizontally if it does not distort in that direction by more than 8 mm. The bracing element here is said to supply 100 B.U.

**Bracing**

Once the loadings for the building (expressed in B.U.) are known, appropriate bracing elements are allocated within each storey in each of the two orthogonal directions. A variety of bracing elements may be used, including diagonal steel straps and concrete walls. The total B.U.’s provided by the bracing elements must at least total the number of B.U.’s required by the loadings.

Local bracing requirements ensure that the bracing is well distributed throughout the building. Internal bracing lines are placed across (and along) the building, with restrictions on the distance between them. Bracing requirements apply to each external wall and internal bracing line. For example, an external wall must have at least 10 B.U. per metre of wall, with additional bracing required under some conditions.

**Code Assumptions**

A number of assumptions are made in the written code about the buildings within its jurisdiction. Presumably these were made to keep the code simple, but they are not spelled out in the code.
One assumption is that a building is uniform in structure; i.e. that all parts of the building have the same number of storeys, with the same height, the same type of roof structure, and the same height and slope of roof. For example, the assumption is made in Table 11 of the written code that all the storeys are of the same height. Another assumption is that the building shape is simple enough that the wind loads on the parts of the building can be easily combined to produce the overall wind load.

3.3 The Research Project

3.3.1 The Problem and the Need for an Application System

BRANZ has found that some people have difficulties in understanding and applying the code (Ten Broeke, 1987). Problems arise from users not knowing how to interpret the tables or clauses of the code: "I don't have a wall at 5m from the exterior wall - what can I do?" The aim of the code is to spell out the bracing requirements; it provides no guidelines on how to meet those requirements. For example, it gives no assistance with questions such as "How can I get enough bracing to meet the total required? Can a ceiling diaphragm be used to fix this?"

A manual design aid had been developed by BRANZ to address these problems: the NZS 3604 Wall Bracing Calculation Sheets (BRANZ, 1986). These spell out the checking procedure for ensuring conformance with the wall bracing requirements of NZS 3604. These calculation sheets are extensively used by building designers, yet some still experience difficulty in making use of the information. Along with the written code, these calculation sheets formed the starting point for the design of WallBrace, as discussed in section 3.4.2.

The immediate aim was to provide a system which could help a user to determine the earthquake and wind loads on a building, to lay out the bracing lines, and then to allocate the wall bracing to the building. On completion of the task, the system was to produce a summary report of the loadings on the building and the bracing that had been provided. This report could then be attached to a plan of the building for submission to the local authority along with the building permit application.

The resulting system was also to play an educative role, making explicit the steps and calculations that were being carried out. This would enable a user to gain a better understanding of the wall bracing requirements and of the steps involved in meeting them.
3.3.2 The Research Project

Several aims were established for the project beyond the immediate aim of constructing a usable system. Of special interest here is the aim of fostering the further development of Class. Other aims and results are not relevant to this thesis; they are discussed elsewhere (Mugridge and Hosking, 1988; Mugridge and Hosking, 1989).

WallBrace was commissioned by BRANZ in 1987 and was completed in 1989. Intended users of the system were building designers and inspectors, who would be familiar with the terminology of the code.

There were two major phases in developing WallBrace: construction of a checker, and extension to a design aid.\(^1\) It was thought that once the checker was complete, it would be easier to develop the design aspects of the project. It would also enable the checking system to be evaluated while the design aspects were being added.

The checking system helps a user check that the wall bracing requirements of NZS 3604 have been met. It was intended that the checker be developed beyond the prototype stage and be the subject of a field trial before being made available to the building industry. Access was to be through a packet-switching network so the user interface was to be restricted to text only.

The second phase of the project extended the checker system to guide or advise the user in designing wall bracing to meet the code requirements. This phase was oriented more towards research and was not to be developed beyond a prototype. The project, however, showed that there were important research aspects in both phases.

As the project developed, it became clear that the written code does not spell out the loadings and bracing requirements for all buildings. The code only explicitly deals with simple buildings, with a minimal variety of building height and roof structure. This led to the checker being developed in two stages. Stage One was concerned only with simple rectangular buildings which could be handled directly by the code as written. Other buildings were dealt with in Stage Two.

---

\(^1\) The design stage of WallBrace is not discussed further here. Other details appear in Mugridge and Hosking (1988).
A prototyping approach was taken throughout the project. This had the following advantages:

- early feedback was available to the expert on the form of the expert system;
- the expert could point out inappropriate dialogue and incorrect logic;
- the programmer could gradually become familiar with the complexities of the code.

A prototyping approach is facilitated by a suitably modular implementation language. This issue is discussed in Chapter 4.

3.4 Stage One: Development of a Simple Checker

The Stage One Checker deals with simple rectangular buildings. It gathers details of a building design and provides information about the loadings on the building and whether the supplied bracing conforms to the code requirements.

Three aspects of the development of Stage One are relevant and are considered further: object-centred design, the use of a goal-directed approach, and user interface issues.

3.4.1 Object-Centred Representation

It is necessary to model the building being considered, in order to organise the information being collected for conformance checking and to avoid asking the user redundant questions. Much of the building model goes beyond the code. For example, the code only has to specify the requirements for a storey; the application system has to deal with each of the storeys within a building.

An object-oriented approach is appropriate for creating a model of a building within which the attributes of the building components and the associated code provisions are represented. The active process of checking a building is also included within this framework. This consists of collecting information from the user about the building, checking the code provisions, and providing information back to the user. Design aid aspects were later incorporated within this framework.
Class Selection

The design of the system was initiated by considering the significant objects (such as building, roof, and wall) that are mentioned in the written code. The building components that play a peripheral role in bracing were left out of the initial design. For example, the size, weight and slope of the roof play a significant role in determining the wind and earthquake loadings on a building, while “dragon ties” play a peripheral role.

Consider the clause shown in Fig. 3.2. Several significant objects are mentioned: bracing elements, walls, storeys, and roof. Some attributes of these objects are also mentioned: gross roof plan area, bracing units of bracing elements, storey height, and roof slope, height, and weight. Finally, relationships between some of these objects are mentioned or are implicit: between storeys, between a storey and the roof above it, the walls within a storey, and the bracing elements within a wall.

6.3.2.1
The total number of bracing units of all wall bracing elements in each of two directions at right angles to each other in any storey shall be not less than the greater of:

(a) The number of bracing units per square metre given by table 11A for earthquake multiplied by the gross roof plan area in square metres of the roof above the storey being considered, provided that for the lower storeys of two-storey and three-storey buildings with light roofs the gross floor area of the floor above the storey being considered may be used instead of the gross roof plan area; ...

(b) The number of bracing units per metre given by table 11B for wind multiplied by the maximum horizontal dimension of the roof above the storey being considered measured at right angles to the wall bracing elements being considered, provided that for roofs not steeper than 25° the maximum horizontal dimension of the external wall of the storey being considered may be used instead of the maximum horizontal dimension of the roof.

Figure 3.2 Clause 6.3.2.1 of NZS 3604

The selection of significant objects suggested the classes to develop initially. These classes provide a framework in which to organise the main elements of the system. The main elements include the code provisions, the information needed about building components in order to check the provisions, the information to be supplied to the user as displayed information and as a report, and the mechanisms of providing design assistance.

Another approach to the initial stages of design is to first list all the objects, and their properties, that are mentioned in the code. This is an approach advocated by Fenves et al
(1987) in applying various computer-based techniques to the process of regulation formulation. This has the advantage that all objects are made explicit early in the project, but the disadvantage that many of the objects will not be relevant to the system development.

**Code Provisions**

Provisions of the code were placed in the appropriate class or classes. Consider the example code provision from NZS 3604 shown in Fig. 3.2. For each direction, there are three main pieces of information that are needed in order to check conformance with the clause:

- the total number of bracing units supplied by bracing elements
- the number of bracing units required, based on the earthquake loading
- the number of bracing units required, based on the wind loading

Each of these pieces of information in turn depend on other, more detailed information, as specified in parts (a) and (b) of the clause.

Conformance of a building with Clause 6.3.2.1 is to be checked for each of the two directions of a building. The logic of this clause is therefore placed in a class concerned with a loading direction. The information needed for checking conformance is gathered from other parts of the model of the structure of a building.

### 3.4.2 A Goal-Directed Approach

Once the overall class structure had been mapped out, it was appropriate to turn to the active, problem-solving aspects of the system. A goal-oriented approach was taken by considering the final report to be produced and the information to be displayed to the user. The report was based on the BRANZ bracing sheets. From this it was possible to work back to the information required from the user.

This approach was also taken in the *Seismic* project (Hosking et al, 1988). Working backwards from the final results of the system focuses attention on the logic of the problem-solving, rather than on the sequence in which it happens to be carried out.

The final report was organised around the subcomponents of the class *Building* (see Fig. 3.7), which generate the different parts of the report. For example, the *Building* class adds information about the building and its environment to the report, while the *Roof* class adds information to the report about the roof.
As the production of the report is a sequential process, the report was encoded as procedures in the various classes. During development of the system, additional properties were added to the classes as they were required in producing the report. This meant defining the questions to be asked of the user, or the rules to be used in finding a value for a property. For example, class Storey was extended with the following properties as the parts of the report were added to the procedure inside that class: height, length, width, buildingPlanArea, and seismicLoad.

This clearly separates those parts of the system which are concerned with the logic of the problem-solving from those parts which are concerned with the overall sequence in which it must be carried out. These are encoded in different forms: the first as rules which are used when needed, and the second as procedures which dictate the flow of control. Procedures make use of property values in producing the summary and displaying information to the user; this has the effect of initiating evaluation of a property when a value is unknown.

### 3.4.3 User Interface Issues

There were some difficulties in making the dialogue sensible to the user, both in the language used and in the order in which the consultation occurs; the major obstacle was the restriction of the user interface to text only.

**Improving the Prompts**

Once multi-storeyed buildings were considered, contextual information was needed in the prompts. The following types of context were provided: the particular wing of the building, the storey within the wing, and the direction (along or across the building) within the storey. For example, in the following, the second question is better than the first because it is explicit about the context:

- What is the height of this storey (in metres)?
- What is the height of the top storey of the library wing (in metres)?

However, this made the prompts more verbose.

**Providing Information During the Consultation**

Feedback is provided to the user about the overall intention behind groups of questions, including the display of intermediate results that have been calculated. Providing such information enables the system to play an educative role with a designer who is not familiar
with the code. For example, as the system determines the earthquake and wind loadings on each part of a building, it makes explicit to the user the process of collecting the appropriate information and the calculations that lead to the resulting loading values.

**Focusing the Dialogue**

Once the basic logic of the system was encoded, it was found that the dialogue was sometimes confusing. To remain coherent to the user, all information about some part of the building needs to be collected at one time, rather than information being collected as it was needed by the system. This meant that it was necessary to alter the sequence of questions that resulted from the execution of the system as a single-assignment program with procedures. This issue is discussed in Chapter 7 when considering I/O in functional programming languages.

**Shortcuts**

It is often appropriate to avoid asking the user many questions about situations which rarely arise in buildings. For example, the general question "Are dragon ties used in the building?" can avoid repeated questions about dragon ties for each storey of the building. Similarly, asking whether all storeys have the same dimensions avoids repetitive questions in a multi-storey building.

**Help Information**

Help text was added later in the development of the system. There are two types of information offered: *what* and *why*. "What" information proved to be useful in explaining questions in greater detail, while "why" information was of questionable value.

### 3.5 Stage Two: A General Checker

As mentioned above, the wall bracing requirements of NZ3604 make some implicit assumptions about the uniformity of the building being braced. Stage Two of *WallBrace* was extended beyond the written code in order to handle a variety of building shapes.

Stage Two also addressed the need to avoid gathering redundant information, both for the user's sake and for retaining consistency. The Stage One Checker did not ensure consistency in all cases. For example, it asked about the diaphragms connected to each wall individually. This led to repetitive questions about diaphragms to which inconsistent answers could be given.
3.5.1 Wind Loadings on Non-Trivial Buildings

The written code only specifies the wall bracing requirements of simple rectangular buildings. It does not acknowledge other buildings, such as those that are U-shaped or those that have different numbers of storeys in different parts of the building. For example, the building plan shown in Fig. 3.3 consists of three parts, with a central two-storey part flanked by single-storey parts.

While it is clear that the provisions of the code should be applied to the individual sections of the building, the code does not specify how to combine the wind loadings from individual sections to calculate the overall wind loading on the individual storeys of the building. A means of load combination had not been formalised in the code, so it was necessary to create such a formalisation for Stage Two. This proved to be more difficult than at first expected.

The written code defines the wind load on a single rectangular building. However, the wind load on a building such as that shown in Fig. 3.3 is not simply the sum of the wind loads of the individual sections of the building. Parts of the building obscure other parts, so their spatial relationships must be taken into account in calculating the overall wind load.

![Figure 3.3 An Example Building Plan](image)

The wind load on a part of a building depends on the number of storeys in that part and the type of roof above it. Thus a rectangular building with two different roof structures is handled as two separate sections (as far as the written code is concerned) because the wind and earthquake load calculations are distinct for the two parts.

The loading calculations incorporated in the system depended on detailed spatial information about a building, leading to the need for a plan entry module.

3.5.2 Plan Entry

The plan entry module determines the position of the external walls of a building. An appropriate interface was found to be difficult to achieve, as the entry of the plan had to be
made entirely through textual questions and answers. Posing suitable questions was difficult, leading to an excessive amount of interaction with the user. Several approaches were taken to plan entry in order to minimise the impact of these problems. None were satisfactory.

The first approach to plan entry allowed a user to "build up" a plan by starting with one section and then defining other sections in relation to known sections. Awkward questioning resulted. This approach was improved slightly by allowing specifically for three common building shapes: rectangular, L-shaped, and T-shaped. These shapes were initially provided in a separate printed sheet, but were then incorporated into the dialogue.

3.5.3 Roof Structures

General information about the roof on a rectangular building was requested from the user in Stage One of the project. More specific information was required in Stage Two.

The slopes and height of the roof on a rectangular section are needed in order to calculate the wind load on it as an individual, isolated section. The relationship between the sections is provided by plan entry, so that the combined loadings can be calculated.

Rather than asking the user for the slope of each roof, in each direction, it was appropriate to categorise the major roof types (gable-end, hip, flat, etc) and to gather information from the user based on these types. This was in order to:

- ensure consistent information. For example, the slope of a symmetrical gable-end roof can be calculated from the roof height, and vice versa.

- reduce the questions asked of the user. For example, it makes no sense to ask for the slope of a flat roof, or the slope of the gable-end of a gable roof.

This would allow for the overlap of adjoining roofs, such as the L-shaped building shown in Fig. 3.4 with a hip roof overlapping a gable-end roof. Slope information from the adjoining roofs is used to automatically scale the area of roof exposed to the wind.
3.5.4 Calculating Loadings

Some code requirements involve relationships between several objects and require non-trivial spatial analysis. Consider the clause shown in Fig. 3.5, in which the dimensions of a diaphragm have an impact on the bracing requirements of the supporting walls.

6.3.5.2
Each edge of the diaphragm shall be connected to a wall containing a total of not less than 10 bracing units per metre of diaphragm dimension measured at right angles to the wall being considered, provided that no such wall shall contain less than 100 bracing units. Where two diaphragms are connected to a wall, then the requirements for that wall shall be the sum of those required for each diaphragm.

Figure 3.5 Clause 6.3.5.2 of NZS 3604

Local bracing requirements are intended to ensure that the wall bracing within a building is distributed throughout the building, to avoid weakly braced parts. In addition, the load of a bracing line is affected by the diaphragms attached to it.

6.9.6.3
Such wall bracing elements shall not be considered to contribute more than 50 percent of the total number of bracing units required in all bracing lines in any direction.

Figure 3.6 Clause 6.9.6.3 of NZS 3604

Such requirements necessitate the need for complex interrelationships between objects modelling a building. These relationships have to be gathered from the user in a careful manner. They also require spatial reasoning in order to make the final connections.
3.5.5 Laying Out Bracing Lines & Positioning Diaphragms

Details of internal walls in a building are not collected during plan entry because of the difficulties of the textual interface. Given that it is unrealistic to expect a user to enter the position and length of all the walls, the system "steps" across (and along) a storey of a building, so that the user can position the bracing lines. Wherever possible, the system makes use of plan information to reduce the questioning. Where diaphragms are used (i.e. the distance between bracing lines is greater than 5 m), the dimensions of the diaphragms are collected too.

Once again, the restriction to a textual dialogue makes this awkward. This process would be much simplified if plan information included the position and dimensions of all walls in a building. The selection of appropriate walls for bracing lines could be suggested, rather than relying on the user to specify the position of some of the walls. The approach taken in WallBrace has been motivated by the need to minimise the questions asked of the user.

When it is not possible to allocate adequate bracing elements in a building, such as in a building with few internal walls, it may be necessary to introduce additional walls before the building conforms to the wall bracing requirements. The current system does not provide assistance with changing the placement of bracing lines nor with adding further bracing lines so as to allocate extra bracing elements. These restrictions were motivated by the difficulties with the user interface.

3.6 Class Organisation of WallBrace

A brief description of the WallBrace program is provided as a basis for examples in the following chapters. WallBrace is a large program with many classes. The primary class organisation is based on the representation of a building in terms of wings, storeys, roofs, walls, bracing elements, etc. However, it is useful to consider this organisation in terms of three main phases of operation of the system (although these phases are not strictly sequential).

3.6.1 An Overview

Fig. 3.7 provides an overview of the class structure of the final WallBrace program. The main object is of class Building (encoding details about a building with one or more wings). It contains an object of class Environment (with wind and seismic details; i.e. the environment of the building) and a set of objects of class Wing.
The three main phases of the system are: Plan Entry, Calculate Loadings, and Bracing. For simplicity in the following discussion of these phases, it will be assumed that a building has only one wing.

### 3.6.2 Plan Entry

The plan entry phase gathers details of the plan of the building concerned, represented as a set of objects of class *Rectangle*.

Once the outside shape is known, the building may have to be broken down into sub-rectangles (and further recursively), such as in the plan shown in Fig. 3.8.
Once the position and dimensions of all rectangles have been found, this information is gathered together in a set of rectangles, within the Wing class. This set is the result of this first phase.

3.6.3 Calculate Loadings

The loadings phase uses details about the storeys (e.g. height) and the roof (e.g. height and slopes) over each rectangle (or “section”) of the plan, and information about the wind and seismic zones, to calculate the wind and seismic loads on each storey of each section. The result of this phase is a set of loading information for each storey of each section of the building.

For each rectangle, the second phase constructs a Section, as shown in Fig. 3.7. A Section corresponds to the storeys and roof above a single rectangle supplied by the Plan Entry phase. For each storey of a section it calculates the wind and seismic loads, based on such factors as the height of the storey, the slope and height of the roof, and the wind or seismic zone.

The wind and seismic loads are calculated in accordance with the written code. An assumption is made in the written code that the wind load on a section of a building is unaffected by other sections. Independent wind and seismic loading details (SectionStorey) for each Storey of each Section are supplied to the final bracing phase of the system, which takes account of the section positions when calculating the overall load for each storey.

3.6.4 Bracing

This final phase considers the building as a set of storeys (or levels), rather than the section view of the previous phase. To do this, it takes the results of the previous phase and merges the loading information on a level (horizontal) basis, rather than on a section (vertical) basis.
For each level of the building, the horizontal loads are calculated by taking account of the relative placement of storeys and sections.

For each such storey of a building the following steps occur:

- the overall loadings are calculated
- the bracing lines are allocated along and across that level
- the local wall bracing requirements are calculated, based on the type of wall and the use of diaphragms and dragon ties.
- the bracing elements are allocated in each bracing line
- a check is made that the local and global bracing requirements are satisfied

3.7 Summary

One of the aims of building WallBrace was that it should act as a catalyst for the further development of Class. In this regard, the process has been extremely successful, as many ideas for enhancements to Class have arisen from this work.

Developing WallBrace beyond the prototype stage has made it clear that there are shortcomings in the user interface. The original aim of the project was to produce a system which could be used through a packet-switching network and hence the dialogue with the user was restricted to the use of simple text. While this has limited the quality of the resulting system, it has made it clear that a graphical user interface is essential for the Class system.

A general critique of the user interface provided by Class is given in Chapter 4. Some of the issues raised are addressed in Chapter 7.

Much detailed information is required in WallBrace to check a building. Therefore the value provided by WallBrace in checking a building must be balanced against the effort of data entry. Several approaches are possible to providing a better balance, including:

- integrate a number of related code-checking systems together, so that the user need only enter plan information once.
- share or transport data from another system, such as a drafting package. This also avoids duplicated data entry. Hence access to databases containing building information will be important.
The development of *WallBrace* was surprisingly complex, given the apparent simplicity of the written code. This complexity was mostly due to representing complex spatial relationships and extending the system beyond the written code.

Dealing with these complications in the *WallBrace* program has made it clear that *Class* has strengths (such as classification) and weaknesses (such as poor functional abstraction), as discussed in Chapter 4.
Chapter Four

A Critique of Class

4.1 Introduction

A critique of Class is provided, based on the experience of developing WallBrace. This chapter does not attempt to provide a general evaluation of Class. Such an evaluation is provided in Hamer (1990). Instead, good and poor aspects of the language are discussed within the context of the WallBrace program.

Some of the problems that are isolated are symptomatic of a lack of generality of some aspects of the language. These form the motivation for the language extensions described in later chapters.

While the single-assignment aspects of the language are satisfactory, there is a clear need for functions. The user interface provided by Class is poor, being based only on text. Much procedural specification is required because the only means of producing output in a Class program is through procedure calls.

The strengths of Class lie in the object-oriented aspects of the language, and dynamic classification in particular.

4.2 Single-Assignment Aspects

The single-assignment aspect of Class is weak because of the lack of functions. The functions for processing sets and bags are a useful feature of the language, but sets and bags have the limitation that additional functions for processing them cannot be written in the language. The change facility, which is possible because of Class being a single-assignment language, has proven to be a most useful feature.

4.2.1 Sets and Bags

The pre-defined functions for processing sets (and bags) of objects provide a convenient means of handling collections of objects. For example, the expression for the property
diaphragmLoad in Fig. 4.1 computes the load that may be placed by diaphragms on either side of a wall\(^1\) using the function *any*. The local “argument” *d* within the *any* function call is used to refer to each element in turn and to specify a condition on the element being accepted.

```plaintext
class CalculateWallLoading.
    parameter diaphragmRects : set Rectangle.
        ...
        diaphragmLoad : integer
            := max(100, leftLoad + rightLoad)
                if leftLoad + rightLoad > 0
                | 0.
        leftLoad : integer
            := round(step * 10)
                if any(d in diaphragmRects where
                    wallLine - 0.1 > d\(^\text{yMin}\) and
                    wallLine - 0.1 < d\(^\text{yMax}\))
                | 0.
        rightLoad : integer
            := round(nextStep * 10)
                if any(d in diaphragmRects where
                    wallLine + 0.1 > d\(^\text{yMin}\) and
                    wallLine + 0.1 < d\(^\text{yMax}\))
                | 0.
        ...
end CalculateWallLoading.
```

**Figure 4.1 Use of the Function *any***

The parameter *diaphragmRects* contains a set of *Rectangle* objects, which specify the position of all diaphragms in a storey of a building. The class *Rectangle* is shown in Fig. 4.2; it is used simply as a record. The properties *leftLoad* and *rightLoad* check each side of the wall to see if there are any diaphragms abutting or overlapping the current wall.

```plaintext
class Rectangle.
    parameter xMin, yMin, xDist, yDist : float.
    public xMin, yMin, xDist, yDist, xMax, yMax.
    yMax : float := xMin + xDist.
    yMax : float := yMin + yDist.
end Rectangle.
```

**Figure 4.2 The Class *Rectangle***

Another example of the use of sets is in calculating the relative position of parts of the building; some of this code is shown in Fig. 4.3. The property *shiftedExternalWall-*

---

\(^1\) Based on the provision in Fig. 3.5 from NZS 3604.
Positions calculates the position of each of the external walls within a building, translating the position of the walls so that the first one is on the y-axis. The position of the external walls is calculated by taking the union of the set of the positions of the two sides of each of the rectangles; these rectangles represent the position of the "sections" within the building.

```plaintext
shiftedExternalWallPositions : set float
  := collect(x in ExternalWallPositions, x - minPosition)
ExternalWallPositions: set float
  := collect(s in sections, s^rect^xMin) +
  collect(s in sections, s^rect^xMax).
minPosition : float
  := min(ExternalWallPositions).
```

Figure 4.3 Other Use of Sets

It would be convenient if other functions for manipulating sets could be written within the language, rather than having to be provided as language intrinsics. For example, a function flatten had to be added as it was needed in WallBrace (the function takes a set of sets and returns a set of the elements of the sets, so that flatten( [[a, b], [], [c]]) returns [a, b, c]).

Sets and bags are not created in a lazy manner; this is inconsistent with the rest of the language. Where laziness is required, it is necessary to use classes to provide a sequence, as discussed in Section 4.5.3.

4.2.2 Need for Functions

Class does not have functions. A class can be used to provide a function over values but it is awkward. For example, in WallBrace, the wind load on a part of a building may be affected by other parts which block the wind flow. In order to calculate the wind load it is necessary to calculate the extent that one part is obscured by other parts in each of four major directions.

For each direction, this calculation depends on the amount that the side of the part (a rectangle) is obscured. It is based on the overlap of the two sides and hence on the overlap of two ranges of values. This is encoded in WallBrace as shown in Fig. 4.4.
class Neighbour.
parameter xyRange, 
    xyRangeOther : Range.
...

distanceObscured : float
    := min(xyRangeOther\textsuperscript{distance},
             xyRange\textsuperscript{maximum} - xyRangeOther\textsuperscript{minimum})
    if xyRangeOther\textsuperscript{minimum} >= xyRange\textsuperscript{minimum} and
    xyRangeOther\textsuperscript{minimum} <= xyRange\textsuperscript{maximum}
    | xyRangeOther\textsuperscript{maximum} - xyRange\textsuperscript{minimum}
    if xyRangeOther\textsuperscript{maximum} >= xyRange\textsuperscript{minimum} and
    xyRangeOther\textsuperscript{maximum} <= xyRange\textsuperscript{maximum}
    | xyRange\textsuperscript{distance}
    if xyRange\textsuperscript{minimum} >= xyRangeOther\textsuperscript{minimum} and
    xyRange\textsuperscript{minimum} <= xyRangeOther\textsuperscript{maximum}
    | 0.0.
...
end Neighbour.

Figure 4.4 Repetitious Code

The class Range, as shown in Fig. 4.5, is a simple data structure which holds the details of a range of values.

class Range.
    parameter minimum, distance : float.
    public minimum, distance, maximum.
    maximum : float := minimum + distance.
end Range.

Figure 4.5 Class Range

The code for the property distanceObscured in Fig. 4.4 checks whether a point is within a range in three places. This is repetitive and obscures the fact that there is a common function involved.

Fig. 4.6 shows that using a class to encode a function to determine overlap is unsatisfactory, as all the information has to be passed as parameters. This is clumsy because a class is more general than a function and because classes can’t be nested.
4.3 The User Interface and Procedures

The user interface is one of the weakest parts of the Class system. This, in part, was due to the BRANZ requirement of providing application systems over a packet-switching network, where only simple text could be used.

In addition, other forms of input-output are required for Class programs. For example, there is a need for access to databases and files, and for communication with other, specialist programs, such as for plan entry.

Procedures are used in WallBrace both for defining the sequence of execution and for specifying the sequence of text produced by display and summary statements. However, excessive use of procedures was required in order to coordinate the production of the report (with summary), collect information from the user in a coherent manner (with ask), and provide additional helpful information to the user (with display).
4.3.1 Limitations of a Textual Interface

Clausen (1989) discusses a number of problems with the Class user interface, based on the interface of WallBrace and those of other application systems, such as Seismic (Hosking et al, 1989). Problems with the user interface include:

- Limitations of text for interaction. Pictures are often more appropriate.
- Misinterpretation of input, with weak feedback. The difficulty of using text for such activities as plan entry increases the likelihood of errors in user input.
- Poor error messages when incorrect input is provided.
- Input of repetitious data is tedious. For example, in using WallBrace a user has to enter information about the bracing in each wall, even though the bracing in several walls is the same.
- Choice of data input is awkward. For example, in WallBrace the user may know either the roof slope or the roof height; one can be calculated from the other (given other dimensions of the roof). Extra questions are needed to provide the user with a simple choice of which data to provide.
- Lack of user control: the "sequencing problem". Users are forced to answer the questions provided by the system; they have little control over the flow of the dialogue. This issue is discussed further in Chapter 7.
- Poor interface for making changes to previous inputs. In order to change the answer to a previous question the user needs to know the question number of that question. The 'review' command is used if the question involved is no longer displayed on the screen but it can be tedious for the user to find the question to be changed.
- Poor responses for requests for 'what' and 'why' explanation.

In addition, Clausen suggests that a graphical interface could be used to improve the programming environment of Class, although he does not propose any specific approaches.

4.3.2 Single Service: Undue Emphasis on the Report

WallBrace checks a range of code requirements, providing information on conformance or otherwise as it goes. One weakness of the system is that the overall flow of control is
organised around the production of the final report. This is one aspect of the “sequencing problem” mentioned above.

Information collected from the user and supplied in displays has to be synchronised with report production. This leads to excessive use of procedures and places undue emphasis on the report, even when it is not needed. While this is to some extent a criticism of the WallBrace program, it is difficult to take a radically different approach because of the need to use procedures in Class to produce output to the user.

4.3.3 Coherence of Questions

It would be confusing to a user if WallBrace did not collect all the relevant information about some building part at the same time. Coherence is achieved by using procedures and when procedures to ensure that related information is collected together.

For example, consider the class shown in Fig. 4.7. When an object of this class is created, the “when true” when procedure is executed to force the evaluation of each of the properties, in turn causing the user to be asked a series of questions.

As the example in Fig. 4.7 shows, many aspects of the user interface have to be explicitly encoded in the program. Even simple constraints on input values, such as the requirement that the planLength be a positive value, have to be handled with when procedures. Hence the programmer must take considerable care in ensuring that appropriate synchronisation and constraint-checking takes place. There is clearly a need for a better approach to handling the user interface.

---

2 The change statement was introduced into the language so that the user could be re-prompted for a value. This statement specifies the property value that is to be changed; this is treated the same as if the user had requested the change.
class DiagonalBracesWithSheetMaterialInterface.
parameter braceName : text.
    minimumLength : metres.
    bracingProvided : bu.
public diagonalBrace: [steelAngle, letinTimber, 
    pairsFlatSteel, pairsCutinTimber]
    := ask('Which diagonal brace: '&
       ' 1. Steel angle brace'&
       ' 2. Let-in timber'&
       ' 3. Pairs of flat steel strip'&
public sheetMaterial : [oneFace, bothFaces]
    := ask('Is the sheet on: '&
       ' 1. One face'&
       ' 2. Both faces'&prompt).
public planLength : metres
    := ask('What is the plan length of', braceName, 
       ' (in metres) ? ').
    when planLength <= 0 do
        display('Please enter a positive number!').
        change(planLength).
    done.
    when planLength > 0 do
        if planLength < minimumLength then
            display('The brace must be at least ',
                fmt(minimumLength, 1, 2), ' m long.').
            change(planLength).
        endif.
    done.
public sheetLength : metres
    := planLength if fullSheeting
       | ask('What is the length of the wall entirely',
           ' covered to full height by such sheet',
           ' material (in metres) ? ').
    when sheetLength <= 0 do
        display('Please enter a positive number!').
        change(sheetLength).
    done.
    when true do % Force all questions together
        determine(planLength).
        determine(diagonalBrace).
        determine(sheetMaterial).
        determine(sheetLength).
        display('This provides ', bracingProvided, ' B.U.').
    done.
end DiagonalBracesWithSheetMaterialInterface.

Figure 4.7 An Example of Interface Control

4.4 Object-Oriented Aspects

The benefits of an object-oriented approach are well known (Meyer, 1988). Two aspects of classes in Class are considered here: modularity and "type loss".
4.4.1 Modularity

The value of classes in WallBrace is illustrated in Fig. 4.8. An object of class Wall represents a bracing line or external wall running in one direction in a storey level in a building.

Walls are complicated because there are three semi-independent aspects to a wall, as discussed in Section 3.6:

- the process of selecting the next wall for a bracing line
- calculating the bracing requirements of the wall, based on such factors as diaphragms used
- allocating bracing elements in the wall to meet the bracing requirements

In order to reduce the complexity of this part of the program, three partially independent "views" of a wall are created within the class structure.

The role of each of the classes is now considered briefly. At the bottom is the class RecursiveWall, which handles the recursive³ creation of a set of walls, based on the allocation of walls across (or along) a storey of a building.

The class Wall is the only class of the family that is intended to be accessed elsewhere. It is introduced to make explicit the public properties which are for use outside the family. The class RecursiveWall has additional public properties which are used in the recursive generation of the sequence of walls.

The three classes above Wall deal with processing each of the aspects of a wall, as outlined above: allocation, loading calculation, and bracing. The two classes above those (GeneratedWall and LoadedWall) contain information that is common across these three sub-phases. Finally, the most general information is held in the top-most class, WholeWall.

³ This process is discussed later in Section 4.5.3.
Information hiding is needed at a level above classes, as many classes are only relevant in one context and are not meant to be used elsewhere. For example, only the class Wall in Fig. 4.8 is intended to be used outside of the Wall class family.

4.4.2 The “Type Loss” Problem

Class has a “type loss” problem, as illustrated in Fig. 4.9, in which specific type information is unavailable about the object referenced by a. There is no facility within the language to access the public property bl in class A1, even though the classification property this can be tested.
4. A Critique of Class

4.5 Classification

Classification is a useful means of refining the class of an object at runtime. For example, an object of class *Roof*, as shown in Fig. 4.10, is dynamically classified when the user selects the type of roof on a part of a building.

![Diagram of Roof classification]

**Figure 4.10** Classification in *Roof*

4.5.1 Multiple Classification

Multiple classification allows for several independent aspects of a class to be dealt with separately, without needing to provide for all the valid combinations. For example, the class *BraceWall* from Fig. 4.8 has two classification properties, as shown in Fig. 4.11.

One important distinction in *WallBrace* is whether a wall is an external wall or a bracing line (or internal wall). For example, the loading and bracing requirements differ between the two

```java
class A.
  classification this : [A1, A2].
  ...
end A.

class A1.
  generalisation A.
  public b1 : integer.
end A1.

a : A := new ....
i : integer
  := a^b1 if a^this = A1  % Invalid Access
  | a^b2 if a^this = A2.  % Invalid Access
```

**Figure 4.9** Classification Type Information is "Lost"
types. The other distinction that is made in Fig. 4.11 is concerned with the amount of help that is supplied to the user in allocating bracing in a wall.

Figure 4.11 Multiple Classification

Hamer et al (1989) make the point that without classification, all valid combinations of categories have to be encoded outside the class. This leads to redundancy in the code, is non-lazy because the selection between the valid possibilities has to be made all at once, and it means that properties used in the selection have to be encoded outside the class. For example, without classification the code for creating an object of class BraceWall would require expressions as shown in Fig. 4.12.

```
wall : BraceWall
    := new InternalBraceNovice(direct := direct,
                              storey := storey,
                              previousWall := ...)
    if internalWall and noviceBracing
      | new InternalBraceExpert(direct := direct,
                              storey := storey,
                              previousWall := ...)
    if internalWall and not noviceBracing
      | new ExternalBraceNovice(direct := direct,
                              storey := storey,
                              previousWall := ...)
    if not internalWall and noviceBracing
      | new ExternalBraceExpert(direct := direct,
                              storey := storey,
                              previousWall := ...).
```

Figure 4.12 Without Classification

4.5.2 Classification and Inheritance

Classification and inheritance are not always in direct correspondence. For example, the classification structure for the Roof class family shown in Fig. 4.10 differs from the inheritance structure, as shown in Fig. 4.13.
Dynamic classification is made directly to the individual roof types, based on information supplied by the user. Two intermediate classes are used to organise the roof types further: RidgedRoof and NonFlatRoof. For example, the class RidgedRoof deals with the orientation of the ridge.

### 4.5.3 Sequence: Recursive Classes

Where explicit sequences were needed in WallBrace they were encoded through the use of recursive classes. A recursive approach is essential when the number of elements in a set is not known at the time the set is created, such as in the Wall class family.

The class RecursiveWall, as shown in Fig. 4.14, constructs a sequence of walls recursively, providing a set of walls as a public property. An object of the class is classified to either the class EndOfWalls (the base case of the recursion) or the class FurtherWalls (the recursive case). The latter class contains a property of class RecursiveWall; this is the remainder of the sequence.
A similar approach was taken in handling plan entry. A generalisation of the approach taken is illustrated in Fig. 4.15. However, it was not possible to abstract out the common aspects of recursively created sequences because different type information was required for plan entry and for walls. Hence there is a need for generic classes, in which some type information is left undefined in the abstract definition. In addition, it is not possible to write higher-order functions such as `select` on such recursively defined collections.

![Figure 4.15 A Recursively-Defined Sequence](image)

**4.5.5 Inconsistent Clusters**

There is a general problem with multiple inheritance which is discussed here as three related issues.
The first issue is illustrated in Fig. 4.16, in which the class $A$ has a classification property $p$ which selects between the classes $A_1$ and $A_2$. When an object of the class $A_1$ is created, the Class system does not ensure that $p$ takes the value $A_1$. An inconsistency can then occur if the classification property $p$ subsequently takes a value of $A_2$.

![Figure 4.16 Inconsistency with Classification](image)

The second issue is illustrated in Fig. 4.17, in which a class $B$ has generalisations $A_1$ and $A_2$. However, this is inconsistent, as $A_1$ and $A_2$ are mutually-exclusive subclasses of $A$. The multiple inheritance of $B$ should be rejected as invalid, but this is not done.

![Figure 4.17 Invalid Multiple Inheritance](image)

The third issue also arises with mutually-exclusive subclasses, but where there is no classification property. This is illustrated in Fig. 4.18, in which the classes $A_1$ and $A_2$ are mutually exclusive. The issue of mutually-exclusive classes (clusters) has been raised in the database community (Smith and Smith, 1977). Assuming that the previous two problems have been corrected, this third problem can be handled in Class by adding a classification property to the class $A$ so that the class relationships are equivalent to those in Fig. 4.17.
Figure 4.18 Mutually-Exclusive Classes

This third issue is significant to any object-oriented programming language with multiple inheritance, but this problem has not been generally recognised. The classification-consistency issues are addressed by Hamer (1990).

4.6 Conclusions

The experience of using Class to develop WallBrace has clarified the good and poor points of the language. The object-oriented aspect of Class is more refined than the other aspects, with laziness and classification being a strong feature of the language. However, some abstraction facilities are missing because, for example, it is not possible to abstract out the notion of a recursively-defined sequence.

The single-assignment aspect of the language is weak as functions are not provided. However, the change facility within the Class system is important, and should be retained in the user interface.

The procedural aspects are suitably limited in the language. However, procedures are the only means of producing output from a Class program, which leads to excessive use of procedures.

These criticisms of Class motivate the extensions proposed in the following four chapters. These extensions to Class are integrated cleanly with the existing language, while retaining the valuable aspects.
Chapter Five

Functional Abstraction

5.1 Introduction

Functional extensions to Class are proposed in this chapter and integrated within the object-oriented structure of the language. These extensions provide the language with much of the power of functional languages like Hope (Field and Harrison, 1988).

As discussed in Chapter 4, Class provides good facilities for the representation of structure. However, programs can be unnecessarily repetitive because functional abstraction is not available in a useful form.

The next two sections introduce relevant work (issues and features) in functional and object-oriented programming languages. The proposed functions to be added to Class are modelled on functional languages, but within the context of an object-oriented approach. This leads, for example, to discrimination functions. A new type-loss problem is isolated and a new type-inference system introduced as a solution.

5.2 Functional Programming Languages

A number of relevant ideas and techniques have been associated with functional languages, including pattern matching and the provision of higher-order and polymorphic functions in a statically-typed setting. Other features of functional languages are considered in the following chapter. In the following, Hope (Field and Harrison, 1988) is used to illustrate aspects of pure functional languages which are relevant to the Class extensions.

5.2.1 Pattern Matching

The Hope function doubleList in Fig. 5.1 illustrates the use of "pattern matching" on the list data type which appears as the argument of the function. Two cases are given in the function: one for when the argument is an empty list and the other for a non-empty list. In the latter case, the pattern "head :: tail" is used to isolate the two parts of the data structure,
the head and the rest (or tail) of the list, to be used as separate values in the body of the function.

```haskell
def doubleList: list(num) -> list(num);
   --- doubleList(nil) <= nil;
   --- doubleList(head :: tail) <= (2*head) :: doubleList(tail);
```

Figure 5.1 Pattern Matching

The type of `doubleList` is specified as “list(num) -> list(num)”, a function where the parameter and result type are both “list(num)”. Patterns in Hope may include constants. A number of approaches to executing pattern matching are possible; Hope uses “best-fit” matching, while Standard ML tests the patterns in order (Wikstrom, 1987). A compiler can ensure that patterns are exhaustive (all the cases are considered) and not redundant.

5.2.2 Higher-Order Functions

Functions in Hope can be passed as parameters (giving higher-order functions) and returned as the result of functions. For example, the function `numMap`, as shown in Fig. 5.2, takes a function as argument and applies it to each of the elements of a list, returning a list of the resulting values. The function `doubleList` of Fig. 5.1 may be rewritten using `numMap`, as shown in Fig. 5.2. The anonymous function provided as argument to `numMap` is written as a lambda expression; it returns a number double the number provided as argument.

```haskell
def numMap: (num -> num) # list(num) -> list(num);
   --- numMap(f, nil) <= nil;
   --- numMap(f, head :: tail) <= f(head) :: numMap(tail);
   --- doubleList(aList) <= numMap(lambda x => 2 * x, aList);
```

Figure 5.2 Higher-Order Function
5.2.3 Polymorphic Functions

A polymorphic higher-order function which applies a "folding" function to each element of a list is shown in Fig. 5.3. The type parameters\(^1\) \(alpha\) and \(beta\) may take any type consistent with the declaration. The function is of type \("(alpha \# beta \rightarrow beta)\"\); it takes two parameters with the result being of the same type as the second parameter.

```haskell
dec fold : (alpha # beta -> beta) # beta # list(alpha)
    -> beta;
--- fold(fn, identity, nil) = identity;
--- fold(fn, identity, head :: tail)
    <= fn(head, fold(fn, identity, tail));
```

Figure 5.3 Fold: A Polymorphic Function

The function \(fold\) is used in the function \(sum\) shown in Fig. 5.4, where \("+"\) is a function of type \("(num \# num \rightarrow num)\"\). In this case the function \(fold\) is used as a function of type \("(num \# num \rightarrow num) \# num \# list(num) \rightarrow num\"\).

```haskell
dec sum : list(num) -> num;
--- sum(alist) <= fold(+, 0, alist);
```

Figure 5.4 Using Fold

5.2.4 Lazy Constructors

Data constructors in Hope are lazy, allowing for infinite structures which are evaluated only as much as is needed. Infinite structures in a lazy language allow the termination conditions to be separated from the generator (Hughes, 1989). For example, the Hope function \(from\) in Fig. 5.5 is a generator which can produce an infinite list of integers (adapted from Field and Harrison, 1988, p67).

\(^1\) Called polytypes in Hope.
The "::" list constructor is lazily evaluated and so the recursive function call of `from` is not evaluated until the value is needed. The example function `sum` in Fig. 5.5 sums the first `number` elements of a list. The evaluation of `sum` forces the construction of the list element by element as each is needed; no more of the list is constructed than is necessary. Hence the terminating condition is defined within `sum` and not in the `from` generator. For example, the function application "`sum(10, from(20))`" forces the construction of the first 10 elements of the infinite series 20, 21, 22, ...

5.3 Object-Oriented Languages

The object-oriented nature of `Class` must be taken into account when adding functions to the language. Two important issues arise: the "type loss" and "contravariance" problems. In the following each of the problems is introduced and possible solutions from a number of object-oriented languages are considered. The "contravariance" problem is solved in section 5.5 by extending `Class` with "discrimination functions", a form of pattern matching. An extended form of the type loss problem is introduced in Section 5.6. This extended "type loss" problem is solved with a multi-level type inference system.

5.3.1 The "Type Loss" Problem

A number of "type loss" problems are introduced here, along with some solutions used or proposed by others. We show in Section 5.6 that there are other forms of type loss.

`Class`

A simple form of the "type loss" problem arises with a class that classifies to a number of subclasses, as shown in Fig. 5.6. The expression for the property `i` is type incorrect, even though the condition ensures that the correct class is being referenced. There is no means of handling this problem in the current `Class`. 
Treillis/Owl

Treillis/Owl has a means of testing the type of objects at runtime, using a type-case expression (Schaffert et al, 1986). For example, the operation (function) in Fig. 5.7 from Schaffert et al (1986) adds a Complex number to a Real, where Integer and Real are subtypes of Complex.

```
operation add(me, addend: Complex) returns Complex is
  ! Add an Integer, Real or Complex to a Real.
begin
  type_case addend
    on Integer do
      me + toReal(addend)
    on Real do
      me + addend
    otherwise do
      add(addend, me) ! let Complex code handle it
  end type_case;
end;
```

Multi-Methods in CLOS

This problem is also solved with multi-methods in the Common Lisp Object System, CLOS (Keene, 1989). Multi-methods are similar to the pattern-matching of Hope (Field and Harrison, 1988).

In most object-oriented languages, such as Smalltalk, the method executed depends on the call and on the class of the object sent the message. CLOS provides for multiple dispatch, in which all the arguments of a message are used to choose the appropriate method (Keene,
1989). This is appropriate when the code to be executed depends on the type of more than one object. As a default, CLOS makes the match by considering the arguments from left to right.

For example, with the multi-methods in Fig. 5.8 the code selected depends on both the object to be drawn and the device to be used to display the drawing (from Moon (1989), p67).

```
(defmethod draw ((object triangle) (destination screen)) ..)
(defmethod draw ((object triangle) (destination printer)) ..)
(defmethod draw ((object ellipse) (destination screen)) ..)
(defmethod draw ((object ellipse) (destination printer)) ..)
```

Figure 5.8 Multi-Methods in CLOS

The CLOS methods are written outside of the classes to which they apply. This is in contrast with the approach of Smalltalk (Goldberg and Robson, 1983), Eiffel (Meyer, 1988), and Trellis/Owl (Schaffert et al, 1986), in which functions are included within the class that corresponds to the first argument of the multi-methods in Fig. 5.8.

5.3.2 The "Type Loss" Problem for Functions

Fun

A more complex form of the "type loss" problem occurs with functions. Consider the Fun function `moveX0` in Fig. 5.9, adapted from Cardelli and Wegner (1985, p489). The function takes a value of type `Point` and returns a `Point` with the x value offset by `dx`; i.e. it is of type "Point # Int -> Point". However, if this function is used with a parameter which is a subtype of `Point`, such as `Cursor`, type information is "lost" because the returned value is only known to be of type `Point`.

```
moveX0 = fn(p : Point, dx: Int) p.x := p.x + dx;  p
moveX = all[P <= Point] fun(p: P, dx: Int)
        p.x := p.x + dx;  p
translateX = fn(p: Point, dx: Int) {x=p.x+dx, y=p.y}
```

Figure 5.9 Type Loss Problem

---

2 Using the Hope type notation.
Cardelli and Wegner (1985) handle this problem with bounded parametric polymorphism, as shown by the function `moveX` in Fig. 5.9. The notation “all[P <= Point]” means that the argument to the polymorphic function must be of type `Point` or a subtype of it; the result of the function is of the same type. For example, if `moveX` is called with a `Cursor`, the specific function is of type “Cursor # Int -> Cursor”.

Canning et al (1989) note a corresponding “parameter loss” problem when creating a new object. For example, the function `translateX` in Fig. 5.9 returns a new point with the x value offset from the provided point by `dx`. However, if the function is used with a `Cursor` parameter, the result will be a `Point`. Hence a specific `translateX` function is also required for values of type `Cursor`.

**Eiffel**

Eiffel does not have parametric polymorphism with functions. The type-loss problem is solved by using “declaration by association” as shown in Fig. 5.10. The construct “like Current” means that the type of the function is the same as the type of the current object.

```eiffel
class POINT

... translate(xOffset, yOffset : REAL) : like Current is
  do
    Result.Clone(Current);
    Result.offset(xOffset, yOffset)
  end -- translate;

end -- class POINT
```

**Figure 5.10 Avoiding “Type Loss” in Eiffel**

The body of the function creates a copy (Clone) of the current object (Current) and calls the procedure `offset` to modify the position of the new point, returning the result. This approach thus avoids the “parameter loss” problem.

One problem with this approach is that the programmer has to make a special effort to provide for the inheritance of a class. As shown in section 5.6, there is a more fundamental limitation to “like Current”.

---

3 We show in section 5.6 that it is not a solution.
Canning et al

A similar mechanism to "like Current" in Eiffel is provided in Canning et al (1989, p496), with "myclass", as shown in Fig. 5.11. However, Canning et al (1989) are unclear as to the role of "myclass" in type relationships.

The "parameter loss" problem is solved by a special "translating" mechanism. As shown in Fig. 5.11, the class color_point overrides the new statement inherited from polar_point in order to include the extra Color argument.

```eiffel
class polar_point (x:Real, y: Real)
    ...
    method move(dx: Real, dy: Real) returns Point
        return new myclass(self.x()+dx, self.y()+dy)
end polar_point

class color_point(a:Real, b:Real, c:Color)
    inherits polar_point(a,b)
    translating new myclass(x,y) to
        new myclass(x,y, self.color())
end color_point
```

Figure 5.11 Avoiding "Parameter Loss"

5.3.3 The "Contravariance" Problem

Contravariance is a problem because it prevents subclass relationships to be defined in some cases. The contravariance rule specifies that a function \( f \) of type "AA -> B" is a subtype of function \( g \) of type "A -> BB" (i.e. \( f << g \)\(^4\)) if and only if \( A << AA \) and \( B << BB \). That is, the subtype may "narrow" down the result type but can only "widen" the parameter type.

Eiffel

Cook, W (1989) points out that the contravariance rule is violated in Eiffel (Meyer, 1988), leading to a type problem. The "contravariance" problem can be expressed in Eiffel as shown in Fig. 5.12, in which the class \( B \) has "narrowed" the type of the parameter \( x \) of the inherited function \( f \).

\(^4\) The notation "\( f << g \)" is used to mean that \( f \) is of type \( g \) or a subtype of \( g \).
Consider the function call at the bottom of the program, which leads to the function \( f \) inside the class \( B \) being called. When that function refers to the \( g \) attribute of the parameter an error occurs because the actual parameter is of type \( A \) and has no attribute \( g \).

```plaintext
class A export f
feature
  f(x: A) : Integer is do Result := 0 end;
end

class B export g
inherit A redefine f
feature
  g: Integer;
  f(x: B) : Integer is do Result := x.g end;
end

local
  a : A;
  b : B;
  refA : A;
  i : Integer;
do
  a.Create;
  b.Create;
  refA := b;
  i := refA.f(a);
end
```

Figure 5.12 Contravariance Problem in Eiffel

Cook, W (1989) argues that the contravariance rule must be enforced in Eiffel to avoid this problem, hence preventing the "narrowing" of inherited function parameters.

Canning et al

Canning et al (1989) avoid the contravariance problem by defining separate interfaces (signatures). For example, the interface `ColorPoint` shown in Fig. 5.13 is not a subtype of `Point` because the `equal` function in `ColorPoint` requires a `ColorPoint` parameter.
Their approach is not very satisfactory, because it prevents what appear to be natural supertype relationships from being defined. An alternative approach is discussed for Class in section 5.5, using "discrimination functions"; these are based on pattern matching and the multi-methods of CLOS.

5.4 Adding Functions to Class

Instances and properties can be considered to be functions with no parameters. Class is extended in a straightforward manner by permitting parameters. To be consistent with the rest of the language, function arguments are lazily evaluated, as with object parameters. The types of parameters are defined so that functions can be statically type checked.

This section discusses the addition of simple, higher-order, and polymorphic functions to Class. Integration of functions with the object-oriented aspects of the language is also discussed. Sections 5.5 and 5.6 discuss extensions to solve the problems raised in 5.3; these extensions go beyond the capabilities of the functional and object-oriented languages discussed in 5.2 and 5.3.

5.4.1 Simple Functions

Simple functions, corresponding to Class instances with arguments, provide functional abstraction similar to most conventional languages. For example, the function \texttt{min}, as
shown in Fig. 5.14, takes two arguments of type integer and returns a value of type integer (i.e. it is of type "fn(integer, integer) -> integer").

```plaintext
instance
min(a: integer, b: integer) : integer
:= a if a < b
  | b.

lower : integer
:= min(x, y).
x, y : integer.
```

Figure 5.14 An Example Function

The parameters of a function are local to the body of the function; they cannot be used elsewhere. The parameter and result types of a function may be class types, as shown in the example in Fig. 5.15, and function types, as discussed in section 5.4.3.

### 5.4.2 Functions within Classes

While the value of a property depends only on values available in the class where that property appears, the value of a function within a class can also depend on the function parameters.

**Public Functions**

Public functions provide a convenient solution to the problem discussed in section 4.2.2. The function `overlap` is defined in the class `Range`; it returns the amount of overlap of two ranges, as shown in Fig. 5.15.

---

5 Corresponding to the Hope type "(num # num) -> num"

6 Of type "Range^fn(Range) -> float", where the class of the public function is written in the front with "^".
class Range.
    parameter minimum, distance : float.
    public minimum, distance, maximum, overlap, withinRange.
        maximum : float
            := minimum + distance.
        overlap(other : Range) : float
            := min(other^distance, maximum - other^minimum)
            if withinRange(other^minimum)
            | other^maximum - minimum
            if withinRange(other^maximum)
            | distance
            if other^withinRange(minimum)
            | 0.0.
        withinRange(point : float) : boolean
            := point >= minimum and point <= maximum.
    end Range.

Figure 5.15 Public Functions

The use of the function overlap is illustrated in Fig. 5.16.

distanceObscured : metres
    := xyRange^overlap(xyRangeOther).

Figure 5.16 Using a Public Function

Inheritance and Classification

As with properties, an inherited function may be over-ridden in a subclass. An example of function inheritance is given in Fig. 5.17. Classification is activated when a function is called and where that function also appears in a class which could be classified to. Examples of the interaction of functions and classification are provided in Chapter 6.

The two classes in Fig. 5.17 together define a data structure for a list of integers. The class IntList defines the public functions available and provides the code for the empty list case. The class IntListNode includes the code for the non-empty list case.

This approach differs from that of Hope, in which functions are defined with patterns for the two cases of a list ("nil" and "head :: tail"). The Class code is organised according to the two cases, with two classes each including the functions. The class IntList is the empty list case for two reasons: creating an empty list is simple (just create an object of type IntList); and the non-empty case extends the code of the empty list case.
Functional Abstraction

5.4.3 Higher-Order and Polymorphic Functions

Functions in Class may be higher-order and/or polymorphic, as with Hope. For example, the higher-order function map, as shown in the class IntList in Fig. 5.17, is of type "IntList→fn(fn(integer) → integer) → IntList". The function map is similar to the Hope function in Fig. 5.2; it returns the list of elements of the subject list which satisfy the function keep (a predicate).

Functions may also be polymorphic, such as the function fold in Fig. 5.17, in which the type parameter is written within square brackets following the function name. This function corresponds to the Hope function fold in Fig. 5.3 and is of type

---

7 A refined version of these classes is introduced in Section 5.6.

8 Corresponding (approximately) to the Hope type "list(num) # (num -> num) -> list(num)".
"[Other]IntList(fn(fn(integer, Other) -> Other, Other) -> Other". For example, fold is used in Fig. 5.18 to sum the elements of an IntList, in which the actual type of the function application is "IntList(fn(fn(integer, integer) -> integer), integer)) -> integer". An anonymous function of the form "fn(i) => i + 1" is provided as an argument to fold; this is similar to the Hope lambda expression shown in Fig. 5.2.

\[
\begin{align*}
\text{ints} & : \text{IntList} \\
& \ := \ \text{new} \ \text{cons}(-1) \ \text{cons}(2) . \\
\text{positives} & : \text{IntList} \\
& \ := \ \text{ints} \ \text{filter}(\text{fn}(i) \Rightarrow i > 0) . \\
\text{increment} & : \text{IntList} \\
& \ := \ \text{ints} \ \text{map}(\text{fn}(i) \Rightarrow i + 1) . \\
\text{sum} & : \text{integer} \\
& \ := \ \text{ints} \ \text{fold}[\text{integer}](\text{fn}(i, \ \text{total}) \Rightarrow i + \ \text{total}, 0) .
\end{align*}
\]

**Figure 5.18 Using filter, map, and fold**

Type parameters, such as "Other" in fold, are inferred if they are not specified in the function call. For example, the type of the application of fold in Fig. 5.19 is inferred to be "IntList(fn(fn(integer, integer) -> integer), integer)) -> integer".

### 5.4.4 Lazy Functions

The IntList class can be used to produce lazy finite and infinite lists; a number of example functions are shown in Fig. 5.19.

\[
\begin{align*}
\text{fromTo}(\text{lower}, \text{to} : \text{integer}) & : \text{IntList} \\
& \ := \ \text{new} \ \text{if} \ \text{lower} > \ \text{to} \\
& \quad \ | \quad \ \text{new} \ \text{IntListNode}(\text{lower}, \ \text{fromTo}(\text{lower}+1, \ \text{to})) . \\
\text{from}(\text{here} : \text{integer}) & : \text{IntList} \\
& \ := \ \text{new} \ \text{IntListNode}(\text{here}, \ \text{from}(\text{here}+1)) . \\
\text{factorial}(\text{n} : \text{integer}) & : \text{integer} \\
& \ := \ \text{fromTo}(1, \ \text{n}) \ \text{fold}(\text{fn}(\text{el}, \ \text{tot}) \Rightarrow \text{el} * \ \text{tot}, 1) .
\end{align*}
\]

**Figure 5.19 Use of Lazy Lists**

Functions are not necessarily lazy within a lazy language; care is required when writing functions to ensure laziness. For example, a strict form of the function fromTo from Fig.

---

9 Corresponding to the Hope type "list (alpha) # (num # alpha -> alpha) # alpha -> alpha"
5.19 is shown in Fig. 5.20. It is strict because it needs to produce the tail before adding the head.

```
fromTo2(lower, to: integer) : IntList
  := new if lower > to
  | fromTo2(lower+1, to)^cons(lower).
```

Figure 5.20 Strict Form of Function \textit{fromTo}

5.5 Discrimination Functions

Discrimination functions added to \textit{Class} are an extension of the multi-methods of CLOS (Keene, 1989). In \textit{Class}, however, a discrimination function may be any one of the following: a public function of a class where the "first argument" is implicit; a private function of a class; or a function separate from any classes (corresponding to an instance). In addition, discrimination functions are statically typed and the result type of a discrimination function can be narrowed down in a subclass. Discrimination functions offer the same advantage as multi-methods: the code chosen for execution depends on the type of all the parameters, rather than just the type of the primary object.

The execution of a discrimination function implies finding the appropriate function. This process depends on whether the function appears in a class or not.

5.5.1 Simple Discrimination Functions

At runtime\textsuperscript{10} the functions of the required name are checked in sequence\textsuperscript{11} until one is found in which the types of the actual parameters of the function call match the types of the formal parameters of the particular function. A match of the parameters means that the type of each actual parameter conforms to the type of the formal parameter. Matching may force classification of the actual parameters to occur. As with Hope, a compile-time check is made to ensure that the functions supplied cover all the cases.

\textsuperscript{10} In some cases, the selection of a function can be made at compile-time, as discussed in section 5.5.4.

\textsuperscript{11} The sequence is based on the order of the functions. An alternative is for the \textit{Class} compiler to order them automatically, as in CLOS; this is still to be explored.
For example, the discrimination function *discrim* shown in Fig. 5.21 solves the type-loss problem of section 5.3.1. This corresponds to the way in which multi-methods are defined in CLOS.

<table>
<thead>
<tr>
<th>class A.</th>
</tr>
</thead>
<tbody>
<tr>
<td>classification this : [A1, A2].</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>end A.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>class A1.</th>
</tr>
</thead>
<tbody>
<tr>
<td>generalisation A.</td>
</tr>
<tr>
<td>public b1 : integer.</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>end A1.</td>
</tr>
</tbody>
</table>

```plaintext
a : A := new ....
i : integer := discrim(a).
discrim(a : A2) := a^b2.
discrim(a : A) := 0.
```

Figure 5.21 Classification Type Information Regained

The function *discrim* has the type "fn(A) -> integer", as the most general parameter type is A (A1 and A2 are subtypes). The contravariance rule is not violated because the function can only be called with an argument of type A or a subtype and all cases are covered by the discrimination function. A compile-time error would be given if the first *discrim* function was omitted.

Overloading of functions is not permitted; it must be possible to derive a single type for the set of functions of a discrimination function.12

**5.5.2 Discrimination Functions in Classes**

The functions defined within a class are checked in sequence. If none of these functions are selected, functions in superclasses are tried, based on the partial order of the type lattice, as defined for property rules in Chapter 2.

---

12 Ordinary functions are just singleton discrimination functions.
An example discrimination function \textit{lastOne} is shown in Fig. 5.22, in which a public property \textit{last} is added to the class \textit{IntListNode}. An element of a list is the last one only if the following list element is the empty list; hence a type distinction has to be made.

\begin{verbatim}
class IntListNode.
...
  public last : integer := lastOne(tail).
  lastOne(ntl: IntList) : integer := lastOne(i : IntListNode) := i^last.
end IntListNode.
\end{verbatim}

\textbf{Figure 5.22 Discrimination Function for Function last}

An example of a public discrimination function is provided in Fig. 5.23, in which the function \textit{addList} takes two lists and adds them element by element. If one list is shorter than the other, it is treated as if it has zeroes concatenated on the end.

\begin{verbatim}
class IntList.
...
  addList(other: IntList) : IntList := other.
end IntList.

class IntListNode.
...
  addList(other: IntListNode) :
    := new IntListNode(head + other^head,
                      tail^addList(other^tail)).
  addList(other: IntList) := self.
end IntListNode.
\end{verbatim}

\textbf{Figure 5.23 A Discrimination Function}

This function is used in Fig. 5.24, where the resulting \textit{IntList} is [2, 2].

\begin{verbatim}
a : IntList := new^cons(1)^cons(2).
b : IntList := new^cons(1).
c : IntList := a^addList(b).
\end{verbatim}

\textbf{Figure 5.24 Using addList}
5.5.3 Narrowing the Result Type

A function body for a discrimination function may specify a result type which is a subtype of that given in the definition of the discrimination function; this is called narrowing.

The result type of a discrimination function can only be narrowed if an expression for the new function is also provided; otherwise a type problem arises if a more general value is supplied elsewhere. This arises as a problem in Eiffel, as discussed in Cook, W (1989). To be consistent with the view that functions are properties with parameters, inherited properties may also have their type narrowed. A parameter may also have its type narrowed; an expression is obviously not needed in this case.\(^\text{13}\)

For example, the definition of the function `addList` shown in Fig. 5.23 may be instead written as shown in Fig. 5.25. An extra function is added to class `IntList` which allows for the parameter to be an `IntListNode`.

\begin{verbatim}
class IntList.
  ...
  addList2(other: IntList) : IntList := other.
  addList2(other: IntListNode) : IntListNode := other.
end IntList.

class IntListNode.
  ...
  addList2(other: IntListNode) : IntListNode := new(head * other^head, tail^addList2(other^tail)).
end IntListNode.

one : IntListNode := new(1, new).
two : IntListNode := one^addList2(one).
\end{verbatim}

Figure 5.25 A Discrimination Function

\(\text{13} \) Narrowing of a parameter does not violate the contravariance rule. The type of the actual parameter used when creating an object of that class is constrained to be of the narrowed type.
The function `addList2` has the type "IntList^\text{fn}(IntList) \rightarrow IntList" for objects of type `IntList`.\footnote{See Section 5.6, which discusses the use of more specific type information.} It has the more specific type of "IntListNode^\text{fn}(IntList) \rightarrow IntListNode" for objects of type `IntListNode`. The application of a function can assume a more specific type, as shown in the code at the bottom of Fig. 5.25. The property `two` of type `IntListNode` is assigned a value from a function of type `IntList`. The `Class` type inference system ensures that this expression is type-correct by considering the specific functions in `IntListNode`, the class of the property `one`.

In the some cases, the `Class` system can generate code which avoids the runtime discrimination function selection because it can be determined statically. It would not be possible to do this for the expression in Fig. 5.25, for example, if there were subclasses of `IntListNode`.

### 5.5.4 A Solution to the Contravariance Problem

The contravariance problem of section 5.3.3 is solved simply with discrimination functions. The class `ColorPoint` is made a subtype of `Point`, as shown in Fig. 5.26. The class `ColorPoint` is permitted to have `Point` as a supertype because it does not need to restrict the type of the function `equal`. Instead, it provides explicit code to handle the extra case.

```plaintext
class Point.
public x, y, move, equal.
parameter x, y.
x : float.
y : float.
equal(p: Point) : boolean
  := x = p^x and y = p^y.
move(dx: float, dy: float) : Point
  := new(x + dx, y + dy)
end Point.

class ColorPoint.
generalisation Point.
public color.
parameter color : Color.
equal(p: ColorPoint)
  := x = p^x and y = p^y and color = p^color.
move(dx: float, dy: float) : ColorPoint
  := new(x := x + dx, y := y + dy, color := color).
end ColorPoint.
```

Figure 5.26 Contravariance and Parameter-Loss Solution
The code for the function *equal* in Fig. 5.26 means that the two uses of *equal* in Fig. 5.27 provide the same result.

```plaintext
a : Point := new(x := 0.0, y := 0.0).
b : ColorPoint := new(x := 0.0, y := 0.0, colour := red).
consistent : boolean := a^equal(b) = b^equal(a).
```

*Figure 5.27* Consistency of result from *equal*

A different notion of equality may be required. For example, where points and colorPoints are not equal, the *firmEqual* could be defined as shown in Fig. 5.28, separate from the classes *Point* and *ColorPoint*.

```plaintext
firmEqual(p: ColorPoint, q: ColorPoint) : boolean
p^x = q^x and p^y = q^y and p^colour = q^colour.
firmEqual(p: ColorPoint, q: Point)
false.
firmEqual(p: Point, q: ColorPoint)
false.
firmEqual(p: Point, q: Point)
p^x = q^x and p^y = q^y.
```

*Figure 5.28* Defining *firmEqual* Outside the Classes

### 5.5.5 A (Partial) Type-Loss Solution

The "type loss" and "parameter loss" problems are partially solved in Fig. 5.26. The discrimination function *move* is defined in the class *ColorPoint* with a more specific result type declaration. The type of *move* is "Point^fn(float, float) -> Point" and "ColorPoint^fn(float, float) -> ColorPoint".

Examples of the use of the *move* function are shown in Fig. 5.29.

```plaintext
redPt1 : ColorPoint := new(x := 1.0, y := -1.0, color := red).
redPt2 : ColorPoint := redPt1^move(1.0, 0.0).
pt1 : Point := redPt1^move(1.0, 1.0).
```

*Figure 5.29* Using Function *move*
5.5.6 Pattern Classes

Pattern matching with a discrimination function is weaker than the pattern matching provided by Hope as only the type of the parameters and result may be specified in Class. However, pattern matching is less important than in Hope because the Class code that is executed on a function call depends on the class of the object, giving pattern-matching on the implicit “first argument” of the function.

More general “patterns” can be defined in terms of the generalisation relationships of classes, and through the use of classification properties which optionally classify an object of a class to a “pattern class”. A simple example is shown in Fig. 5.30, in which the discrimination function origin makes use of the classification “property” Origin of the class Point.

```
class Point.
  ...
  classification Origin : boolean
    := x = 0.0 and y = 0.0.
end Point.

class Origin.
end Origin.

origin(Origin) : boolean
  := true.
origin(Point)
  := false.
```

Figure 5.30 Pattern Matching and Classification

The class Origin has been introduced here simply for the pattern match.

5.6 Further Type Loss

An additional type loss problem is isolated which is not solved by the Eiffel notion of “like Current” (Meyer, 1988). An extended form of type inference in Class solves this new problem. This problem does not appear to be handled by the proposals of Canning et al (1989).

A revised version of the IntList family, as shown in Fig. 5.31, introduces the discrimination function newList so that subclasses of IntList may redefine it to avoid the “parameter loss” problem.
class IntList.
public cons, filter, map, fold, append.
  cons(front : integer) : IntListNode
  := newList(front, self).
  filter(keep: fn(integer) -> boolean) : IntList
  := self.
  map(trans: fn(integer) -> integer) : IntList
  := self.
  fold[Other] (accum: fn(integer, Other) -> Other,
    identity: Other) : Other
    := identity.
  append(other: IntList) : IntList
  := other.
  append(other: IntListNode) : IntListNode
  := other.
  newList(head : integer, tail: IntList) : IntListNode
  := new(head := head, tail := tail).
end IntList.

class IntListNode.
generalisation IntList.
parameter head : integer.
tail : IntList.
public head, tail.
  filter(keep)
    := newList(head, tail^filter(keep))
    if keep(head)
    | tail^filter(keep).
  map(trans) : IntListNode
    := newList(trans(head), tail^map(trans)).
  fold[Other] (accum, identity)
    := accum(head, tail^fold[Other] (accum, identity)).
  append(other) : IntListNode
    := newList(head, tail^append(other)).
end IntListNode.

Figure 5.31 Revised IntList Class Family

5.6.1 Type Loss

The functions filter, map, append and cons of class IntList in Fig. 5.31 are too specific in their result types. For example, consider the ExtIntList class family in Fig. 5.32, which inherits from the IntList family as illustrated in Fig. 5.33.
class ExtIntList.
generalisation IntList.
public addList.
    addList(other: IntList) : IntList
        := other.
    newList(head : integer, tail: ExtIntList) : ExtIntListNode
        := new(head := head, tail := tail).
end ExtIntList.

class ExtIntListNode.
generalisation ExtIntList, IntListNode.
tail : ExtIntList. % Discussed in 5.6.2
    addList(other: IntListNode) : IntListNode
        := new(head + other^head, tail^addList(other^tail)).
    addList(other: IntList) : IntListNode
        := self.
end ExtIntListNode.

e1 : ExtIntListNode
    := new(head := 3, tail := new ExtIntList).
e2 : ExtIntList
    := e1^filter(fn(i) => i > 1).

Figure 5.32 Inheriting filter in the ExtIntList Classes

The filter function is defined to have a result type IntList in Fig. 5.31 and so the use of filter in Fig. 5.32 is rejected as type invalid. This is an extension of the type loss problem described in section 5.3.1 and 5.3.2.

![Diagram](image-url)

Figure 5.33 The Relationship between the Classes
To avoid the type-loss problem of the example in Fig. 5.32, the result type of each of the functions should be as follows:

- *cons* should be *ExtIntListNode* when the object is of class *ExtIntList* or *ExtIntListNode*
- *filter* should be *ExtIntList* when the object is of class *ExtIntList* or *ExtIntListNode*
- *map* should be the same type as the object
- *append* should be *ExtIntListNode* when the object is of class *ExtIntListNode* and the same type as the parameter when the object is of type *ExtIntList*

### 5.6.2 Type Inference

For the classes *ExtIntList* and *ExtIntListNode*, the types of the inherited functions can be derived through type inference, as the actual function that will be executed can be isolated statically.\(^\text{15}\)

For the present it is assumed that the class *ExtIntListNode* has the parameter *tail* explicitly narrowed to the type *ExtIntList*, as shown in Fig. 5.32. The reason for this is discussed once the derivation of types has been introduced.

The type derivation of the functions *cons*, *filter*, *map*, and *append* in the classes *ExtIntList* and *ExtIntListNode* is provided informally as follows:

**ExtIntList**

The class *ExtIntList* inherits the functions from *IntList*. The result type of the inherited functions in *ExtIntList* is derived as follows:

- *cons*: "*ExtIntList^A fn(integer) -> ExtIntListNode*". The derivation is as follows:

  ```
  newList(integer, ExtIntList)
  -> ExtIntListNode
  {as the newList in ExtIntList will be used}
  ```

\(^{15}\) This requirement of static isolation sets limits on this form of type inference; these limits are still to be explored.
5. Functional Abstraction

- filter: “ExtIntList^fn(fn(integer) -> boolean) -> ExtIntList”; i.e. the result type is the type of the object (as the object is now of class ExtIntList).

- map: “ExtIntList^fn(fn(integer) -> integer) -> ExtIntList”; i.e. the result type is the type of the object.

- append: There are two cases: “ExtIntList^append(ExtIntListNode) -> ExtIntListNode” and “ExtIntList^append(ExtIntList) -> ExtIntList”. In each case the result type is the type of the parameter. Notice that the contravariance rule is considered here; the type “ExtIntList^append(IntList) -> ExtIntList” is clearly wrong as it would allow an IntList to be turned into an ExtIntList.

**ExtIntListNode**

The class ExtIntListNode inherits these functions from IntListNode because that class is “closer” than IntListNode in the class lattice. The types of each of the inherited functions in ExtIntListNode are derived as follows:

- cons: “ExtIntListNode^fn(integer) -> ExtIntListNode”, as derived from IntListNode (and ExtIntList).

- filter: “ExtIntListNode^fn(fn(integer) -> boolean) -> ExtIntList”, as follows:

  \[
  \text{newList}(\text{integer}, \text{tail}^\text{filter}()) \mid \text{tail}^\text{filter}() \\
  \rightarrow \text{newList}(\text{integer}, \text{ExtIntList}) \mid \text{ExtIntList} \\
  \rightarrow \text{ExtIntListNode} \mid \text{ExtIntList} \\
  \rightarrow \text{ExtIntList}
  \]

- map: “ExtIntListNode^fn(fn(integer) -> integer) -> ExtIntListNode”, as follows:

  \[
  \text{newList}(\text{integer}, \text{tail}^\text{map}()) \\
  \rightarrow \text{newList}(\text{integer}, \text{ExtIntList}) \\
  \rightarrow \text{ExtIntListNode}
  \]

- append: “ExtIntListNode^append(ExtIntList) -> ExtIntListNode”, as follows:  

  \[
  \text{newList}(\text{integer}, \text{tail}^\text{append}(\text{ExtIntList})) \\
  \rightarrow \text{newList}(\text{integer}, \text{ExtIntList}^\text{append}(\text{ExtIntList}))
  \]

---

16 The contravariance rule affects append here as well as in the ExtIntList class.
Problem with tail

As mentioned, there is a problem with the inherited property `tail`. This problem is illustrated with the function `filter`, whose type is derived as: “ExtIntListNode^fn(fn(integer) -> boolean) -> IntList”. The derivation is as follows:

```plaintext
-> newList(integer, tail^filter()) | tail^filter()
-> newList(integer, IntList) | IntList
-> IntListNode | IntList
-> IntList
```

The third step derives the type of the `newList` call as `IntListNode` because the function `newList` in class `ExtIntListNode` will not be selected. The second argument in this case is of type `IntList` and hence the function `newList` in `IntListNode` will be selected instead. The type of the recursive call to `filter` is `IntList` because `tail` is an `IntList` and the `filter` in `IntList` has result type `IntList`.

The problem here is that the class `ExtIntListNode` inherits the parameter `tail` which is of type “-> IntList”; there is no reason for deriving the type “-> ExtIntList” for `tail`. Hence a subclass of `IntListNode` which needs to inherit type-preserving forms of these functions must narrow the type of the inherited property `tail` appropriately.\(^\text{17}\)

5.6.3 Levels of Type Inference

The type inference discussed in Section 5.6.2 is complicated when the type of an application of the discrimination function `newList` has to be inferred, as the type inference system has to consider the dynamic selection of the appropriate discrimination function. Fortunately, this type inference is only done when it is needed, such as with the `ExtIntList` family.

\(^{17}\) An alternative solution is explored in Chapter 9.
It can be considered that there are three levels of type checking of discrimination functions within \textit{Class}, where more specific type information can be derived at each level. These levels are illustrated with the function \textit{map} as follows:

\textbf{Level 1}

The type of a function is simply the type of the most general declaration of the function over all the classes. For example, the result type of both \textit{map} and \textit{filter} in the class \textit{IntList} and its subclasses is (at least) \textit{IntList}.

\textbf{Level 2}

The type of the application of a function can be narrowed down when there is adequate type information about the object to reason about the selection of a particular function body. For example, the result type of the function \textit{map} applied to an object of type \textit{IntListNode} can be derived as \textit{IntListNode} because the function in class \textit{IntListNode} will be selected before the one in class \textit{IntList}.

\textbf{Level 3}

The type of the application of a function can be narrowed down by considering the type of expressions involved. For example, the result type of the function \textit{map} applied to an object of type \textit{ExtIntListNode} can be derived as \textit{ExtIntListNode} by type inference from the expression in the function \textit{map} in class \textit{IntListNode}.

The level of inference required depends on the particular function applications used within a program. Hence a lazy type checking system is appropriate, in which the level of type inference used depends on what is required. As function applications are checked, appropriate type inference is carried out, but only as much as necessary. The examples in later chapters of this thesis do not make use of level 3 type checking.

\textbf{5.6.4 Mixing Types in Append}

The Class type system described in the previous section allows for an object of type \textit{IntList} to be appended to an object of type \textit{ExtIntListNode}, resulting in a value of type \textit{IntListNode}. Consider the example in Fig. 5.34.
The type of the function application "eln^append(list)" is derived at compile-time as follows:

- the function `append` in `ExtIntListNode` is of type "`ExtIntListNode^append(ExtIntList) -> ExtIntListNode`". This would not be selected at runtime, because `list` is not an `ExtIntList`. The function `append` in `IntListNode` is considered next (before the one inherited from `IntList` through `ExtIntList` because of the order of inheritance defined by `Class`).

- the function `append` in `IntListNode` is of type "`IntListNode^append(IntList) -> IntListNode`". As this would be selected at runtime (both the object and parameters conform), the result type of the original application is `IntListNode`.

Two classes with a common supertype can also be mixed. For example, consider the introduction of `RedIntList` and `RedIntListNode`, as shown in Fig. 5.35 (where the notation is extended to show the relationships between the families).
The result types of mixing some of these are as shown in Fig. 5.36.

<table>
<thead>
<tr>
<th>append</th>
<th>object-type * parameter-type</th>
<th>-&gt; result-type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RedIntList * ExtIntList</td>
<td>-&gt; IntList</td>
</tr>
<tr>
<td></td>
<td>ExtIntList * ExtIntListNode</td>
<td>-&gt; ExtIntListNode</td>
</tr>
<tr>
<td></td>
<td>ExtIntList * IntList</td>
<td>-&gt; IntList</td>
</tr>
<tr>
<td></td>
<td>ExtIntList * RedIntList</td>
<td>-&gt; ExtIntListNode</td>
</tr>
</tbody>
</table>

For example, consider the application of “ExtIntList^append(RedIntListNode)”. The append function of ExtIntList does not permit an argument of RedIntListNode. The next function to be considered is the one inherited from IntList, which is of type “IntList^append(IntListNode) -> IntListNode”. This would be selected, and hence the result type of the original expression is IntListNode.
It is possible to collect only the ExtIntList elements from a IntList list, as discussed in Chapter 6.

These results are related to the work of Ait-Kaci and Nasr (1986), who describe LOGIN, a system which extend unification with inheritance.

### 5.6.5 "like Current" is Inadequate

The notion of "like Current" (the class of the current object) in Eiffel (Meyer, 1988) is inadequate. This is demonstrated in Fig. 5.34 in which the declaration of the filter function in IntList is changed: the result type of the function is now constrained to be IntList or a subclass of IntList. The type of filter in IntListNode becomes "IntListNode^fn(fn(integer) -> boolean) -> IntListNode", leading to the expression for b in Fig. 5.34 being accepted as type valid.

```
class IntList.
...  
  filter(keep: fn(integer) -> boolean) : like Current
      := self.
  end IntList.

a : IntListNode
   := new(1, new).
b : IntListNode                % Wrongly accepted as type
   := a^filter(fn(i) => i > 1). % valid

Figure 5.37 A Problem with "like Current"
```

The form "like Current" is only appropriate for the function map, which has a result type the same as the type of the object.

### 5.7 Summary

The need for functional features in Class was made clear from the work on WallBrace, as discussed in Chapter 4. The functional abstraction provided in Hope (Field and Harrison, 1988), a representative functional language, was considered. Pattern matching, higher-order and polymorphic functions, and lazy constructors were discussed as being relevant to providing functional abstraction in Class.

Two problems arise when considering functions within an object-oriented language: "type loss" and violation of the contravariance rule. An outline of solutions to these problems in
other languages was provided, and a parallel drawn between the pattern matching of Hope and the multi-methods of CLOS (Keene, 1989).

Functions were added to Class which may be higher-order and polymorphic, as with Hope. These were integrated with classes and extended to discrimination functions. These functions provide much of the power of the pattern matching of Hope and the multi-methods of CLOS. Unlike the multi-methods of CLOS, however, discrimination functions in Class are statically typed. A type inference system was introduced with three levels of type derivation; this type system solves a new “type-loss” problem which arises from class families.

These functional abstractions go well beyond the immediate needs for functions in WallBrace. Along with the generic class extensions that are introduced in Chapter 6, much of the capability of Hope is provided.
Chapter Six

Generic Classes

6.1 Introduction

Generic classes are proposed for Class to provide additional capability for abstraction within the language. The need for generic classes was made apparent in Chapter 4: For example, sequences, rather than sets or bags, were often needed in WallBrace. In some cases it was necessary to use recursive classes to provide for sequencing, but it was not possible to abstract out the common, sequential aspects.

Because sets and bags are built into the current language, it is not possible to write additional higher-order functions to operate on these structures. In addition, sets and bags are not constructed in a lazy manner. Generic classes will allow for general-purpose data structures such as sets and bags to be written as classes in the language.

Relevant aspects of functional and object-oriented programming languages are considered before discussing the proposed extensions to Class. The extensions are similar to the generic class facilities of Eiffel (Meyer, 1988). Lazy lists are introduced as an example generic class family, followed by generators, which relate generic classes and classification.

The language resulting from the proposed extensions successfully integrates the facilities of the functional and object-oriented languages; this is the major contribution of this Chapter.

6.2 Functional Languages

Polymorphic and abstract data types are two features of functional languages that are relevant to the addition of generic classes to Class.

6.2.1 Polymorphic Data Types

Polymorphic data types may be defined in Hope, as illustrated in Fig. 6.1 (adapted from Field and Harrison, 1988). Pattern matching is used in the function toList to discriminate between each of the data constructors defined for the data type.
data tree(alpha) \equiv empty ++ leaf(alpha) ++
node(alpha # tree(alpha) # tree(alpha));

dec toList: tree(alpha) \rightarrow list(alpha);
--- toList(empty) <= nil;
--- toList(leaf(value)) <= [value];
--- toList(node(value, left, right)) <=
toList(left) <= [value] <= toList(right);

dec IsEqualTree: tree # tree \rightarrow \text{trueval};
--- IsEqualTree(empty, empty) <= true;
--- IsEqualTree(leaf(m), leaf(n)) <= n = m;
--- IsEqualTree(node(l1,v1,r1), node(l2,v2,r2)) <=
  (v1 = v2) and IsEqualTree(l1,l2) and IsEqualTree(r1,r2);
--- IsEqualTree(t1, t2) <= false;

Figure 6.1 User-Defined Data Types

The program is organised around the functions that may be applied to values of the data type. Pattern matching is a convenient means of handling each of the cases of the data type; a compiler can also check that all the cases have been considered. This is in marked contrast to the object-oriented approach taken in Class, in which the program is organised around the data types and there is less emphasis on pattern-matching.

6.2.2 Abstract Data Types

It is not always appropriate to "expose" the implementation of a data type, as in Fig. 6.1. Abstract data types can be defined in Hope and other functional languages in order to hide away the implementation details of a data structure. For example, a set type in Standard ML is implemented with lists, as shown in Fig. 6.2 (adapted from Wikstom, 1987, p307).¹

¹ The type variable 'a serves the same purpose as alpha in Hope.
The set type offers an interface with the constructors *Empty* and *Insert* for creating sets, and the functions *member*, *union*, *intersection*, and *eqset*. The set type is implemented here by using lists, but this is hidden away within the abstract data type.

### 6.3 Object-Oriented Languages

Generic classes provide a form of parametric polymorphism in object-oriented languages. Bounded polymorphism is a specialised form of generic class in which there is an upper bound specified for the type parameter.

#### 6.3.1 Parametric Polymorphism in Classes

A number of object-oriented languages have generic classes. Examples from Eiffel are used to demonstrate the use of generic classes. Trellis/Owl provides for parametric polymorphism in a similar form (Schaffert et al, 1986).

A generic class in Eiffel has type variables (Meyer, 1988). For example, an Eiffel generic class for sets with the type variable $T$ is shown in Fig. 6.3. The type variable is used in the declaration of the parameter *element* of the procedure *insert*. Because nothing is known about the type of *element* in general, it may only be assigned to a variable of the same type (i.e. $T$), passed as a parameter, and tested for equality with a variable of the same type.²

```
class SET [T] export
    insert, member, union, intersection, eqset
feature
    insert (element : T) is
        -- Add the element to the set
        do ... end;
...
end -- class SET
```

**Figure 6.3** Generic Class SET

A variable may be declared to be of the type SET, and used as shown in Fig. 6.4. The actual generic parameter INTEGER is part of the type declaration of *intSet*; all types are resolved statically. This means, for example, that "intSet.insert(true)" is not type-correct.

---

² Functions are not "first-class objects" in Eiffel, so an equality test for all types is acceptable.
Inheritance is somewhat limited in Eiffel: a class must specify types for every type variable in its superclass(es).

### 6.3.2 Bounded Parametric Polymorphism

Type variables in Eiffel (Meyer, 1988) may be constrained to be a subclass of a specified class, as illustrated in Fig. 6.5, in which the type variable `T` is restricted to be a subclass of `RING`. Variables of type `T`, such as `el` in Fig. 6.5, may use the publics of class `RING` (such as the function `plus`).

```eiffel
class MATRIX [T -> RING] export
...
feature
  add(m : T) is
    -- Add the matrices
    local el : RING;
    do
      loop ...
        el := entry(i, j);
        el.plus(other.entry(i, j));
        entry(i, j, el);
      end;
    ...
  end -- class MATRIX
```

**Figure 6.5 Bounded Type Variables**

A different approach is taken in Ada (DOD, 1981) in which a generic class may require a function of a specified type to be available. Canning et al (1989, p462) take a similar approach, as illustrated in Fig. 6.6.

```eiffel
class Set[T] implements Set[T]
  where T contains equal(T) returns Boolean
...
```

**Figure 6.6 Constraining the Type**
6.3.3 Contravariance and Generic Classes

The contravariance rule also applies to generic classes, limiting the type relationships between classes. As Cook, W (1989) points out, the classes $A[X]$ and $A[Y]$ may be in one of three type relationships, depending on the use of the types $X$ and $Y$ within the generic class $A$. Assuming that $X << Y$, the type relations are as follows:

- $A[X] \ll A[Y]$ if $A$ does not have any functions with a parameter whose type depends on the type parameter of the class $A$;
- $A[Y] \ll A[X]$ if $A$ does not have any functions with results whose type depends on the type parameter;
- Otherwise, there is no type relation.

6.4 Adding Generic Classes to Class

The generic classes proposed for Class are similar to those of Eiffel, with an important difference. In Class, a class may inherit from a generic class without providing types for the type parameters of the superclass. The importance of this difference is demonstrated in sections 6.4.2, for example.

6.4.1 A Simple Example

An example of a generic class is shown in Fig. 6.7. Pair defines 2-tuples in which the type of the two elements is the same. The type parameters are listed within square brackets after the class name. A property $aPair$ is also shown, declared to be of class $Pair[integer]$, a class with two parameters of type integer.

```java
class Pair [Element].
parameter first, second : Element.
pUBLIC first, second.
end Pair.

```

Figure 6.7 Generic Class Example
The type parameter of class Pair can be specified in either of two ways: when a property of some Pair is declared (as in Fig. 6.7) or in a subclass of Pair (as shown in Fig. 6.8).

```plaintext
class Point.
  generalisation Pair[float].
  public translate(dx: float, dy: float) : Point
     := new(first := first + dx, second := second + dy).
end Point.
```

Figure 6.8 Generic Parameters in a Subclass

Strictly speaking, the relationship between the classes Point and Pair in Fig. 6.8 is not generalisation, as the class Point is not a subtype of Pair. The class Pair is a "type-schema" which is instantiated by the specification of the type parameter in the class Point.

### 6.4.2 Subclasses of a Generic Class

A class need not define the type parameters of its generalisation class(es). For example, consider the generic class family for the classic stack shown in Fig. 6.9, in which the class StackNode simply inherits the type variable Element.

In Eiffel, a subclass is required to provide types for inherited type parameters (Meyer, 1988). This is not a restriction in Class; it would prevent class PopableStack from being defined as a subclass of Stack.

```plaintext
class Stack[Element].
  public push.
     push(el : Element) : PopableStack[Element]
        := new(top := el, pop := self).
  end Stack.

class PopableStack.
  generalisation Stack.
  parameter top : Element.
  pop : Stack[Element].
  public top, pop.
end PopableStack.
```

Figure 6.9 Generic Class Stack and its Subclass

The function push within the class Stack is polymorphic, based implicitly on the type parameter of the class.
An example integer stack is shown in Fig. 6.10. A discrimination function is needed to pop an element off the resulting stack. While the test for whether it is valid to pop an element off `intStack` occurs at runtime, the discrimination functions can ensure type-correctness statically.

```
intStack : Stack[integer]
    := new^push(3)^push(4).
  top : integer
    := getTop(intStack).
getTop(s: PopableStack) : integer := s^top.
getTop(s : Stack) := 0.
```

Figure 6.10 An Integer Stack

### 6.5 An Extended Example: Lazy Lists

The generic class `List` shown in Fig. 6.11 generalises the class `IntList` from the previous chapter and illustrates the use of generic classes to build general data structures. The class `ConsList` is introduced here to construct lists; it defines the inherited properties `head` and `tail` as parameters. This allows for lists to be constructed in other ways, as illustrated in Section 6.7.

The higher-order functions provided in `List` are generalisations of those in the class `IntList` in the previous chapter. The function `map` has a type variable to allow for the `trans` function of `map` to take an value or object of type `Element` and produce a value of some other type. This is illustrated in Fig. 6.13, in which `map` is used to "collect" one public property from each of the `Pair[integer]` objects in `aList`. 
class List[Element].
public cons, filter, map, fold, append, any, all.
cons(front : Element) : List[Element]
  := newList(front, self).
filter(keep: fn(Element) -> boolean) : List[Element]
  := self.
map[Other](trans: fn(Element) -> Other): List[Other]
  := self.
fold[Other](accum: fn(Element, Other) -> Other,
  identity: Other) : Other
  := identity.
append(other: ListNode[Element]) : ListNode[Element]
  := other.
append(other: List[Element]) : List[Element]
  := other.
any(pred: fn(Element) -> boolean) : boolean
  := false.
all(pred: fn(Element) -> boolean) : boolean
  := true.
newList(head : Element, tail: List[Element]) : ConsList
  := newList(head, tail).
end List.

class ListNode.
generalisation List.
published last, head, tail.
  head : Element.
  tail : List[Element].
  last : Element
    := lastOne(tail).
  lastOne(List[Element]) : Element.
    lastOne(l: ListNode[Element]):= l^last.
    lastOne(l: List[Element]):= head.
filter(keep)
  := newList(head, tail^filter(keep)) if keep(head)
    | tail^filter(keep).
map[Other](trans)
  := newList(trans(head), tail^map(trans)).
fold[Other](accum, identity)
  := accum(head, tail^fold(accum, identity)).
append(other) : ListNode[Element]
  := newList(head, tail^append(other)).
any(pred)
  := pred(head) or tail^any(pred).
all(pred)
  := pred(head) and tail^all(pred).
end ListNode.

class ConsList.
generalisation ListNode.
published head, tail.
end ConsList.

Figure 6.11 Lazy Lists
6.5.1 Set/Bag Functions Replaced by List Functions

Collect and Select

The functions map and filter in the class List can provide the capability of the set/bag functions collect and select. For example, the expressions shown in Fig. 6.12 which use the set functions can be rewritten using lazy lists as shown in Fig. 6.13.

```
pairSet : set IntegerPair.
intSet1 : set integer := collect(i in pairSet, i^first).
intSet2 : set IntegerPair := select(i in pairSet where i^first > 3).
```

**Figure 6.12 Collect and Select**

```
apList : List[Pair[integer]].
intList : List[integer] := aList^map(fn(i) => i^first). % collect
intList2 : List[Pair[integer]] := aList^filter(fn(i) => i^first > 3). % select
```

**Figure 6.13 Collect with Map and Select with Filter**

The selection criterion of the filter function cannot be based directly on the type of the elements, but this can be simulated with the function fold. As shown in Fig. 6.14, fold is used with a discrimination function to select those elements of a list of Point which are of a particular subclass, Cursor. This works by mapping non-Cursors to empty lists of Cursor and concatenating all the lists together.

```
points : List[Point] := ....
cursors : List[Cursor] := points^fold(fn(el, cursors) =>
                             cursors^append(isCursor(el)),
                             new Cursor).
isCursor(c: Cursor) : List[Cursor] := new ConsList(c, new).
isCursor(p: Point) := new. % it was not a Cursor
```

**Figure 6.14 Filtering Subclasses**
Create

A general function for creating lists of objects is shown in Fig. 6.15, based on the function \textit{fromTo} provided in section 6.4.4.

\begin{verbatim}
create[X](total : integer, f: fn(integer) -> X) : List[X] := fromTo(1, total)^map(f).

walls : List[Wall] := create(totalWalls, fn(w) => new(wallNumber := w, ...)).
\end{verbatim}

Figure 6.15 \textit{Create from Map}

6.6 Bounded Parametric Polymorphism

\textit{Class} also provides for bounded parametric polymorphism.

6.6.1 Sets and Bounded Polymorphism

The generic class \textit{Set} shown in Fig. 6.16 illustrates the use of bounded parametric polymorphism. The notation “[Element <: Comparable]” specifies that the type parameter of class \textit{Set} is \textit{Comparable}, which provides a set of relations as public functions, including the function \textit{eq}, which is needed to test whether an element already appears in a set.
class Set [Element <: Comparable].
public insert, member, union, intersection.
insert(el : Element) : PrivateSet[Element] := new(rep := new^cons(el)).
member(el : Element) : boolean := false.
union(s : PrivateSet[Element]) : PrivateSet[Element] := s.
end Set.

class PrivateSet.
generalisation Set.
public rep.
parameter rep : List[Element].
insert(el) := self if member(el) |
new(rep := list(el, self)).
member(el) := rep^any(el^eq).
union(s) := new(rep := rep^append(
s^rep^filter(fn(el) => not member(el))).
intersection(s) := new(rep := s^rep^filter(member)).
end PrivateSet.

Figure 6.16 Set Family

The class Comparable provides the public function eq, among others, as shown in Fig. 6.17.

class Comparable.
public eq, neq, less, greater, lessOrEq, greaterOrEq.
% Only eq and less need to be defined in subclasses
eq(c : Comparable) : boolean := false. % by default
neq(c : Comparable) : boolean := not eq(c).
less(c : Comparable) : boolean := true. % by default
greater(c : Comparable) : boolean := not lessOrEq(c).
lessOrEq(c : Comparable) : boolean := less(c) or eq(c).
greaterOrEq(c : Comparable) : boolean := not less(c).
end Comparable

class Integer.
generalisation Comparable.
public int.
parameter int : integer.
eq(i : Integer) := int = i^int.
less(i : Integer) := int < i^int.
less(r : Real) := int < Round(r^real).
end Integer.

Figure 6.17 Class Comparable
In order to provide the same encapsulation properties as the Standard ML example of Fig. 6.2, it would be necessary to make the class PrivateSet unavailable to programmers using the class Set. This requires a form of encapsulation above the level of classes, as discussed in Chapter 9.

### 6.6.2 Inheritance with Bounded Polymorphism

The contravariance rule also applies to bounded type parameters. The example shown in Figure 6.18 is invalid. This is corrected in Fig. 6.19 with the use of a discrimination function.

```plaintext
class A[P <: A].
  public f(x: P): integer := 0.
end A.

class B.
  generalisation A[B].
  public a: integer.
  f(x) := x^a.
end B.

...
display(m^f(new A[A])).
```

Figure 6.18 Invalid Inheritance

```plaintext
class A[P <: A].
  public f(x: P): integer := 0.
end A.

class B.
  generalisation A[B].
  public a: integer.
  f(x: B) := x^a.
end B.

...
display(m^f(new A[A])).
```

Figure 6.19 Valid Inheritance with a Discrimination Function
6.7 Generators

A list generator creates lists of things, based either on object parameters or on input from outside the program (such as from the user). A generator could be used in WallBrace, for example, to lay out the bracing lines in a building.

The classes shown in Fig. 6.20 illustrate the use of inheritance to build a simple generator.

```java
class AskGenerator [Element].
generalisation List[Element].
parameter first : boolean.
classification AskGeneratorNode : boolean
  := ask('Are there any elements? ') if first
     | ask('Are there any more elements? ').
end AskGenerator.

class AskGeneratorNode.
generalisation AskGenerator, GeneratorNode.
  head := ask('What is the first element? ') if first
         | ask('What is the next element? ').
  tail := new AskGeneratorNode(first := false).
end AskGeneratorNode.
```

Figure 6.20 The AskGenerator Class Family

The use of this generator is illustrated in Fig. 6.21. When sum is evaluated, the call to filter causes the evaluation of numbers. This in turn causes an AskGenerator object to be created. As definition of filter appears within a potential classified-to class, classification is initiated. The classification property is tested, leading to the first question of the user: “Are there any elements?” If the answer is “yes”, the classification occurs and filter can access the head of the list to test whether it is accepted by the filter predicate. Assuming it is acceptable, filter calls the function newList which lazily creates a new ConsList. This is then available for fold to start processing. This continues, with the lists (the original and the one passed on by filter) being created until the user says there are no more elements. The function fold gets to the end of the list and totals the elements, returning a value which is assigned to the instance sum. The sum is then displayed.
numbers : AskGenerator[integer]
    := new(first := true).
sum : integer
    := numbers^filter(fn(i) => i > 0)^
        fold(fn(e1, tot) => e1 + tot, 0).
procedure main.
    display('The sum of the positive integers is ', sum).
end main.

Figure 6.21 An Example of the use of AskGenerator

The relationships between the classes is shown in Fig. 6.22.

Figure 6.22 The Class Relationship

6.8 Generic Classification

Generic parameters of a generic class can also affect classification, as demonstrated by the
generic class Generator shown in Fig. 6.23. This class is an abstraction of the generator
aspect of the class family AskGenerator in Section 6.7. The actual classification property in
the class Generator is defined in a subclass through the type parameter; it must be a subclass
of GeneratorNode. The type parameter is bounded to be a subtype of GeneratorNode to
ensure type-correctness of the classification.
The second type parameter of class *Generator* specifies the type of the element of the list.

The use of the Generator family is illustrated in the class family *AskGenerator* in Fig. 6.24, which acts in the same manner as the class family *AskGenerator* in Fig. 6.20. As class *AskGenerator* specifies that the type parameter *Node* of class *Generator* is to be *AskGeneratorNode*, classification of an object of this class will be to *AskGeneratorNode*. The second type parameter of *Generator* is defined to be *integer* so a *List[integer]* results from the generator.

This approach is a rather awkward way to abstract out the generator aspects of the *AskGenerator* class family; a simpler approach is suggested in Chapter 9.
6.9 Summary

The language resulting from these extensions successfully integrates the facilities of the functional and object-oriented languages discussed in Sections 6.2 and 6.3.

Extending *Class* with generic classes means that user-defined data types of functional languages can be written in an object-oriented style, as illustrated by the *List* class family. A functional style is also possible; for example, the Hope data type *tree* shown in Fig. 6.1 could be encoded as shown in Fig. 6.25.

```
class Tree[Element].
  end Tree.
class EmptyTree.
  generalisation Tree.
  end EmptyTree.
class TreeLeaf.
  generalisation Tree.
  parameter value : Element.
class TreeNode.
  generalisation Tree.
  parameter left : Tree.
  parameter value : Element.
  parameter right : Tree.
  end TreeNode

toList[Element](t: EmptyTree[Element]) : List[Element]
  := new.
toList[Element](t: TreeLeaf[Element]) : ListNode[Element]
  := new(t, new).
toList[Element](t: TreeNode[Element]) : ListNode[Element]
  := toList(t^left)^cons(t^value)^append(toList(t^right)).
```

Figure 6.25 Class *Tree* Written in a Functional Style

Abstract data types can be defined in *Class*, although the encapsulation of the Standard ML example of Fig. 6.2 is not provided. For example, the *Private Set* class in Section 6.6.1 is still available. This issue is discussed in Chapter 9.

Further examples of the use and value of these extensions are provided in Chapter 8.
Chapter Seven

A Forms Interface

7.1 Introduction

As discussed in Chapter 4, the user interface to Class is inadequate and there is no facility for access to files and databases. A general-purpose forms interface is proposed which solves a number of the user-interface problems. Other problems with the user interface, such as plan entry, are to be addressed by specialist interfaces (Hosking, 1989; Mugridge et al, 1989).

Section 7.2 considers various approaches to I/O and referential transparency in functional languages. A "set of services" approach is then considered. Duda et al (1987) provide a model which is suited to Class. This is followed by a discussion of a form-filling, graphics-based user interface proposed for Class by Clausen (1989). This proposal fails to solve the "user control" problem and is lacking in some essential details, such as the handling of monotonicity and integration with the execution of Class.

The proposed forms interface is introduced by example, starting first with simple forms before illustrating form elaboration ("visual classification") and variable-length tables.

This chapter also uses the forms interface to focus on the needs of a general-purpose external interface to Class that will allow Class programs to be used with user-interface systems, external programs, files, and databases. A critical issue in providing an external interface is to retain referential transparency and the associated change facility of Class, while integrating the interface cleanly with the object-oriented and procedural aspects of the language. This requires a careful separation of the responsibilities of Class and any external code.

7.2 Functional Languages and I/O

Functional languages have some difficulties in resolving the incompatibility between the notion of referential transparency within the language and the program having effects.
through input and output (I/O). There have been a number of approaches to handling I/O and the related issue of sequence in functional languages.

7.2.1 Streams: Miranda

A stream in Miranda\(^1\) is a lazy list of characters. Dwelly (1988) discusses the construction of interactive programs in Miranda (Turner, 1986), a lazy functional language. A Miranda program returns a stream (the output) and accepts as argument an input stream. Dwelly is concerned with the manner in which input and output can be synchronised so as to be sensible to the user. Dwelly uses the Miranda program in Fig. 7.1 to illustrate the problem of all the prompts appearing immediately.

```ml
sum inputStream = 
  "Type a number" ++
  "Type another number" ++
  "The result is" ++
  numToChars(n1 + n2)
where
  (n1, rest) = charsToNum inputStream
  (n2, tail) = charsToNum rest

Figure 7.1 Synchronising Streams in Miranda
```

Dwelly's solution is to build feedback into the program in order to ensure appropriate synchronisation. This is done by requiring the program itself to echo characters typed by the user, rather than leaving this to the operating system.

7.2.2 Procedures Added: Standard ML

In Standard ML, referential transparency is abandoned when it comes to I/O; imperative features are provided in which "functions" have side-effects. For example, the function `echo` in Fig. 7.2 (from Wikstrom, 1987, p322) displays entered text. It is executed for its side-effects rather than for its result (as made clear by the empty parameter and the empty result of the function).

```ml
fun echo () = if lookahead std_in = "." then ()
  else (output (std_out, input (std_in, 1));
       echo());

Figure 7.2 I/O in Standard ML
```

\(^1\) Miranda is a trademark of Research Software Ltd.
This was the approach first used in Class, but it was soon rejected in favour of a completely separate procedural form. This allows the single-assignment part of the language to be kept pure referentially.

### 7.2.3 Result Continuations: Hope+C

A program in Hope+C is a function which returns a “result continuation”; this includes a request for I/O (or other operating system service) and a continuation function which is to be called on the completion of the I/O (Perry, 1989). Once the I/O request has been handled by the operating system, the continuation function is called (as a new program) with results of the request. Hence the continuation function encodes the state of execution, and the sequence of continuation calls define the sequence. Thus referential transparency is retained within a single execution of a program.

### 7.3 Providing a Set of Services

The programs-as-functions model of the functional programming languages is fine when a program is expected to provide a single result. However, when a range of services are to be provided to the user, this model is less appropriate.

#### 7.3.1 A Program as a Set of Services

The current execution model of Class assumes that there is a single focus of interaction with the user. A better model for what is required in a user interface is that of an extended spreadsheet in which a program offers a set of services to the user, who is able to select from those services and decide what information to supply next within a set of possible input fields.

This “set of services” model is the view of Meyer (1988) with regard to software design. He argues that a functional (or top-down) approach is often problematic: “Top-down methods assume that every system may be properly described, at the most abstract level, by its main function”, but that “… practical software systems are more appropriately described as offering a number of services” (p47).

The functional approach is not particularly suitable here because the user may enter data in any one of several fields. While it is possible to write such a system in a functional language, it would need to interpret the user’s requests and inputs and act on them, thus
simulating the model required. Syntel, as introduced next, is closer to what is required for Class (Duda et al, 1987).

7.3.2 Syntel

Syntel is an application system in which a data-driven functional language is interfaced to a separate form-filling user-interface system (Duda et al, 1987). Syntel avoids the problem of the captive user, "... in which systems ask long series of questions that users must patiently answer ...". Duda et al (1987, p19) argue that "... frequent users must be able to control the sequence in which things are (or are not) done".

A mapping is specified between variables in a functional program and fields in screens (forms). The form-filling interface is defined independently with a "forms language". Changes to input values on the screen cause output values on the form to be recomputed, so that "... Syntel behaves like a spreadsheet program, propagating variable values through an equation network" (ibid, p23).

The forms system of Syntel is non-procedural and allows programmers to "... define any number of screens, providing viewports into selected parts of the knowledge base .... permitting different ways of grouping variables ..."). The language is used to specify "... (1) the hierarchical organisation of the screens, (2) the layout and format of data items within any given screen, and (3) the association between these data items and Syntel variables".

Where a screen (or region of a screen) is only sometimes relevant, it is possible to specify that it is visible conditional on the values of variables. "As users enter information, additional viewports into the knowledge base become enabled. Users have complete freedom to move to any enabled screen and to enter or change any displayed item on an enabled screen, but are restricted to what is currently visible".

Data propagation needs to be restricted in order to give a good response to the user. Compile-time analysis determines all the screens that a variable can affect, and "... changes are propagated through the equation network as long as and only as long as the current screen is affected" (p26).

The Syntel work assumes a non-procedural interface, although an exception is made in allowing the user to request explanation information.

Syntel is compared with the proposed forms interface in section 7.6.
7.3.3 Clausen’s Proposal

Clausen (1989) proposes a preliminary forms-based user interface to *Class* to enhance the quality of the user interface. As with Syntel, Clausen’s design allows for multiple forms appearing as independent windows on the screen. Forms have the following features:

- static text and graphics
- menus and radio boxes are available for input and output
- multiple input and output text fields, with input validation
- buttons for requesting help and other information
- buttons for traversing the current set of forms, allowing the user to change previous inputs in place

The work of Clausen provides the first step in the definition of a forms interface to *Class*, but a number of issues remain.

*User Control*

Clausen partially solves the “sequencing” problem, as the user may enter data in those forms that are visible on the screen. By grouping together input and output fields into a single form, the user is able to answer the questions in any order. However, this provides only local control to the user, within the context of currently available forms. Clausen proposes that a program begins “... by asking the user to identify which of a number of results that they wish to obtain”, so affecting the execution order (Clausen, 1989, p43).

However, this is not very satisfactory. As a *Class* program retains complete control of the course of execution, forms would appear when input values are needed by the running program. The user could only provide data in the currently visible forms, and would be unable to direct the flow of control away from that prescribed by the *Class* program.

*Tabular Information*

Clausen makes no provision for tables in forms, although tables are likely to be useful, such as in the display of information about each of the bracing elements allocated on a wall in *WallBrace*. 
Form Traversal and Help Information

Clausen proposes that the user may move (traverse) between existing forms, but he does not specify how this could be done: "Users could peruse previous cards [forms] in a sequential fashion by using 'buttons' on cards or leap immediately to any given card by selecting the icon for that card" (Clausen, 1989, p48). However, this simple organisation is inappropriate when modelling a complex structure like a building with many components. For example, in a system like WallBrace (Mugridge and Hosking, 1988) it would be better to access the walls within a storey of a building via a form representing the storey.

Clausen also proposes that 'buttons' be provided for the user to request help information. He suggests that an independent system could use data from Class to provide information to the user, but does not specify a mechanism for achieving this.

A general hypertext capability (Aks cyn et al, 1988) would be desirable for traversal between forms. Hypertext (or hypermedia) provides for pieces of information, such as cards in Hypertalk (Apple, 1988), to be linked together in arbitrary ways so that a user can move between them in order to follow topics of interest, for example.

7.4 An Extended Forms Interface

A forms interface is proposed that solves the problems with Clausen's preliminary proposal. The forms interface is introduced through examples which provide solutions to these problems. In addition, these examples are used to isolate the important attributes of an external interface to Class. Further details of the forms interface are provided in Mugridge (1989).

7.4.1 Forms and Widgets

Forms include widgets, which are general, active graphical objects.2 Widgets that are used in the forms interface include:

- menus (pull-down and pop-up) and radio boxes for making selections between several choices, corresponding to enumerated types.
- toggle buttons for selecting an option, corresponding to booleans.

---

2 The visual structure of a form is defined separately from the Class program.
7. A Forms Interface

- scales, selection boxes, and simple text for the entry of integer and float values with range checking of numbers.
- simple text for text entry and display.
- radio boxes, toggle buttons, and scales may also be used for output.
- scrolling windows for variable amounts of information.
- push-button widgets for requesting some action or information.

An example form for data entry for a diagonal brace is shown in Fig. 7.3. This form consists of five main areas. The top area provides information about the bracing element being considered, and is supplied by the program. The next two areas below contain input options, based on radio boxes. The user clicks on one of the radio boxes to select that option. The next area down contains two input fields for the entry of length information. The user selects an input field by clicking on the text box before typing.

![Example Form](image.png)

**Figure 7.3 An Example Form**

The bottom-most area contains two output fields and a help button. The user may click on the "Help" button for some help information, as shown in Fig. 7.4.
7.4.2 Property Values: Inputs and Outputs

When the form is first displayed, the default options of the two sets of radio boxes are selected. As the user selects other options and enters length information, the output field value for “B.U.’s of Bracing Provided” changes appropriately, providing immediate feedback to the user.

The user may change any input value at any time. Outputs that are affected by a changed input are recomputed and redisplayed. The user has to be informed if a prompt that appears on a form is affected by a change. One approach is to “ghost” the form on the screen to signify to the user that the form needs to be checked. Forms may also be “destroyed” by a change, such as when a user reduces the number of storeys in a building from three to two, and “reappear” if the change is reversed later.3

A form corresponds to an object of a class; widgets on the form correspond to properties in Class. There are three sorts of properties used, corresponding to input values, output values, and values which appear as prompts. This distinction has to be made in the external interface in Class, as discussed in the following chapter.

As values are entered or selected on a form, the values “flow” through to the Class properties. When a value is available for a property corresponding to an output widget, the form is updated appropriately. Thus the interface reacts as the user makes changes, requiring careful coordination between the Class system and the forms system.

Simple constraint checking, such as for a positive value, is handled by the forms system, as shown in Fig. 7.4.

3 These issues will be investigated once the forms interface has been built.
Constraints and Errors

For more complex constraints, an error or warning can be provided through a property of type `text` which is mapped to an output field in the form. For example, the value of the property `lengthError`, as shown in Fig. 7.5, depends on the constraint being tested. For example:

```plaintext
private lengthError : text := 'The brace must be at least the minimum length of ' + minLength + ' m.' if planLength < minLength
```

Figure 7.5 An "Error" Property

The full text appears when the constraint is violated. It "disappears", by taking an empty string value, when the constraint is no longer violated. In the example of Fig. 7.5, this
occurs when the user enters a valid value for the input field corresponding to the planLength property.

7.4.3 Procedures and User Control: Buttons

A user may click on a button to request any of a number of actions, as follows:

- to request help or explanation information; a form appears with appropriate information (which may depend on values in the running Class program). An example appears in Fig. 7.4.
- to request a new form to be displayed; the new form appears on the screen, ready for the user to enter information and make other requests.
- to request an existing form to be displayed or exposed on the screen.
- to request a form or group of forms to be temporarily closed.
- to request a file to be created.
- to request that a window of another interface (such as for plan entry) be initiated or exposed on the screen.

These are clearly requests for some action, rather than the entry of some data, and therefore correspond to procedure calls. Each button on a form corresponds to a procedure within the Class program. Buttons and procedures provide for a whole range of actions to occur, unlike the limited actions that are predefined within the Syntel system (as described in section 7.3.2). This leads to a natural way of handling a button that “creates” another form: the procedure simply calls a procedure within the new form, requesting it to display itself; if it doesn’t yet exist, it is created.

Buttons solve the sequencing problem by giving much more control to the user, giving the impression of freely browsing and extending a model of their design, rather than answering a series of questions as dictated by the execution flow of the program. In this way, the user is able to drive the order of execution, deciding at any point to focus on some other part of the application by selecting another form.

The calculation of a value of an output field on a visible form may depend on input from another form that does not yet exist. This dependency causes the second form to be created and displayed to the user. Different blends of control between the program and the user are possible. For example, a naive user can “pass control” to the program by specifying that they are unfamiliar with the package; in this case procedures within the Class program direct
the sequence of execution. On the other hand, an experienced user can “retain control” by using a button to create the next form of interest. Of course, the Class programmer is responsible for ensuring that appropriate buttons, with supporting procedures, are provided to allow this to happen.

The relationship between forms can be structured according to the needs of the application, rather than being based on the order in which the forms appeared. The interface of a program can therefore have some of the qualities of a hypertext system, as discussed in section 7.5.2.

7.4.4 Classification: Form Elaboration

The information on a form may be elaborated as the user enters more information. This is appropriate, for example, when a form provides information about a Class object which can be dynamically classified, as in Fig. 7.6.
The values entered in the form shown in Fig. 7.6 (a) corresponds to the object in Fig. 7.6 (b) which in turn is one part of the *Roof* class family, as shown in Fig. 7.7. It only makes sense to gather information about the ridge if the roof is a gable-end or a hip roof, so the bottom two boxes in the form in Fig. 7.6 only appear if a gable-end or hip roof is selected.

A *Solution to the Height-Slope Problem*

Classification can be used to solve the height-slope constraint problem, in which it is necessary to ask the user whether they wish to enter the slope or the height of a roof. Consider the form shown in Fig. 7.8, in which the base of a triangle is known and the user may wish to enter either the slope or the height. The form can classify in two ways, depending on whether the height or slope is to be entered. The default classification treats the height as input and the angle as output, with the cursor appearing in the height input field. If the user clicks on the angle box, the alternate form is selected, with the angle as input and the height as output. An invisible input field corresponds to the classification property, with two choice buttons: one over the height box and the other over the angle box.
Select either the height or slope

height: 1 m

angle: 0°

base = 12.4 m

Figure 7.8  Constraints

7.4.5 Components: Subforms

Rather than providing error messages, as in the form of Fig. 7.4, constraints on input values can be incorporated in the form, as shown in Fig. 7.9. In addition to the brace constraint of Fig. 7.5, the sheet material length must be between the minimum length and the plan length. Scales are used for the length inputs here. The minimum brace length defines the lower range of the plan length value. As the user changes the plan length, the upper bound on the other length scale is altered to be the plan length.

The slider for the sheet material length is handled as a sub-form; the reason for this is provided in Section 8.2.6.
7.4.6 Components and Classification: Tables

A form may contain component forms. For example, the form shown in Fig. 7.10 contains a table with a row for each storey. A row contains an input field for a height, and a button which leads to a form providing information about the wind and earthquake loads on that storey.

The user has created two storeys and entered height information. A further storey may be added by clicking where the third row will appear.4

This interface appears to be at odds with the single-assignment nature of Class. This is resolved by handling the addition of another row in a similar way to the triangle form in Fig. 7.8. There is a hidden “more rows” selection associated with each row, which is false by default. The user adds another row by making “more rows” true, causing a change which

---

4 The means of addition, copying, and deletion of rows in a table will be resolved as the forms interface is developed.
results in another row appearing. The implementation of the table in Fig. 7.10, using the class *List* from the previous chapter, is discussed in detail in the next chapter.

Figure 7.10  A Table in a Form

The introduction of tables requires that forms may contain sub-forms, corresponding to the "nesting" of objects inside objects within *Class*. This provides for considerable generality, as an element of a table may in turn be another table.

5 The forms system is responsible for laying out the rows of the table on the screen, introducing scrolling when necessary.

6 In fact, a property whose type is a class refers to an object; this provides for multiple references to an object within *Class* and multiple references to a sub-form in the forms system!
7.5 Applications of the Forms Interface

Two distinct uses of the forms interface are provided to illustrate the generality of the approach taken.

7.5.1 WallThermalResistance

The WallThermalResistance module allows a user to "construct" the layers of a wall so that the thermal resistance of the wall can be calculated (Mugridge et al., 1989). The facilities of the forms interface are flexible enough to allow for it to be reimplemented with the forms system, making a specialist user-interface program unnecessary.

The module could be built within the forms interface as shown in Fig. 7.117. This consists of three tables:

- Icons of the layers, laid out across the form within a horizontal table. Each element of the table is a single icon, selected to be displayed by the Class program.

- A table of forms, one for each layer. A single form is displayed at a time, as there is only enough space for one element to be shown at a time. The user can scroll between the elements by using the scroll bar at the base of the table. A form for a layer allows the user to select the main material and then select (or enter) other information about the particular material (thus classifying it).

- A table of icons of the layers, along with the name of the material type. This serves as a key to the icons used in the first table.

The first and third tables become scrollable if too many elements (rows/columns) are added.

7.5.2 Hypertext

The qualities of the proposed forms interface open up interesting possibilities for providing hypertext-like systems. For example, a Class program could act as a front-end to a code of practice, in which the user can "traverse" among elements of the code of practice based on selections and inputs which select and narrow down what is seen. Rada and Barlow (1988)

---

7 The design of the interface here is based on the interface of the prototype WallThermalResistance module.
7. A Forms Interface

point out the role of expert system rules within hypertext, where rules are used, for example, to select appropriate paths or limit the information provided.

Figure 7.11 WallThermalResistance in the Forms Interface
There need be no distinction between hypertext and an applications program. For example, a system for checking compliance with a code of practice, like *ThermalDesign* (Hamer, 1989), could also provide helpful information in a hypertext form. Hamer et al (1989) discuss a range of possible computer-based systems for helping a user with codes of practice, as follows:

- online copies of a code, perhaps with an associated indexing method
- hypertext systems for traversing a code. For example, Varnier (1989) discusses the use of hypertext to assist in building design.
- checker for code compliance
- a system which provides design assistance in meeting code requirements

The proposed forms-based user interface allows for this range of code-based systems to be built within a common framework and to be integrated in a variety of ways. In addition, it could be provided to work along with other tools, such as a graphical system for plan entry.

The forms system could also be used without change to provide part of a debugging system for a *Class* programmer (Mugridge, 1989). This could be done by automatically generating forms for each class in a program. When the program is running, the programmer is able to press a button in order to view the program objects in forms.

### 7.6 Conclusions

A flexible, general-purpose forms interface has been proposed, which incorporates the model of dynamic classification from *Class*.

The proposed forms interface system appears to be more general than Syntel (Duda et al, 1987) as it incorporates a procedural aspect through the use of buttons; for example, the hypertext qualities of the proposed forms interface are beyond Syntel. In addition, classification from *Class* maps to form refinement in the proposed forms interface. It is unclear from (Duda et al, 1987) whether Syntel provides for variable-length tables.

The proposed forms interface solves the outstanding problems arising from the work of Clausen (1989): The “sequencing problem” is solved through the use of buttons; tables are included; and traversal between forms and through help information is integrated within a general hypertext system.
7.6.1 Needs of an External Interface

The needs of an external interface to Class have been considered, based largely on examples from a proposed forms interface. The structuring facilities of Class, including classification, have all have a role in the interface. The correspondence between forms and classes is as follows:

- A form is like an object of a class.
- Input and output fields correspond to properties in an object.
- Buttons correspond to procedure calls to an object.
- A request to open or close a form corresponds to a procedure call from an object.
- A subform corresponds to a property within an object which refers to another object.
- Refinement of forms corresponds to classification.
- The commonalities between different forms (e.g. all help forms could have a common header) corresponds to inheritance.

This list seems to cover much of the "structural" parts of Class.

7.6.2 Other Needs of an External Interface

The interface needs of Class go beyond forms-based interfaces. Access is also needed to files, databases, and other programs, including "direct manipulation" interfaces such as WallThermalResistance (Mugridge et al, 1989) and PlanEntry (Hosking, 1989).

Both a file and a relational database are similar in structure to a table in a form; a row corresponds to both a record in a file (with classification for a limited form of variant records) and a tuple in a relational database. Hence no further requirements for the interface to Class arise from handling files and databases. The needs of an interface to other programs is to still to be explored.

Chapter 8 introduces an external interface for Class which interfaces a Class program to the forms system.
Chapter Eight

External Interface to \textit{Class}

8.1 Introduction

The design of a general-purpose external interface to \textit{Class} is described that allows the integration of \textit{Class} programs with a range of user-interface systems, external programs, files, and databases. This design meets each of the requirements discussed in the previous chapter.

The design builds on preliminary proposals of Clausen (1989). Clausen proposes an architecture in which \textit{Class} and a User-Interface Manager System (UIMS) run as separate systems in parallel, with control and data passing between them. He suggests that user interface classes be defined within the \textit{Class} program, corresponding to forms on the screen. Information needed by the user interface is passed as parameters, and information supplied by the interface is defined as public properties of the class. Once any data is required from a user interface class, the corresponding form is created and so the user may enter any data on the form. Changes are propagated back to the UIMS, specifying the interface information which is no longer valid.

Clausen's suggestions are lacking in detail: he does not seriously consider the management of change and he does not specify how the interface between a UIMS and \textit{Class} is to be incorporated into the execution model of a \textit{Class} program.

Section 8.2 outlines the external interface at the \textit{Class} program level. The notion of \textit{external classes} is introduced into the language. An aim in the design of the external interface is to allow for the communication of arbitrarily complex structures. An example using variable-length tables is presented which shows the generality of the approach.

The implementation of the proposed external interface within \textit{Class} is then discussed. Processes are introduced into the implementation of \textit{Class} in order to handle concurrent activity with the user. The design allows for laziness to be preserved through the interface.

The final section concludes the chapter.
8.2 External Classes

The notion of an external class is introduced in Class for the purpose of defining an external interface. An external class has different types of components from classes because the needs differ considerably from those of a class. For example, the external class has to make explicit the dependency relationships between properties.

An external class may only contain the following parts:

- properties, which correspond to values transferred between Class and the forms system.
- procedures, which correspond to buttons in the forms system.
- components, which correspond to sub-forms in the forms system.
- generalisation relationships: external classes may be arranged in a type hierarchy. While an internal class may inherit from an external class, the reverse is not permitted.

An external object is an object which is of an external class and is used to transfer data and control between Class and external code (such as a forms system).

8.2.1 Dependency Issues

Within Class, the value of a property is dependent on the values that are used in evaluating it. With predefined functions, such as \( \sin \), it is assumed that the result of a function is dependent on all of the arguments. Dependencies need to be handled through the external interface. One approach is to expect external programs (such as the forms system) to record dependency information and coordinate dependency management with the Class system. However, such management is complex and so the interaction between Class and other programs would be error-prone.

Instead, it is better if the Class system handles dependencies. This makes it is necessary for the Class program to provide specific information for dependency management, using determinantal properties. However, this requires that the programmer understands these issues when using the external interface.
8.2.2 Properties

The following sorts of properties may be used:

- Determinant properties. The dependency management system records that input property values provided in an object of the external class are dependent on the determinant properties. They correspond to parameters of the ask function, but need not be used by the external code. Determinant properties include form and file names.

- Output properties. Output properties provide data to the external code (but these should not be prompts). For example, an output property corresponds to an output field in a form, which is displayed as soon as it has a value. An output property may use input property values of the same interface, and often will.

- Input properties. A value for an input property is supplied by the external code. For example, it corresponds to a value entered by a user in an input field of a form. Input properties are dependent on the determinant properties.

The properties above may only be of the simple types (integer, float, text, boolean, and enumerated type). On the other hand, a component property may only have a type of an external class; a component property of an external object refers to another external object.

8.2.3 Procedures

There are two sorts of procedures:

- procedures. A procedure is associated with a request for some action from the external code. For example, procedures are associated with push buttons on a form. When a button in a form is selected by the user, the corresponding procedure is called.

- external procedures. An external procedure is called from within a Class program to request some action from the external code. For example, the Class program may call the external procedure show to ensure that a form is visible on the screen.
8.2.4 Using External Classes

External classes can only be used through inheritance from an ordinary class, as shown in Fig. 8.1 and Fig. 8.2. This minimises the capability required of external classes, so they are purely for defining the external interface. A subclass is responsible for providing values for the determinant and output properties and using the input property values provided by the external code. It is also responsible for providing the procedure body of an event procedure.

As values of input properties are required by a Class program, they are requested from the external code. Values for output and determinant properties in the external object are evaluated and provided on the request of the external code.

```plaintext
external class DiagonalBracesInterface.
  determinant braceNo : integer.
  determinant braceLine : text.
  output minimumLength : metres.
  output bracingProvided : bu.
  output lengthError : text.
  procedure help.
    input diagonalbrace: [steelAngle, letinTimber,
         pairsFlatSteel, pairsCutinTimber].
    input sheetMaterial : [oneFace, bothFaces].
    input planLength : metres.
    input sheetLength : metres.
    external procedure show.
    external procedure hide.
end DiagonalBracesInterface.

class Bracing.
  generalisation DiagonalBracesInterface.
  parameter minimumLength, braceNo, braceLine.
  bracingProvided
    := fnOf(planLength, sheetLength, buPerMetre).
  buPerMetre : integer
    := afunctionOf(diagonalBrace, sheetMaterial).
  lengthError
    := 'The brace must be at least the minimum length.'
      if diagonalPlanLength < minLength
    else '...'
  procedure help.
    ...
  end help.
end Bracing.
```

Figure 8.1 An Interface to a Forms-Based User-Interface
8.2.5 An Example

An example external class is provided in Fig. 8.1; this corresponds to the form in Fig. 7.3. The class *Bracing* provides all the processing, such as the code to calculate the output property *bracingProvided* based on the four input properties *diagonalBrace*, *sheetMaterial*, *planLength*, and *sheetLength*.

The relationship between these classes is illustrated in Fig. 8.2. The external class is drawn as an "oval box"; it provides the means of communication with the external code.

8.2.6 Classification

The *Class* code in Fig. 8.3 (a) and (b) corresponds to the form of Fig. 7.8. This illustrates the use of classification.

There is a mapping between the classification occurring within the program and the selection of sub-forms in an external form, as shown in Fig. 8.4. The class *Triangle* has the classification property which causes the selection to occur.
external class TriangleForm.
    determinant base : float.
    input enterSlope : boolean.  % "classification property"
end TriangleForm.

external class SlopeFromHeightForm.
    generalisation TriangleForm.
    output aSlope : float.
    input aHeight : float.
end SlopeFromHeightForm.

external class HeightFromSlopeForm.
    generalisation TriangleForm.
    input aSlope : float.
    output aHeight : float.
end HeightFromSlopeForm.

Figure 8.3 (a)  Classification: External Classes

class Triangle.
    generalisation TriangleForm.
    parameter base : float.
    classification either : [SlopeFromHeight, HeightFromSlope]
        := HeightFromSlope if enterSlope
        |  SlopeFromHeight.
    public height : float.
    public slope : float.
end Triangle.

class SlopeFromHeight.
    generalisation Triangle, SlopeFromHeightForm.
    height := aHeight.
    slope := atan(height / base).
    aSlope := slope.
end SlopeFromHeight.

class HeightFromSlope.
    generalisation Triangle, HeightFromSlopeForm.
    slope := aSlope.
    height := base * tan(slope).
    aHeight := height.
end HeightFromSlope.

Figure 8.3 (b)  Classification: Classes
An object of class *Triangle* will be classified according to the value of the input property *enterSlope* once a value for *height* or *slope* is required. An example object, in which *enterSlope* is true, is shown in Fig. 8.5.

8.2.7 Components

The *Class* program outlined in Fig. 8.6 illustrates the use of components in an external class. This corresponds to the form shown in Fig. 7.8. The relationship between a form and its component forms is modelled in *Class* in the corresponding external classes. An external
class which corresponds to a form containing a sub-form has a component property which is of a suitable external class type. Fig. 8.7 shows the relationship between the classes and illustrates that there are two connections made to the external code - for the form and the subform.

The value of the property sheetLength is dependent on the value of the property planLength because the plan length is a part of the "prompt" of the sheet length and hence a change of the plan length will affect the sheet length. If sheetLength was contained within the same external class as planLength, there would be no means of specifying that the sheetLength is dependent on the planLength: it makes no sense to specify that a determinant property is also an input property.

external class DiagonalBracesForm.
  determinant braceNo : integer.
  determinant braceLine : text.
  determinant minimumLength : metres.
  output bracingProvided : bu.
  procedure help.
    input diagonalBrace: [steelAngle, letinTimber, pairsFlatSteel, pairsCutinTimber].
    input sheetMaterial : [oneFace, bothFaces].
    input planLength : metres.
    component sheetSlider : SheetLengthForm.
    procedure show.
    procedure hide.
end DiagonalBracesForm.

class Bracing.
  generalisation DiagonalBracesForm.
  parameter minimumLength, braceNo, braceLine.
  public bracingProvided.
    bracingProvided := fnOf(sheet^sheetLength, planLength, buPerMetre).
    buPerMetre : integer
      := aFunctionOf(diagonalBrace, sheetMaterial).
    sheet : SheetLength
      := new(minimumLength := minimumLength, maximumLength := planLength).
    sheetSlider := sheet.
    procedure help.
      ...
    end help.
      ...
end Bracing.

Figure 8.6 (a) Component Forms
The component property *sheetSlider* of the external class *DiagonalBracesForm* is of type *SheetLengthForm*; this component is used by the forms system to relate the form to the subform externally. It is necessary to ensure that a subform is not created independently from the external object that contains it. For example, an independent external object of external class *SheetLengthForm* doesn’t make any sense.

Hence the containing form needs to be created first. This requires that the determinant properties of the component object depend for their value on input properties from the containing external object.¹ For example, the determinant property *maximumLength* (of an object of the external class *SheetLengthForm*) is assigned a value which comes from the input property *planLength* (of an object of the external class *DiagonalBracesForm*). This value assignment is through the second parameter of an object of the class *SheetLength*.

The determinant properties of an external object of a class which appears somewhere as a component property are evaluated eagerly when the object is created. This ensures that the object which will contain the new object as a component will be connected to the external code first. For example, when an object of class *Bracing* from Fig. 8.6 is first created there is not yet a connection with the external code, as no access has been made to the external class part of the object.

---

¹ It is only sensible if this is the case, but it is impossible for the *Class* system to check this in general.
Details of the order of execution are as follows:

- When a value is required for the public property `bracingProvided`, the property `sheet'sheetLength` will be evaluated.

- This causes the expression of `sheet` to be evaluated, creating an object of class `SheetLength`; then the public property `sheetLength` of `sheet` is evaluated.

- The property `sheetLength` is an input property within the external class `SheetLengthForm`, and so a connection needs to be made with the external code. As the class `SheetLengthForm` appears as the type of a component elsewhere (in fact in external class `DiagonalBracesForm`), its determinants are eagerly evaluated.

- The evaluation of the determinant `maximumLength` causes the evaluation of the actual parameter, which is `planLength`.

- The evaluation of this input property in class `DiagonalBracesForm` causes a connection to be made for that form.
- The connection of the subform does not occur until a value has been provided back by the external code for planLength, and assigned to the determinant MaximumLength in the subform object.
8.3 Variable-length Tables

Components, classification, and recursion are required for handling variable-length tables. Consider the form in Fig. 8.8. Each row of the table is defined as a separate sub-form and hence has a separate external object. The outputs and inputs within a row are handled just like the fields of any other form. The rows of the table are linked together recursively, as the user adds new rows. When a row is first created, the default input for that row is that the row is empty and is not visible to the user. When the user adds another row by clicking below the last visible one, it is treated as a change of the value "the row is empty", and that row becomes a visible row with three fields; it also links to the next row. Thus a row is a generator as discussed in Section 6.7.

Figure 8.8 Relationship between Rows in the Table and Objects in Class

---

2 Adapted from Fig. 7.10.
Fig. 8.9 illustrates the classes used in constructing the table and its rows. The classes in the large box represent one level of recursion, in which the Row after the one shown is empty. When this last Row is changed from being “empty”, it classifies to a FullRow, which in turn both changes the List element to a ListNode and changes the empty RowInterface into a FullRowInterface (i.e. with fields in it). Hence the single assignment causes the refinement of three parts of the Row at once, retaining a mapping between the rows in the table and the elements in the list (of heights) that is being created.

The program outline for the table handling is shown in Figure 8.10 (a,b,c). While this is rather complex to handle a single table, an abstraction of the generator can be made which reduces the complexity. Such a generator is also appropriate for handling sequences of records to/from a file and sequences of tuples to/from a database.

![Figure 8.9 Class/Interface Structure for the Table](image-url)
external class TableInterface.
   output totalHeight : float.
   component rows: RowInterface.
   procedure roofInformation.
   procedure closeStoreys.
   procedure help.
end TableInterface.

class Table.
   parameter roof : Roof.
      storeys : List[Storey].
   generalisation TableInterface.
   public storeyHeights : List[float]
      := tableRows.
   totalHeight := tableRows^fold(fn(1, tot) => 1+tot), 0).
   rows := tableRows.
   tableRows := new(number := 1, storeys := storeys).
   procedure roofInformation.
      roof^loadings.
   end roofInformation.
   procedure closeStoreys.
      storeys^call(closeLoading).
   end closeStoreys.
   procedure help.
      helpForm^show.
   end help.
   helpForm : BuildingHelp := new.
end Table.

Figure 8.10 (a) Variable-Length Table: Table

external class RowInterface.
   input empty : boolean.
end RecordInputInterface.

external class FullRowInterface.
   generalisation RowInterface.
   input height : float.
   component rowTail : RowInterface.
   event storeyInformation.
end FullRowInterface.

Figure 8.10 (b) Variable-Length Table: External Classes
class Row.
  generalisation RowInterface, List[float].
  parameters
    number : integer.
    storeys : List[Storey].
  classification FullRow : boolean
    := not empty.
end RecordInput.

class FullRow.
  generalisation Row, FullRowInterface, ListNode.
  rowTail := tail.
  tail := new Row(number := number + 1,
  storeys := storeys).
  head := height.
  myStorey : List[Storey]
    := storeys^filter(fn(s) => s^num = number)).
  procedure StoreyInformation.
    myStorey^call(show).
end StoreyInformation.
end Row.

Figure 8.10 (c) Variable-Length Table: Row
8.4 Multiple Inheritance and Reference of External Classes

Multiple inheritance of external classes and multiple references to an external object raise interesting possibilities, as illustrated in Fig. 8.11.

Multiple Inheritance

Multiple inheritance of external classes is handled in a similar way to other classes. However, the mapping to external code has to be specified. For example, the case shown in Fig. 8.11 (a) has two independent external classes which corresponds to two separate views of an object of the underlying class. This is in contrast with the case in Fig. 8.11 (b), in which there is a common "ancestor", so that a single view is provided.

![Diagram of multiple inheritance and reference of external classes](image)

Figure 8.11 Some Interesting Possibilities

Multiple Reference of External Objects

Consider the case shown in Fig. 8.11 (c), in which an external class is a component of two separate external classes. In the forms interface, this corresponds to a subform which
appears in two completely separate forms. The Class system must ensure that when the user enters an input value in one “copy” of the common sub-form, it is updated in the other “copy”.

8.5 Implementation of the External Interface

The external interface requires modification of the execution model of Class, to allow for concurrent activities through the external interface. For example, Class must be able to receive values for input properties and handle events from any form at any time.

8.5.1 Concurrency and Laziness

Class is a demand-driven system, in that expressions are evaluated when they are needed. A value may be needed in another expression or because a procedure has forced the evaluation to occur. It is appropriate that laziness be provided in the same way through the external interface, so that an input property is only transferred to the Class system when it is required. Laziness is especially important with lists of external objects, such as with input files; each record should only be provided by the external code as it is needed.

Output is data-driven in the current Class system, with procedure calls to display and summary. Three approaches to the evaluation of output properties are possible with the external interface: the external code requests the values of output properties as they are needed (lazy); output procedures are eagerly evaluated automatically; or procedures are used by the programmer to explicitly force their evaluation. The lazy approach is chosen as it provides for more flexibility than the eager approach and is simpler for the programmer. This approach is also used for determinant properties, except where the external object concerned may be a component of another external object (as discussed in Section 8.2.7).

8.5.2 Connecting to the External Code

A “connection” of an external object to the external code is made when the Class program or the external code initiates it. The Class system passes to the external code a unique external object identifier (EOI), which is used in all further communication between the external code and the Class system concerning that external object.

Initiated by Class

A Class program initiates a connection when an input property value is required or an external procedure in the external object is called.
A process is created when the external code signifies that an event has occurred, such as when a user selects a push button. This process executes the appropriate procedure associated with the external object concerned and then terminates. When a request for an output or determinant property is received by the Class system, a process is created to evaluate the property and return it to the external code; the process then terminates.

*Initiated by the External Code*

Connection is initiated by the external code when it needs a component of an external object. This occurs, for example, when the external code is writing a file and needs the next record from Class. When the request is received by the Class system, a process is created to collect the reference to the component object.

### 8.5.3 Transferring Data and Control

The values of input and output properties are passed between the Class system and the external code as they arise, along with a specification of the EOI and the property involved. For example, an output value is transferred to the external code along with the EOI and the relative output property number within the external object. A reference to a component is passed as an EIO.

Procedure calls are signalled by the external code, again by specifying the EOI and relative procedure. The Class system creates a process to evaluate the procedure. External procedure calls to the external code are made in a similar manner.

### 8.5.4 Processes in Class

The new Class system allows for multiple threads of execution of a program, as separate processes. A process may be in one of four states of execution:

- currently executing.
- blocked, waiting on an input property value of an external object.
- blocked, waiting on a value for a property that is currently being evaluated by another process.
- waiting, having been preempted by another process.

When the Class system receives an input property value from the external code, it interrupts the current process if any processes are waiting on that input value. These waiting processes
are then initiated. When they have blocked or completed, the delayed processes can continue. Similarly, when a request for a procedure call is made, a new process is created and runs immediately, preempting the currently running process. Requests from the forms system (and other user-interface systems) may need to be given higher priority, to give preference to recent actions of the user.

Once a process has finished evaluating a property, it provides the value to the external code and terminates. If at some later time a change affects an output property, a new process is created to re-evaluate the property.

8.5.5 Change

A change occurs when the external code provides a value for an input property which already has a value. The Class system ensures that all properties which depend on the changed value are re-evaluated. A change may affect the interface by invalidating a determinant property of an external object.

The external code is informed of the invalidation of an output property. It is expected to request a new value at some point.

A change affecting a determinant property of an external object invalidates the input properties of that object. The external code is informed about any external object which has been invalidated by a change. In the case of the forms system, the change is brought to the attention of the user, as discussed in section 7.4.2. The external code is expected to request values for all output properties, given that they have been invalidated by the change.

A change may invalidate the existence of an external object, such as where the creation of an interface object was dependent on some condition which is no longer true. In this case, the external code is informed that the external object is invalid. After further changes, the rejected external object may be validated once again; the external code is informed.

Change is common under the forms interface, because many inputs have an initial state which the user immediately changes. In addition, adding a row to a table is handled as a change. This implies that change management within the new Class must be rapid.\(^3\)

---

\(^3\) The design of a fast dependency-management system is under-way (Hamer and Mugridge, 1989).
8.5.6 Protocol of Interaction

Communication between the Class system and the external code is asynchronous.

The following calls are made to the external code (with the information supplied given in brackets):

- Connect to external code (EIO#, ExternalClassName).
- Request the value of an input property (EIO#, Input#).
- Supply the value of a requested output property value (EIO#, Output#, Value) or determinant property value (EIO#, Determinant#, Value).
- Supply a reference to a component (EIO#, Component#, SubEIO#).
- External procedure call (EIO#, ExternalProcedure#).
- Determinant change (EIO#, Determinant#).
- Interface rejection (EIO#)

Calls made from the external code to Class include:

- Supply the value of an input property value (EIO#, Input#, Value)
- Procedure request (EIO#, Event#).
- Output property value request (EIO#, Output#).
- Determinant property value request (EIO#, Determinant#).
- Component reference request (EIO#, Component#)

8.5.7 Adding Strictness

Determinant and output property values are usually needed by the forms system. This leads to an unnecessary communication overhead as the forms system requests each of these values. If this overhead becomes a problem, the Class system could ensure that the determinant and output properties of an external object connected to the forms system were evaluated eagerly at the time that the connection was made. Requests for input property values could also be eliminated, in which the forms system transfers input property values to the Class system as they are provided by the user.
8.6 Summary

The design of a general-purpose external interface to Class has been described that allows for the integration of Class programs with a range of user-interface systems, external programs, files, and databases. While examples are provided for the forms interface, each of the techniques that have been used also applies to other interfacing needs.

*External classes* have been added to the language; these classes contain properties, procedures and components. It is the responsibility of ordinary classes to provide for the processing and "linking" external objects together. External objects allow for the asynchronous communication of data and procedure calls in each direction between a Class program and external code.

*Class* program outlines have been provided for simple examples as well as more complex ones which illustrate the value of components and classification for transferring complex structures through the interface. For example, variable-length tables can be encoded with a mixture of recursive classes and classification.

Processes are introduced into *Class* in order to allow for concurrent activity with the user. Laziness is preserved through the interface and dependency management largely retained by the *Class* system.
9. Conclusions and Future Work

Chapter Nine

Conclusions and Future Work

9.1 Conclusions

The overall aim of this thesis was to further the development of the programming language Class. This process was begun by building a substantial application system with the language in order to find its strengths and weaknesses. This work has been successful in both clarifying the important aspects of the language and isolating a number of weaknesses. This has led to a number of proposed language extensions which enhance the quality of the language.

Chapter Three discussed the development of WallBrace, a system for checking the conformance of a building against the wall bracing requirements of a code of practice. This work highlighted a number of problems with Class, the implementation language, as discussed in Chapter Four.

Chapter Four provides a critique of Class. The object-oriented aspect of Class is more refined than the other aspects, with laziness and classification being a strong feature of the language. However, some abstraction facilities are missing because, for example, it is not possible to abstract out the notion of a recursively-defined sequence. There is a need for functional abstraction and the user interface is poor. These failings, along with a number of less important ones, have been addressed in the following four chapters which discuss a number of extensions to Class.

Chapter Five introduced functional abstraction into Class. Higher-order and polymorphic functions were added, based on ideas from the functional languages. Discrimination functions were introduced and were shown to provide both the pattern-matching capability of functional languages, and the capability of multi-methods from CLOS. The statically-typed functions of Class extend multi-methods: a discrimination function can be used within one or more classes and the result type can be narrowed in a subclass. A type-loss problem was isolated which could not be handled by available type systems. This led to the informal introduction of a multi-level type inference system.
Chapter Six added generic classes to *Class*. These allow for general-purpose data structures to be written in the language and hence enable the predefined sets and bags to be eliminated from the language. Generic classification clarified the need for another approach to organising class families.

Chapter Seven discussed the design of a forms interface which solves many of the problems with the user interface that are discussed in Chapter Four. The "sequencing problem" was solved through the use of buttons, which are connected to procedures within a *Class* program. This approach opens up the possibility of using *Class* to build "hypertext" systems. Classification was important in the interface, allowing for the contents of a form to be elaborated as the user enters information. The forms interface was also used to outline the needs of a general-purpose interface to *Class*.

Chapter Eight proposes a design for an external interface to *Class*. External classes are added to the language; these are used to interface with external code. Input, output, and determinant properties may be defined in an external class to allow for the transfer of data between a *Class* program and external code. Procedures allow for the transfer of control in each direction. Component properties allow for groups of external objects to be passed through the interface. Classification is also mapped through the interface. Processes are introduced into the *Class* system to provide for concurrent activity with the user. Laziness is preserved through the interface and dependency management largely retained by the *Class* system.

### 9.2 Future Work

#### 9.2.1 Types

*Type Semantics*

The most important future work is to define the type system of *Class* and show that it is sound. A static type inference algorithm must be defined that can handle the typing of discrimination functions. A starting point is Ohori and Buneman (1989), who propose a static type inference system for parametric classes in a language in which there is an explicitly defined lattice of class types, as in *Class*. They have a notion which appears to be similar to "like Current"; hence the limitations of their type system need to be explored.
Predefined Types

The predefined types in Class are separate from the class types. While it is possible to mix them in a restricted manner, as shown in Section 6.6.1, it would be more convenient if the predefined types could be integrated with classes.

Eiffel offers an approach with the introduction of "expanded classes" and operators such as "+" for functions in a class (Meyer and Nerson, 1989). For example, it would be convenient in Class to introduce infix forms for some of the functions of the generic class List.

Unification and LOGIN

It will be beneficial to explore the relationship between type inference and discrimination functions in Class with the work of Ait-Kaci and Nasr (1986), who incorporate inheritance into unification within a logic programming language.

9.2.2 I/O

Database

Rather than providing an interface to a relational database at the level of relations, it would be convenient to also implement the object-oriented aspects of Class with a relational database. This could be based on the work of Smith and Smith (1977), who introduced the notion of "clusters" for statically describing the classes and subclasses of an object stored in a relational database.¹

Change and Transactions

The external interface design of Chapter Eight assumes that the files and databases are unchanging during the execution of a Class program. Ideally, the Class system should take a snapshot of any file or database at the time that the first access is made to it. This issue, one in common with other functional languages, needs to be explored.

¹ Their notion of "clusters" was an important influence in the development of multiple classification in Class (Hamer, 1990).
There is also a need for a user to make a change to a database or file and then continue working. For example, a user of the *WallThermalResistance* module mentioned in Section 7.5.1 may want to add a new definition of a wall to a database. This would require them to quit from the program so that the database could be updated; this is inconvenient if they have entered information about a building, as this would be lost. One approach is to introduce the notion of transactions within a single execution of a program so that a change to the database would be treated as a change to be managed by the dependency management system. This would allow a user to make the change to the database and continue running the program.

### 9.2.3 Encapsulation

**The Problem**

Organisation and encapsulation at a level higher than classes is needed. Many classes are only relevant in a single context; for example, most of the classes in Section 8.3 could not be used in another context. The use of classification and multiple inheritance can lead to large numbers of such “local” classes.

A related issue arises when specifying subclasses of the *List* class family. This requires the introduction of a whole set of related classes (such as *Generator* and *GeneratorNode* from Section 6.8). In addition, all the generalisation relationships have to be specified carefully, leading to complex class diagrams, such as in Fig. 8.9.

LaLonde (1989) argues that even with distinct hierarchies, class systems “... require too many classes to be in the specification hierarchy...” (ibid, p222) and claims distinct advantages for an exemplar approach. He outlines the need for a higher-level organisation of classes in order to avoid the problem of browsing through a large number of classes in a system to find those which are relevant.

**Possible Solutions**

Several approaches are possible, including:

- allow nesting of classes;
- introduce modules, another structuring mechanism;
- provide tools for using libraries of classes.
9. Conclusions and Future Work

Modules

The introduction of modules is considered briefly here; the approaches used in other languages will also need to be explored. For example, the module shown in Fig. 9.1 includes the classes Set and PrivateSet. Only the class Set is exported from the module.

```
module Sets.
    public Set.

    class Set[..].
        ...
        end Set.
    class PrivateSet.
        generalisation Set.
        ...
        end PrivateSet.
    end Sets.
```

Figure 9.1 Module for Set

By allowing a module to inherit from another, it is possible to eliminate the need to introduce additional names for the classes within a sub-family. For example, the module shown in Fig. 9.2 encapsulates the two generic classes List and ListNode. This is specialised in the module shown in Fig. 9.3.

```
module ListFamily.
    public List.

    class List[].
        public cons, filter, map, fold, append, any, all.
        ...
        end List.
    class ListNode.
        generalisation List.
        end ListNode.
    end ListFamily.
```

Figure 9.2 Module for ListFamily
module ConsListFamily.
generalisation ListFamily.

class ListNode.
  parameter head.
  parameter tail.
  end ListNode.
end ConsListFamily.

module ExtListFamily.
generalisation ConsListFamily.

class List.
  public addList.
    addList(other: IntList) : IntList :- other.
    newList(head : integer, tail: ExtIntList) : ExtIntListNode
      := new(head := head, tail := tail).
  end List.

class ListNode.
  addList(other: IntListNode) : IntListNode
    := new(head + other^head, tail^addList(other^tail)).
  addList(other: IntList) : IntListNode
    := self.
  end ListNode.
end ExtListFamily.

Figure 9.3 Sub-Families of List

The class Generator from Section 6.8 is now simple to define, as there is no need to pass the classification property as a generic parameter. This is illustrated in Fig. 9.4.
Conclusions and Future Work

module GeneratorFamily.
  generalisation ListFamily.

class List.
  classification ListNode : boolean.
  end List.
end GeneratorFamily.

module AskGeneratorFamily.
  generalisation GeneratorFamily.

class List.
  parameter first: boolean.
  ListNode := ask('Are there any elements? ') if first
  | ask('Are there any more elements? ').
  end List.

class ListNode.
  head := ask('What is the first element? ') if first
  | ask('What is the next element? ').
  tail := new(first := false).
  end ListNode.
end AskGeneratorFamily.

Figure 9.4 The GeneratorFamily and a Sub-Family

Modules, Classes, and Functions

There are many similarities between modules, classes, and functions. For example, a class within a module is similar to a function within a class. It could be fruitful to unify the three notions into one construct and allowing for nesting.

9.2.4 Catalysts for the Further Development of Class

The development of WallBrace has provided considerable feedback about the quality of Class. Therefore it is appropriate to consider what other practical work can act as a catalyst for the further enhancement of the language.

One possibility is to apply Class to the development of specialist user-interfaces; this will provide feedback on the external interface. Another possibility is to apply Class to the construction of a compiler for Class; the type inference system described in Chapter 5 is probably best encoded in a lazy language and the extended Class should be ideal for this.
References


(Goldberg and Robson, 1983). Goldberg A, Robson D, 1983. Smalltalk 80: The Language and its Implementation, Addison-Wesley.


References


