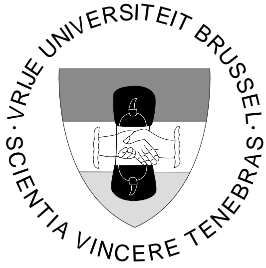


Open Design of Object-Oriented Languages

A Foundation for Specialisable Reflective Language Frameworks

Patrick Steyaert

Vrije Universiteit Brussel
Departement Informatica

Proefschrift ingediend met het oog op het behalen van de graad van doctor in de wetenschappen

Promotor : Prof. Dr. Theo D'Hondt

Preface

This thesis develops an open design for a reflective object-oriented programming language. The focus of the thesis is on how programming language concepts, rather than mere implementations thereof, can be made explicit for refinement, extension and recombination in their fullest abstract form. Therefore a mixture of techniques is studied. For the particular case of developing an open design for an object-oriented language, we look at object-oriented frameworks and how they can be used to make explicit the major design issues of a programming language. Emphasis is put on those specialisations of a framework that respect the design of the framework. In particular, concretisation into a computational system and, additionally, refinement and extension of a framework are discussed. The notions of full abstraction and compositionality are adopted from programming language semantics to judge whether programming language concepts are represented in their fully abstract form in the framework. The notion of reflection is studied in order to make the open design self-contained.

As a case study, a two layered framework is developed in which two object-oriented languages are expressed. The first language (Simple) is an object-based programming language. Its semantics is given by a calculus for objects. The second language (Agora) is an object-oriented programming language that features a generalised form of mixin-based inheritance. Simple is defined in the context of a framework that essentially features encapsulated polymorphic objects that accept a well defined set of messages. For the definition of Agora it is shown that a layer can be added to the framework to include the generalised form of mixin-based inheritance. Descriptions of extensions to Agora are given. Among others, extensions are described that allow flexible, controllable and dynamic construction of multiple inheritance hierarchies.

In a final stage the framework is extended with reflection. The link between open systems and reflection is studied. Open designs and linguistic symbiosis replace the conventional meta-circular interpreters in the account of reflective systems presented in the dissertation. It is concluded that systems with an open design form an excellent basis for the definition of reflective systems. Moreover it is concluded that they are an important step in the demystification of reflection.

Acknowledgements

I thank my advisor Prof. Theo D'Hondt for setting up the inspiring environment in which this work has found its roots. He created the intellectual environment by bringing the group of people together that surrounded me during the past years. I would like to thank him for introducing me into the field of object-oriented reflection and for his support. He also had a substantial influence on the structure of this text.

I owe special thanks to the people that form the heart of the Agora group: Wim Codenie, Koen De Hondt, Carine Lucas, Marc Van Limberghen. They were always open for discussion and helped me in solving all kinds of problems. Especially in the design of Agora they were influential. All four are co-responsible for its current shape. Wim, Koen, Carine and Marc proof-read this dissertation, some parts more than once.

Kris De Volder, Karel Driesen, Serge Demeyer also proof-read parts of this text. I thank Kris for the numerous and long discussions on reflection. I would like to thank Karel for the numerous discussions on whichever topic. I thank Serge for his continuous interest and support.

Thanks, also for comments on the text, to the members of the thesis committee: Prof. Pierre Cointe, Prof. Viviane Jonckers, Prof. Pattie Maes, Prof. Robert Meersman, Prof. Luc Steels, Prof. Dirk Vermeir.

Several other people have been helpful. I owe thanks to the former members of the Agora group and the students that helped in shaping and implementing Agora. In the early stages of this research Michel Tilman and Thanh Son Du have been helpful.

Franz Hauck has been open for discussion each time we were able to meet. More recently, Niels Boyen, Wolfgang De Meuter, Kim and Tom Mens have helped by showing their interest. I thank Prof. Viviane Jonckers for giving helpful comments on the introduction of the text.

All other people of the Computer Science Department and the Programming Technology Lab each play an important role in maintaining an optimal working environment. I hereby also thank all people, in and outside the department, that have shown interest in my work.

Michel Beke and Arlette Vercammen are responsible for the layout of the text. I thank OO Partners and the people of the IT department of VTM for providing the infrastructure and the calm environment to finish this text. Finally I thank my friends and family for the necessary distraction once in a while. I especially thank Arlette for her support.

Summary Table of Contents

Preface ... *i*
Acknowledgements ... *ii*
Summary Table of Contents ... *iii*
Table of Contents ... *iv*

Chapter 1 Introduction ... 1
Chapter 2 Computational Reflection and Open Systems ... 11
Chapter 3 A Framework for Object-Based Programming Languages ... 35
Chapter 4 Specialising the Framework with Inheritance ... 103
Chapter 5 A Reflective Framework ... 165
Chapter 6 Related Work ... 191
Chapter 7 Conclusion ... 197

Bibliography ... 201
Appendix A
Appendix B

Table of Contents

Preface

Acknowledgement

Summary Table of Contents

Table of Contents

Chapter 1

Introduction 1
- 1.1 Open Programming Languages 4
- 1.2 Reflective Systems 4
- 1.3 Object-Oriented Frameworks 4
- 1.4 A Framework for an Object-Based Programming Language 5
- 1.5 A Layer for Object-Oriented Programming 6
- 1.6 A Layer for Reflective Object-Oriented Programming 8
- 1.7 Related Work 8

Chapter 2

Computational Reflection and Open Systems 11
- 2.1 Introduction 11
- 2.2 Model of Computation 12
- 2.3 Absorption and Reification in Programming Languages 14
- 2.4 Open Implemented Computational Systems 17
- 2.5 Reflection: Accessing One's Own Meta-system 22
  - 2.5.1 Reflective Architectures 23
  - 2.5.2 Reflective Facilities 24
  - 2.5.3 Meta-Programming 25
- 2.6 Managing Reflective Overlap and Tower Architectures 26
- 2.7 Computational Systems with an Open Design 27
- 2.8 Full Abstraction and Compositionality in Programming Languages 28
  - 2.8.1 Full Abstraction and Compositionality in Semantics of Programming Languages 29
  - 2.8.2 Full Abstraction and Compositionality in Implementation of Programming Languages 31
- 2.9 Conclusion 33

Chapter **3**
A Framework for Object-Based Programming Languages 35
3.1 Introduction 35
3.2 Design Issues in Object-Oriented Programming Languages 36
3.2.1 Objects, Interfaces, Messages and Encapsulation 40
3.2.2 Alternative Object Models 44
3.2.3 Operations on Objects 48
3.2.4 Classes and Class-based Inheritance 49
3.2.5 Classless Delegation 55
3.2.6 Mixin-Method Based Inheritance 58
3.2.7 Encapsulation as an Explicit Operation on Objects and Generators 61
3.2.8 Objects with State, State Changes and Object Identity 63
3.3 Object-Oriented Frameworks 65
3.3.1 Reusability in Object-Oriented Programs 66
3.3.2 Reusability in Object-Oriented Frameworks 71
3.3.3 Abstract Classes 71
3.3.4 Operations on Abstract Classes 72
3.3.5 Role of Abstract Classes in Frameworks 75
3.3.6 Frameworks, Conclusion 76
3.4 A Simple Object-based Programming Language 77
3.4.1 A Calculus for Object-based Programming 77
3.5 Definition of the Framework 81
3.5.1 Representation of Programs and Compositionality 81
3.5.2 Representation of Objects and Full Abstraction 85
3.5.3 Message Passing 86
3.6 Concretisation to a Simple Object-based Language 87
3.6.1 Abstraction Expressions and Object Structures 87
3.6.2 Association Expressions and Slots 89
3.6.3 Message Passing 90
3.6.4 Implementation of Simple, Summary 91
3.7 Improving the Framework 92
3.7.1 Reifier Methods 92
3.7.2 Extra Indirection Needed in Context and Client Objects 94
3.7.3 Evaluation Categories and Category Patterns 96
3.7.4 Making the Layered Structure Explicit 100
3.8 Conclusion 100

Chapter **4**
Specialising the Framework with Inheritance 103
4.1 Introduction 103
4.2 Inheritance, Design Issues 104
4.2.1 Inheritance and Encapsulation Problems 104
4.2.2 The Need for Flexible and Controllable Inheritance 105
4.2.3 Multiple Inheritance 106
4.2.4 Mixin-based inheritance 118
4.2.5 Mixin-Method Based Inheritance 119
4.2.6 Mixin-based inheritance, A Solution to Multiple Inheritance Problems ? 123
4.3 Visibility and Nesting in Object-Oriented Languages 126
4.3.1 Is There a Need for Scope Rules for Encapsulated Attributes ? 127
4.3.2 Nested Classes, Classes as Attributes 129
4.3.3 Nested Mixins 130
4.4 Design of Agora 132
4.4.1 Introduction 132
4.4.2 Agora Syntax 133
4.4.3 Standard Agora Reifiers 134

4.5 The Agora Framework....141
4.5.1 Abstract Grammar, Expression Objects and Reifier Methods....142
4.5.2 Message Passing....144
4.5.3 Mixin Application and Object Structures....146
4.5.4 Agora Internal Object Structure....147
4.5.5 External Object Structures and Wrapper Objects....151
4.5.6 Extending Objects, Execution of Mixin Methods....152
4.5.7 Object Cloning....154
4.5.8 Mixin, Method and Instance Variable Declaration Reifiers and Slots....155
4.5.9 Summary of the Application of the Framework to Agora....156
4.6 Extensions to Agora....157
4.6.1 Public Instance Variables and Private Methods....157
4.6.2 Cloning Methods....158
4.6.3 Stubs for Multiple Inheritance....159
4.6.4 Single Slot Nested Objects....160
4.6.5 Classes....161
4.6.6 Abstract Methods, and Abstract Class Attributes....162
4.6.7 A Simple Form of Monotonic Reclassification....162
4.6.8 Classifiers....163
4.7 Conclusion....164

Chapter **5**
A Reflective Framework....165
5.1 Introduction....165
5.2 Object-based Reflection....166
5.2.1 Linguistic Symbiosis....166
5.2.2 Simple Meta-Programming Operators for Agora....174
5.2.3 Simple Reflection Operators....177
5.2.4 Nature of Meta-Programs and Reflective Overlap....179
5.2.5 Dynamic Reflection and Infinite Regress....180
5.2.6 Abstraction and Compositionality....182
5.3 Object-Oriented Reflection....183
5.3.1. The Evaluation and Declaration of Reifiers....183
5.3.1. Need for a More Fine-Grained Linguistic Symbiosis....187
5.4 Conclusion and Open Issues....189

Chapter **6**
Related Work....191
6.1 Reflection and Open Systems....191
6.2 Object-Oriented Reflection....192
6.3 Object-Oriented Systems....193
6.3.1 Object-Oriented Frameworks....194
6.3.2 Mixin-Based Inheritance....194

Chapter **7**
Conclusion....197
7.1 Contributions....198
7.2 Future work....199

Bibliography

Index

Appendix **A**

Appendix **B**

Chapter 1

Introduction

In this dissertation a reflective object-oriented programming language is presented. This is not the first work that undertakes such an endeavour. Many proposals for reflective (object-oriented) programming languages have already been made. However, in the current state of affairs, the introduction of reflection in programming languages and systems remains an art rather than a science. Reflective programming languages and systems are being defined in an ad hoc manner, and reflection itself remains a mystical notion.

The first contribution made in this dissertation is a further demystification of reflection by the firm and formal establishment of the link between systems with an open implementation and reflective systems. A system with an open implementation has, like any other conventional computational system, an (object-level) interface by which its functionality can be invoked. Additionally, it has a second (meta-level) interface that shows how the system's implementation can be adapted or extended. A system with an open implementation has much of the characteristics of most reflective systems — structured access to the implementation of the system is provided — without the intricate problems of self referential behaviour. In this respect the concept of open implementations is broader than the concept of reflection; open implementations can, and have been studied independently of reflection. Conversely, in this dissertation an explanation of reflection is given that is entirely based on open implementations. It is shown how, and under what conditions, a system with an open implementation can be turned into a reflective system. It is our opinion that this account of reflection is more satisfying and less mystical than most other accounts. From a practical point of view, open implementations contribute to a division of design concerns. Opening up the implementation and handling self-referential behaviour can be separated.

The second, and perhaps more important contribution is the shift from open implementations to open designs. Programming languages are opened up with the intention of defining a design space of related languages. Such a design space can,

for example, cover all languages belonging to the same programming paradigm in which orthogonal language concepts such as persistency, modularity, reflection (!), typing etc. can be explored. However, an open implementation does not define a coherent design space if it does not mirror the design of the programming language in terms of how it is composed of different language concepts.

An open programming language only mirrors the design of the programming language that is opened up if it has an explicit representation of all the important constituent language concepts. In casu, representations that can be refined and modified (within boundaries) in order to adapt the open programming language. In the implementation of a programming language not all language concepts need to be explicitly represented. A fully abstract and compositional semantics is, for example, a better place to look for the language concepts that a particular programming language is comprised of. In an open implementation where not all the important language concepts are represented for refinement and modification, some of the expressive power will be lost and the intended design space will not be covered. Furthermore, in most open implementations, language concepts are not represented in a fully abstract form. A lack of representing language concepts in their fully abstract form can result in a loss of expressivity and safety. A language concept must obey a set of constraints. For example not everything that transforms a set of arguments into a result is a referentially transparent function. A too operational representation of a language concept can exclude refinements of the language concept that conform to the constraints, and conversely, it can support operations that violate the constraints of the represented language concept. If not all the constraints surrounding a language concept are identified and obeyed, then the 'safety' of the open programming language will be lost and some languages not belonging to the design space will be covered. Such a language is called unsafe because it is very hard or even impossible to reason about programs expressed in a language of which the implementation can be altered without constraints.

A system with an open design differs from a system with an open implementation in the above listed points. It can be claimed that a programming language with an open design defines a coherent design space of programming languages. To support the construction of programming languages with an open design, we show that the complementary notions of compositionality and full abstraction from programming language semantics can be adapted as criteria to judge whether a language concept is represented fully abstract or not. Furthermore, we investigate object-oriented frameworks for the specific purpose of defining open designs for object-oriented programming languages. Object-oriented frameworks have been recently studied for expressing reusable designs. In an object-oriented framework the major design issues of a system are represented in the form of abstract classes and co-operations between abstract classes. A framework is transformed into a concrete system by concretisation, refinement and extension of its abstract classes. When using frameworks to express open designs, special attention must be paid to whether transformations of the framework preserve its initial design.

Not all transformations of a framework turn it into a concrete computational system. In fact most transformations turn a framework into a more specialised one, i.e. a framework that conforms to the initial design but covers a smaller design space — it is less adaptable. In such a case we talk about a layered framework. This also conforms to our intuition about programming language design spaces. Language concepts can be refined. For example in the object-oriented paradigm inheritance can be refined to its different variations. Here again the specialisation structure of the open design should mirror as close as possible the specialisation structure of the intended language design space.

A third, more technical, contribution of this work is the development of a two layered framework for the definition and implementation of object-oriented programming languages. The notion of an object and the constraints an object must obey are discussed. Fully abstract representations of objects are contrasted with representations that are too operational. The role of inheritance, classes and other issues generally related to object-orientation are elaborated upon.

The proposed framework is a two layered framework. The first layer of the framework is one in which object-based programming languages can be expressed. It does not handle nor contain any provision for inheritance or delegation. In our discussion of inheritance we will show that a particular form of object-based inheritance — mixin-methods — can be totally encapsulated in the object. Accordingly it is shown that this form of inheritance can be expressed in a specialisation of the first layer of the framework, i.e. remaining in the design constraints of this first layer. This forms the second layer of the framework, and handles all kinds of inheritance.

The proof of the pudding is in the eating. Different extensions to, and specialisations of, the framework are shown. These extensions handle real-world programming problems. They are inspired by currently debated issues in object-oriented programming language design. One particular specialisation draws our attention. A layer is added to the framework to handle reflection. This brings us back to our initial goal of getting rid of ad hoc defined reflective programming languages. It is shown that for open designs that are powerful enough, reflection can literally be added as an orthogonal language concept. This certainly has its advantages. Not only because it gives considerable freedom in the choice of reflection operators to be added, but more importantly because of the then established link between a reflective system and its underlying open design. We will show how a reflective extension of an open programming language can be based on the notion of a linguistic symbiosis between the open programming language and its implementation language. The idea of a linguistic symbiosis will be explained.

The result is an open, reflective object-based programming language (Agora). Other proposals for reflective object-based or object-oriented languages can be found in the literature. To our knowledge non of them achieves the same high degree of open-endedness in an equally elegant and small language as Agora. The high degree of open-endedness will be illustrated by a selection of 'real world' extensions of Agora by means of its open design. The elegance of Agora is exemplified by its compact syntax. We claim that this is, to a large degree, the result of the consistent application of the techniques and design criteria proposed in this text. It pays off to free ourselves of 'ad hoc' definitions.

In the next sections we proceed with a more detailed overview of the topics covered in the dissertation.

1.1 Open Programming Languages

The main theme of this dissertation is to illustrate how a mixture of techniques can be used to obtain a programming language with an open design (Agora). In particular the open design of an object-oriented language in the form of an object-oriented framework will be discussed. This is mainly a work of integrating and refining (and in particular cases making original contributions to) techniques from different disciplines: open implementations and reflective systems, object-oriented frameworks, design, implementation and semantics of programming languages in general, and design, implementation and semantics of object-oriented programming languages in particular.

1.2 Reflective Systems

The link between open implementations and reflection is three-fold. Firstly, we will show, in a general setting, how each reflective system (or at least the ones currently known in the programming language community) has at its kernel an open system to which reflection only adds a form of 'self-containedness'. Secondly, it is shown how and under what conditions an open implementation can be turned into a reflective one. Particularly we show, by means of a detailed case, how certain open implementations themselves can be used to build reflective systems. This gives us a considerable amount of leeway in the construction of reflective systems. And finally, in the particular case of an object-oriented language, we will explore, by making use of the proposed reflective architecture, a set of novel language concepts that support the construction of open systems. Since reflective languages can be used to explore a design space of programming language concepts, they are particularly suited to explore language concepts that support open-endedness.

1.3 Object-Oriented Frameworks

Object-oriented frameworks have been studied in the context of design and code reuse. In this dissertation they will be used as a means to express open designs. For this purpose two aspects of object-oriented frameworks need to be emphasised. First of all the distinction between a framework's external interface (corresponding to the object-level interface) and the framework's internal interface (corresponding to the meta-level interface) must be made clear. Secondly, special attention must be paid to those transformations on the framework that preserve the design of the framework.

Abstract classes play an important role in object-oriented frameworks. We will pay particular attention to two kinds of abstract classes — those with abstract methods and those with abstract class attributes — and the possible transformations of an abstract class that make it more concrete. We will make a distinction between concretisation, refinement and extension. Concretisation

involves overriding abstract attributes with concrete attributes. An abstract class is refined when concrete or template methods are overridden. It is extended when new attributes are added to the abstract class. Attention is given to the fact that concretisations may be partial, i.e. a concretisation can, for example, introduce new abstract attributes. This will give rise to layered frameworks.

As we will see, object substitutability plays an important role in constraining transformations of abstract classes so that they respect the initial design of the abstract class.

1.4 A Framework for an Object-Based Programming Language

We adopt a compositional view on programming languages. A programming language's definition (implementation) is compositional when with each expression type of the programming language a part of the definition (implementation) is associated and the definition (implementation) of a compound expression is expressed in terms of the definition (implementation) of its subexpressions. The notion of compositionality is adopted from the area of programming language semantics. A compositionally defined implementation is incrementally extensible. It is very well-suited to be captured in an object-oriented framework.

A framework for an object-oriented language not only consists of representations of expressions. Its other major ingredient is the representation of objects. We will show what it means for a representation of objects to be fully abstract. We will also show how message passing can be abstractly expressed by making use of this fully abstract object representation. This will be the kernel of our framework, as is apparent in the syntax of Agora which is essentially a syntax for message passing.

The framework is based on the particular notion of strongly encapsulated polymorphic objects that have a well-defined behaviour. In the object-oriented programming languages design community there is still much debate about what is essential to object-oriented programming. The major components involved i.e. objects, classes, encapsulation, object identity, single inheritance, multiple inheritance, delegation, polymorphism, types, … are pretty well-known. However, many of these concepts are not well-understood, or take on different meanings for different authors.

For our purpose however, i.e. that of designing a language design space of object-oriented languages, we need an understanding of what are the important and what are the less important concepts, and what constraints a language must fulfil to be called object-oriented. At least we need a coherent framework of concepts that clearly delineates a design space of languages that can be called object-oriented. It may well be an impossible task to define a language design space that covers all languages dubbed object-oriented in the literature (and perhaps we don't even want that). Examples can be found of different coherent frameworks of language concepts that are each called object-oriented. Each may define equally interesting and expressive language concepts, but have conflicting design criteria amongst each other.

So we set ourselves the task of doing an analysis of what are called object-oriented programming languages with the intention of defining a coherent framework of concepts that delineates a design space of object-oriented languages. First of all we restrict our analysis to sequential, dynamically typed languages. Furthermore, this analysis must and will follow and integrate what can be found in the literature. Much has already been said about the orthogonality (or non-orthogonality) of concepts such as objects, classes and inheritance, the nature of inheritance, the dichotomy between class-based and object-based (or prototype- or delegation-based) languages, the different variations of delegation and the different variations of (multiple) inheritance.

What differentiates our analysis from others are the criteria that are used. These criteria are the extent to which explicit interfaces, object-based encapsulation and late-binding polymorphism are supported. They are used as yardsticks to evaluate the appropriateness of all other design issues. For example object re-classification can be analysed with respect to how well explicit interfaces are supported. The three proposed criteria correspond to the intuitive notion of an object as a self contained entity that has a well-defined behaviour and responds to a well-defined set of messages.

We use these criteria mainly to (re-)analyse the dichotomy between class- and object-based languages and the different notions of delegation. For example, pure delegation-based languages can be excluded on the basis that they do not support explicit interfaces. A restricted form of object-based inheritance with implicit delegation and a delegation structure that is fixed does fulfil the above criteria. This analysis is a motivation to discard inheritance and classes from the basic structures of the framework. They will be reintroduced at a later stage.

A proposal of a framework is given that incorporates the above criteria to which objects must conform. It is used to construct a simple object-based language. This language is based upon a calculus of objects that incorporates the previously adapted design criteria. The calculus is briefly discussed. The calculus illustrates two important concepts. It features atomic message passing, which is a primitive operation in the calculus. Furthermore, it has an explicit encapsulation operator.

1.5 A Layer for Object-Oriented Programming

In a second stage a layer is added to the framework to include inheritance. Crucial in this extension is, firstly, that the inheritance structure of an object can be entirely encapsulated and, secondly, that the framework can be easily specialised with different inheritance mechanisms. For this purpose the framework will be extended with what are called 'internal objects'. Internal objects are used in the internal representation of objects to represent their inheritance structure. They are entirely encapsulated in the object representation. Different kinds of internal objects and their combinations can be configured in the framework.

An important part of our analysis of object-orientation is devoted to inheritance and visibility of names, two intimately connected issues. The central theme here is that of finding an incremental modification mechanism that is powerful enough, but still preserves encapsulation and allows the derivation of the

interfaces of incrementally defined objects. The design criteria for inheritance — modularity, incremental design, reusability and encapsulation — are relatively well-documented in the literature, the problem is to find the necessary language concepts that are expressive enough.

Whereas the semantics of single inheritance is relatively well-understood, the semantics of multiple inheritance is still a debatable issue. It is not even clear whether it is possible to construct a single, simple, comprehensible and general mechanism that solves all problems related to multiple inheritance.

We argue that it *is* possible to construct such a simple and general multiple inheritance mechanism. This is motivated by a careful analysis of the different forms and problems of multiple inheritance. In this analysis we focus on inheritance in implementation hierarchies, since herein lie most of the problems of multiple inheritance. The problems that occur in type hierarchies are left untouched.

Moreover we argue that the solution has to be found in the fragmentation of the functionality of the inheritance mechanism into its primitive building blocks. This should be done in such a way that a greater flexibility is obtained for the user to adapt the inheritance strategy to specific situations. We claim that this can be achieved by making the underlying mechanisms of inheritance explicit. The proposed multiple inheritance mechanism will be a variant of mixin-based inheritance. Mixin-based inheritance is inspired by mixin-classes in for example CLOS. In its own right mixin-based inheritance has been studied as an inheritance mechanism that underlies different other inheritance mechanisms.

We generalise mixin-based inheritance in three ways. In its original form mixin-based inheritance was introduced in a class-based language, i.e. mixins are used to extend classes. In our case mixins are made applicable to objects to enable object-based inheritance. Applying mixins to objects leads to the above mentioned form of prototype-based programming where each object can have a fixed parent object to which it implicitly delegates. Secondly, our mixins can invoke parent operations of non-direct ancestors. And finally, we address the question of how mixins can be seen as named attributes of objects in the same way that methods and objects, themselves, can be seen as named attributes of objects. This extension of mixin-based inheritance will be called mixin-method based inheritance.

It is shown that mixin-method based inheritance is an expressive mechanism to (dynamically) construct and control the evolution of multiple inheritance hierarchies. The nesting structure that naturally arises from the use of mixin-methods proves to be a useful mechanism to control implementation dependencies between mixins.

A full-fledged programming language (Agora) is presented that features mixin-methods. We will show how a particular configuration of internal objects can be used to implement mixin-methods. The framework is used in the implementation of Agora. What makes Agora special is that it basically incorporates only message passing. All other language concepts are concretisations of its basic framework. A vanilla variant of Agora incorporates mixin-methods. Extensions to Agora that are shown to be supported by the framework are amongst others: name-collision handling for multiple inheritance, cloning methods, block-objects and classifiers for controlling the applicability of mixin-methods.

1.6 A Layer for Reflective Object-Oriented Programming

One particular addition of a layer to the framework that will be discussed is a layer for reflection. What is particular about the presented approach is that the open design itself is used to introduce reflection. Reflection is literally interpreted as a language construct that can be added orthogonally to a programming language (in the same way that inheritance was added).

Turning an open programming language into a reflective programming language, is a matter of 1) achieving a symbiosis between the underlying implementation language and the open programming language itself; and 2) extending the open programming language with the necessary reflection operators that give full access to the open implementation. Our discussion will follow these steps.

First we will show how an open object-oriented language can achieve a symbiosis with its underlying object-oriented implementation language, and that this can be done with a fairly general mechanism. A symbiosis between an object-oriented language and its object-oriented implementation language can be achieved by the introduction of conversion-objects that incorporate reflection principles. These conversion-objects are noting but a special sort of objects that conform to the abstract notion of objects in the framework and allow message passing back and forth between objects expressed in the implementation language and objects expressed in the implemented language.

Based on these conversion-objects, reflection operators can be constructed. In practice, the choice of the reflection operators is an important issue. Reflection operators must give full access to the open implementation. The choice is complicated by the issue of reflective overlap. A selection of different reflection operators is discussed for Agora.

1.7 Related Work

The most widely accepted account of computational reflection in programming languages was given by Smith in [Smith82]. The intuitions behind and motivations for the introduction of reflection are adopted in this text. Still, in [Smith82] [des Rivières&Smith84] and later [Maes87ab] the account of reflection is heavily based on the notion of meta-circular interpreters [Abelson&Sussman84]. We will motivate another approach to reflection where meta-circular interpreters are substituted by language processors with an open implementation [Rao91].

Object-oriented reflection finds its origins in the work of Maes [Maes87ab] and Cointe [Cointe87ab]. Although this work was essential in proving the usefulness, flexibility and power of object-oriented reflection, our work is more related to what could be called a 'second generation' of object-oriented reflection in the form of metaobject protocols [Kiczales,des Rivières&Bobrow91] and open implementations [Rao91]. In this latter work object-oriented software engineering practices — protocol design and documentation and object-oriented frameworks — play a more prominent role. Our approach is a more structured approach by establishing the link between open implementations and reflection and by the introduction of open designs.

The most important sources of inspiration for our analysis of object-oriented programming language concepts are [Wegner90] [Stein,Lieberman&Ungar89] [Dony,Malenfant&Cointe92] and [Cardelli88] [Cardelli&Wegner86] [Cook89] [Ghelli90] [Lieberman86] [Wegner&Zdonik88] for models of object-orientation and inheritance. For an alternative look on object-orientation based on overloaded functions the reader is referred to [Castagna&al.92] [Chambers92]. Mixin-based inheritance was first introduced by [Bracha&Cook90] (the notion of mixins as a particular kind of classes can also be found in object systems on top of Lisp [Moon89]). Generalised mixin-methods are an extension of the mixins of [Bracha&Cook90]. Our analysis of the problems involved in multiple inheritance is a résumé of [Snyder87], [Knudsen88] and [Carré,Geib90].

Object-oriented frameworks find their roots in the practice of object-oriented programming. Good introductions can be found in [Johnson&Foote88] [Johnson&Russo91][Deutsch87]. We investigate object-oriented frameworks in the context of open designs. The relation between open implementation and object-oriented frameworks has already been noted in [Holland92]. In [Helm&al90] and [Holland92] a description is given of contracts — high level constructs for the specification of interactions among groups of objects — and how to refine and reuse contracts in a conforming way. We give a more intuitive explanation of how to specialise a framework entirely based on substitutability [Liskov87] [Wegner&Zdonik88] of objects. Of course this can not substitute a formal description of specifying behaviour compositions in frameworks, but should rather be an intuitive basis for it.

Chapter 2

Computational Reflection and Open Systems

2.1 Introduction

> *"**Reflection hypothesis:** In as much as a computational process can be constructed to reason about an external world in virtue of comprising an ingredient process (interpreter) formally manipulating representations of that world, so too a computational process could be made to reason about itself in virtue of comprising an ingredient process (interpreter) formally manipulating representations of its own operations and structures."*
>
> **Smith (1982)**

The notion of reflection can be found in disciplines as diverse as philosophy, linguistics, logic and computer science. It is not clear how all these different notions of reflection are connected. Even in computer science the notion of reflection, mainly used in the disciplines of artificial intelligence and programming language design, has different connotations, and especially different motivations. The common theme is that of building computational systems that, in a substantial way, have access to, reason about, and act upon their own computational process.

Before plunging into the technical, or less technical, details of what a reflective system looks like and how to build it, we will first try to answer the question "why ?". Why is it necessary to build reflective systems ? Obviously, there is no need for reasoning about one's self if this doesn't increase one's capabilities for reasoning about one's subject domain. A full analysis of representation, efficiency, and reflection in the most general case is certainly beyond the scope of this dissertation, and would necessarily have to follow the analysis that can be found in [Smith86]. Rather than doing that, we analyse these concepts in the more

restricted case of programming languages and computational systems described by programs expressed in a programming language.

For the particular case of programming languages, we will give an in-depth answer to the question why reflection is needed in section 2.4 and section 2.5. The notions of absorption and reification are introduced as two main factors in the need to design systems that open up their implementation. It is shown what it means for one system to access the implementational structures of another system. The notion of systems with an *open implementation*[1] is contrasted with the notion of systems that allow implementational access. *Reflective systems* are then introduced as a specific kind of systems with an open implementation.

While doing so, we will take a less conventional, more constructive, approach to reflective programming languages. It is a constructive introduction of reflection since we introduce reflection almost by saying how to construct a reflective system. It is less conventional because of the firm link that is made between reflection and systems with an open implementation. Rather than directly turning to the question of how a system can have access to, reason about and act upon its own internal structures, we first turn to the question of how one system can have access to, reason about and act upon another system. In particular we use the notion of open implementations where one system can inspect and manipulate the implementation of another system. It is our strong belief that this is an important step in the demystification of reflection.

We conclude this section with a discussion on the difference between systems with an open implementation and systems with an *open design*. We start with some assumptions and some terminology. Note that inevitably the terminology used, can differ from that of other author's.

2.2 Model of Computation

The assumption with which we start is that a *program* expressed in some programming language is turned into a *computational system* by means of another computational system commonly called the *meta-system* [Maes87]. For example, for a program this meta-system can take the form of an evaluator.

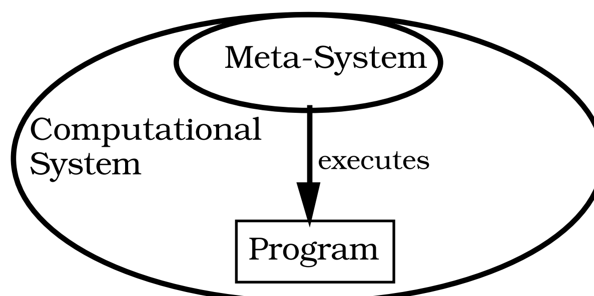


Figure 2.1

[1] It should be stressed that the notion of open implementations must not be confused with that of meta-systems [Maes87a] nor with meta-level architectures. See the section on open implementations for a discussion on the topic.

Although we find that for this special case, the term meta-system is somewhat misleading, it will be adopted in this text. In contrast with what would be expected, the meta-system is not a system that reasons, or acts upon another system, but rather it is a system that reasons about a *program* which is a *description* or *representation* of a computational system[2]. In the section on open implementations examples *will* be given of computational systems that *do* reason about other computational systems.

The meta-system is called a *language processor* in the case where the description of the computational system is a program expressed in some programming language. A language processor can take on different forms. In this text we will focus on evaluators.

A language processor itself can be composed of a program (the processor program) processed by another processor.

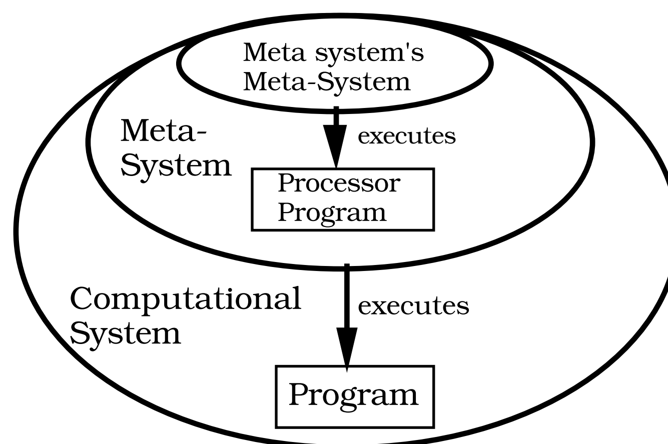


Figure 2.2

In the special case where the language in which the processor program is implemented, is the same as the language implemented by the processor program, the processor program is called *meta-circular* [3] [Abelson&Sussman84].

Programs that describe computational systems that manipulate other programs are often termed *meta-programs*. The program describing a programming environment is an example. The programs manipulated by meta-programs are called *object-level programs*; a relative notion, of course: object-level programs, in their own turn can be meta-programs.

The program of our meta-system is a special sort of meta-program. It is special since it is *the* meta-program that describes how to turn programs into computational systems. The architecture where the program of the meta-system is explicitly available for inspection and modification is a particular instance of a *meta-level architecture*. It has the advantage of being able to modify the meta-system prior to (or even during) the execution of a program.

---

2 It is the author' s conviction that much of the misunderstandings about reflective programming languages comes from a lack of distinction between computational systems and representations or descriptions of computational systems.

3 This process of decomposition can be repeated ad infinitum for meta-circular processor programs. In the literature about reflective systems this ' tower of meta-circular processors' is taken as the basis to introduce reflective programming languages, giving a slightly different view on reflection as it is introduced in this text. See [De Volder&Steyaert94] (also in appendix) for a discussion on the topic.

In general a meta-level architecture is an architecture where the meta-system can be acted upon. In the above case this is done by acting upon the description of the meta-system. Below we will see an example where this is done by acting directly upon the meta-system as is, i.e. acting upon it as a computational system rather than on its description.

Finally, we presume that a computational system has a well-defined interface, called the *base-level interface*, by which its behaviour can be invoked. For example the base-level interface of a language processor comprises the evaluation function.

2.3 Absorption and Reification in Programming Languages

The main reason for constructing reflective systems is efficiency and modularity in the structures used in representing a computational system. Typically only part of the system can be explicitly encoded in such a representation. A substantial part of the system' s behaviour remains implicit in the internal relations between elements of the representation, the process that interprets this representation and the circumstances in the world in which the system is embedded. This is not only an essential characteristic of such representations. It is also an integral part of being able to efficiently express the behaviour of computational systems, if we take a representation in which every aspect of the computational system must be explicitly encoded as an inefficient representation. On the other hand this characteristic puts limits on the generality and power of the underlying representational system. Not all systems can be expressed in an equally efficient way.

This may all seem to be in want of a more thorough explanation, and indeed it is. But, an in-depth analysis of representation, efficiency, and reflection in the most general case is certainly beyond the scope of this dissertation, and would necessarily have to follow the analysis that can be found in [Smith86]. Rather than doing that, we will analyse these concepts in the more restricted case of programming languages and computational systems described by programs expressed in a programming language.

Programming languages are used to describe *implementations* of computational systems. They do so by giving a means to express the internal workings of a computational system that is relatively close to executable code. What differentiates one programming language from another is how the internal workings are expressed. By this we don’t mean the syntactical differences between one programming language and another, but rather the notable differences of how a system is divided into subsystems. How a system is divided into subsystems is determined by the kinds of abstractions (e.g. procedural abstraction, data abstraction) or programming concepts [Maes87a] that are supported by the programming language.

In the discipline of programming language design, one speaks of programming paradigms as those classes of languages that support fundamentally different programming concepts [Wegner90]. The major programming paradigms are: procedural programming, object-oriented programming, concurrent programming, functional programming, logic programming and rule-based programming.

Programs expressed in exemplar programming languages of the different programming paradigms will exhibit different characteristics. They will differ essentially in what aspects of the internal workings of a computational system can be left implicit, and what aspects must be explicitly encoded. For example, it is obvious that for a backtracking problem (e.g. the 8-queens problem) expressed in a procedural programming language, the flow of control that is typical for backtracking must be explicitly encoded, whereas in an implementation in a logic programming language this can be left implicit.

In a program where a certain aspect of the internal workings of the implemented computational system is left implicit, we say that this aspect is *absorbed* (by the programming concepts of the programming language). When it is made explicit we say that it is *reified*. Efficiency, in terms of how concise a computational system can be expressed[4], is defined as the amount of detail that can be absorbed in the implementation language.

Obviously, it is not possible to give a total ordering of programming languages according to this kind of efficiency. Not just because we can only speak about efficiency for implementing a certain system (or a set of systems that belong to a particular problem domain if we are a bit liberal), but also because even within one system, conflicting demands with respect to the programming paradigm can coexist.

Notice that not all aspects of a computational system that can be absorbed in the implementation language also need to be absorbed. This is, for example, the basis on which different language interpreters (in most cases preferably meta-circular interpreters) are compared. A meta-circular interpreter for a Scheme-like language can choose to absorb or make explicit different aspects of the underlying structure of the Scheme language (see also [Abelson&Sussman84], [Maes87a]). The entire evaluation function can be absorbed by falling back on the meta-level programming facilities of Scheme, i.e. by using the explicit evaluation function of Scheme in Scheme. Or, the evaluation function can be implemented in terms of expressions and environments, thereby absorbing continuations and consequently the explicit encoding of the flow of control. Or, the evaluation function can be explicitly encoded with expressions, environments and continuations, but leaving implicit storage handling for lists. Or, an evaluation function can be constructed that makes explicit all machine actions performed by a hypothetical, or real processor. All these differences become relevant in case one wants to reason about or alter this implementation. As we will see in a moment, it is exactly these differences that will determine the theory with which we will be able to reason about our language implementation.

Within one and the same programming paradigm, also, differences exist between programming languages regarding their abilities to absorb implementation aspects of computational systems.

One set of examples are facilities such as garbage collection, persistency aspects of data, scoping issues, modularity etc. (in a mind boggling way, reflection itself can be added to this list, see also [Maes87a]). These facilities are generally considered as programming concepts that can, or should, be added orthogonal to most of the above programming paradigms. Languages that include these facilities have a larger capability to absorb implementation details.

---

4 We hesitate to use the term expressivity here. It is not clear whether expressivity, in its normal usage of "expressivity of a programming language" applies to the efficiency in expression or generality in expression.

Other more specific examples are (lack of) refinements of existing programming paradigms, or programming languages. We will discuss one example that is by now part of the folklore of object-oriented reflection (example from [Kiczales,des Rivières&Bobrow91]). Consider a computational system in which we need to represent data elements that are composed of other (named) data elements. In the object-oriented paradigm it is customary to implement such a data element as an object. The composition structure is reflected in the instance variables the object has. However, most object-oriented languages only provide facilities for representing objects with a small number of instance variables, all of which typically have a non-default value. Sometimes we need to implement a compound data element that has a possibly large number of components of which a large number has a default value for the major part of the object's life-time. Such a data element must be explicitly encoded as a dictionary for example. Only an object-oriented language that has the facility to represent objects with a large number of instance variables of which only a few have a non-default value, can absorb the implementation of this sort of data elements.

So, we observe that programming languages have different potential to absorb implementation details of a computational system, going from large grained programming paradigms, to more fine grained orthogonal sets of language features, to fine grained specific refinements of certain language features. Whereas the efficiency of programming paradigms is very hard to compare relative to each other, within one paradigm it is possible to compare the absorption capabilities of different language features.

One could be tempted to conclude that the more that can be absorbed by the programming language the better programs can be expressed, and thus that programming language design has as its goal the design of programming languages with ever better absorption capabilities. There is a catch however. It has the form of a trade-off between efficiency in expression, and generality of programs expressed in a programming language. Stated otherwise, the more that can be absorbed by the programming language the less general programs expressed in such a programming language tend to be. This is illustrated by the following example (example due to [Agha90]).

Consider writing a program that calculates the product of values that are stored in the leaf nodes of a tree. When expressed in a programming language that supports recursion, a substantial part of the control flow of this program can be absorbed by the programming language. The return stack of procedure calls can remain implicit. Such an encoding is more efficient than an encoding where we explicitly need to keep track of the visited nodes. It is less general, however, since we are unable to express, in a simple way, the fact that when a leaf node with the value '0' is encountered, the entire computation can stop and return the value '0' as a result. In an encoding with an explicit return stack, this *can* be encoded simply by emptying the control stack.

It is true that, in the above example, programs are not forced to use all the facilities (i.e. recursion) given by the programming language. Then again, if programs do not use such facilities out of fear of loss of generality, then why provide them ? One could also say that when such features are given, then all the complementary features to recover the loss of generality must be provided as well. For the above example this means that recursion must be complemented by a feature to 'jump out' of recursion. This, however, leads us to a (very old and often held) discussion on efficiency and generality on the level of programming languages, i.e. a small, concise programming language definition for a less general programming language versus a large, less concise programming language definition for a general programming language.

2.4 Open Implemented Computational Systems

So we seem to be stuck with an apparent contradiction between generality and efficiency. This need not be the case. What we truly wanted, in the above example, is a mechanism where the control stack can be left implicit until it is really needed. At that moment the control stack is made explicit, it is emptied and given back to the implementation to be absorbed. In general we need a mechanism where aspects that are absorbed in the underlying structures of the implementation language can, at any point in time, be made explicit, modified, and absorbed back again in the implementation of the programming language.

A mechanism is needed to inspect and alter the implementation structures of a programming language. Rather than tackling the question of how a program can inspect and alter the implementation structure of its own underlying implementation language, we will first tackle the question of what it means for one system to reason about the implementation of another system[5].

Computational systems, either programming language processors or other systems, that give access to their implementational structures are not new. Systems that provide, for example, facilities to test whether some extension of the system is available and how to use it, facilities for testing what version of the system is running, facilities for setting and testing parameters of internal data-structures (e.g. buffer-sizes, block-sizes, … in the area of operating systems) can be found in abundance.

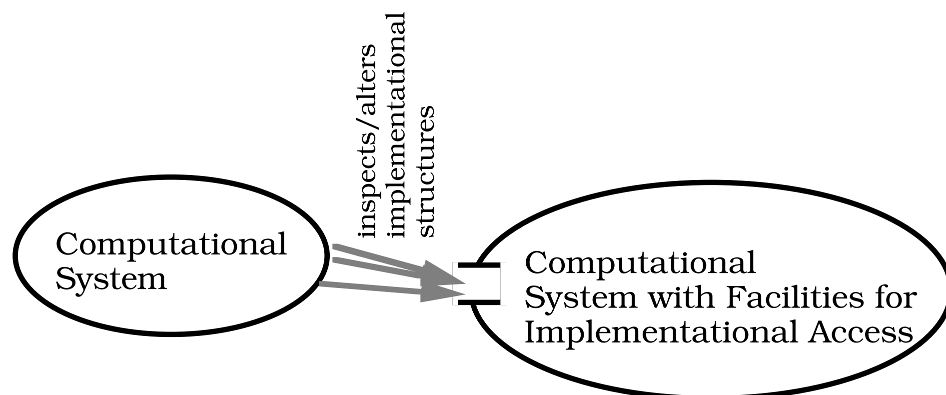


Figure 2.3

In the case of programming languages, access to implementational structures means that one can, for example, inspect the control stack or the variable binding environment, and that one is able to change these or hand back a modified version, so that the changes are reflected in the further execution of the program (i.e. in a causally connected way). Access takes the form of operations such as 'get-environment', 'put-environment' that are defined for the language's evaluator. These facilities are, in most cases, the basis for implementing debugging systems, or can even be put to use to partially solve the problems discussed in the previous section.

5 [Rao91] uses the term "implementational reflection" for inspecting and/or manipulating the implementational structures of other systems used by a program. We prefer to restrict usage of the term reflection to systems that reason about themselves.

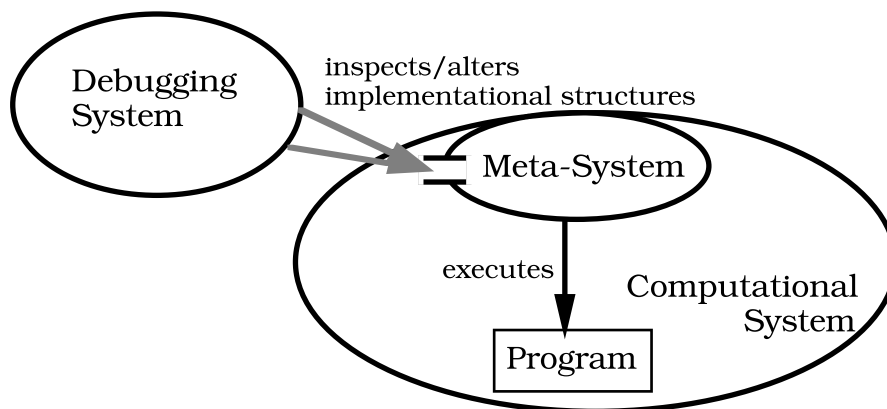


Figure 2.4

All of the above are limited cases of implementational access. First of all, it is not always clear whether such facilities are part of the 'ordinary' usage of the system, or whether they are 'special' in the sense of revealing part of the system's implementation. Moreover, they do not solve all problems of the above section.

For example, in order to solve our problem of absorbing the representation of composite data elements with a large number of components, it is not sufficient to be able to inspect, and possibly change the implementational structures of an object. Rather, an alternative implementation of objects is needed. It is not sufficient to be able to inspect all the instance variables, nor is it sufficient to be able to add or delete instance variables. An instance variable, even if it has a default value, *is* an instance variable that must be represented in the object. What is needed is that, for this particular kind of objects, we can override the mechanism to look up instance variables.

In the general case, a more structured, 'open-ended' access to a system's implementation[6] must be provided. Presuming that a computational system has an interface, the base-level interface, that shields its users from the implementation details that are involved in realising the system, and that is used by all users of the system, we can define the following:

> ***Open Implementations*** *[Rao91] : A system with an open implementation provides (at least) two linked interfaces to its clients, a base-level interface to the system's functionality similar to the interface of other such systems, and a meta-level interface that reveals aspects of how the base-level interface is implemented.*

The idea is that a user of an open implemented system can, by means of the meta-level interface, have a substantial influence on the implementation, and accordingly, the behaviour of the system. The notion of open implementations was introduced by Rao in [Rao91], where an open implementation is given of a windowing system that allows the exploration of different window system behaviours and implementations. The base-level interface of the windowing system is, obviously, an interface that allows the opening and closing of windows, dragging, generating pictures in windows, etc. The meta-level interface allows, for example, for the definition of new windowing relationships (such as window, sub-window relations).

6 The difference between plain implementational access, and structured , open-ended access to a system's implementation is parallel to the difference between reflective facilities and reflective architectures [Maes88].

A programming language with an open implementation will be called an *open implemented programming language*. A well-designed open implementation of respectively our object-oriented programming language and our recursion supporting language can, in principle, solve the respective problems of object representations and access to the control stack of the previous section.

Consider the problem of representing composite data elements with a large number of components. Any well-designed open implementation of an object-oriented programming language (the CLOS meta-object protocol is such an example [Kiczales,des Rivières&Bobrow91]) will provide a meta-level interface that allows alternative implementations for object representations. An object representation can be implemented in which only the instance variables with non-default values are stored.

In an open implementation the meta-level interface specifies points where the user can provide alternative implementations. Such an alternative implementation can differ from the default implementation of the system in performance characteristics, or it can alter the behaviour of the system, or it can extend the system with new behaviour. The extent to which the behaviour of the system can be altered, or extended, depends on the meta-level interface and its link to the object-level interface. To illustrate this we will consider two example open implementations.

Example 1: A Meta-function for a Scheme-like Language

An evaluator for a Scheme like language can be expressed as a dispatcher on the type of expression to be evaluated. Each expression is tagged with an expression type. A tag can for example be an atom at the head of each list that represents an expression. Typical tags for expression types are ' lambda' , ' if' ,…. This tag is used by the dispatcher to invoke an appropriate evaluation function. For the above listed tags these evaluation function would respectively be a function to construct a closure, evaluate an if expression, ….

A useful open implementation would be one in which clauses can be added to this dispatcher, thereby allowing to add new expression types and their corresponding evaluation function. An extension to the dispatcher can be formulated as a list that associates tags to evaluation functions. The open implementation takes the form of a function (the meta-function) that has such a list as argument and returns an extended evaluator.

The base-level interface of this simple open implementation is the evaluator. The meta-level interface is the above meta-function. Base and meta-level interface are linked by the fact that the meta-function, given an appropriate extension to the dispatcher, returns an extended evaluator as a result.

Note that this open implementation implements many different variants of the Scheme programming language. Each particular usage of the meta-level interface *engenders* a different variant.

Example 2: A Class Hierarchy for a Scheme-like Language

Alternatively, a Scheme-like language can be implemented in the form of a class hierarchy in some object-oriented programming language. In this case expressions, lists, closures, and all other components of the evaluator are expressed as objects. To a certain degree the class hierarchy, to which all these objects belong, exposes aspects of the implementation of our evaluator. This has much to do with the often talked about code-reuse facilities that come with object-oriented programming.

To turn this class hierarchy into a true open implementation, however, we need to explicitly identify the base and the meta-level interface, and the link between both. In casu, the base-level interface will have the form of a protocol to which, for example, all objects representing expressions must conform, thereby establishing a contract between implementors of the classes that are used for instantiating ' expression objects' , and users of these objects (e.g. users that invoke the evaluator). The meta-level interface will be expressed as an interface with which new classes that implement expression objects can be added to the class hierarchy, or with which expression objects themselves can be added to a program representation such that they can be used in combination with the already existing expressions.

Not only the protocol of expression objects needs to be specified. All other objects that are part of the implementation may play an important role in the division between base and meta-level interface. The result of such an identification and specification of protocols is called a framework in object-oriented terminology; in the reflection community the term meta-object protocol is used.

This open implementation, just like the previous one, defines many different flavours of the Scheme programming language.

Both of the above open implementations give rise to the meta-level architecture as depicted in figure 2.5. In this meta-level architecture it is possible for a meta-program to act upon the meta-system prior to, or during execution of a program.

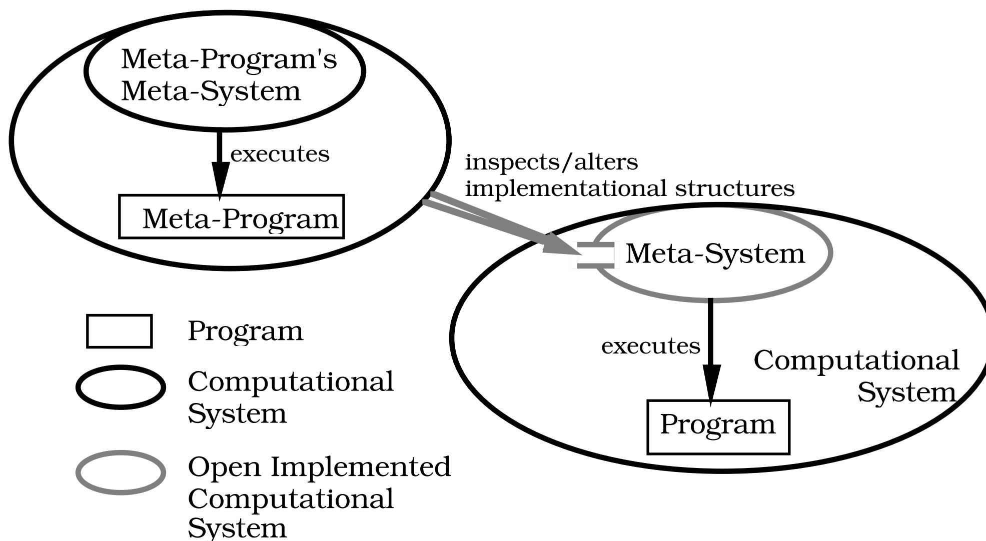


Figure 2.5

What differentiates this architecture from 1) the debugging system of above and from 2) a meta-level architecture where *the program* of the meta-system can be acted upon, is the structured access it provides to the meta-system. In the first case access to the meta-system is too limited (as already shown above); in the second case access to the meta-system is too “unlimited”[7]. Especially the difference with this latter is important.

If modifications to the meta-system’s program are allowed, then the result can be

[7] We can actually define a continuum with four marker points: a ‘plain’ evaluator without access to the implementation, an evaluator with implementational access, an open implemented evaluator and an explicitly encoded evaluator.

just about anything. It is the programmer that explicitly modifies the meta-system. And this is a matter of text-editing the meta-system's program-text.

When the meta-level architecture is based upon an open implementation, it actually is another computational system (admittedly, one programmed by the programmer, but still a separately identifiable computational system) that accesses and modifies the meta-system. It does so by using the meta-level interface to extend the meta-system with functions or objects (*not* program text but first class values !). The meta-level interfaces of the meta-system constrains the sort of modifications that can be done.

Finally, note that in the above meta-level architecture, the meta-program explicitly handles implementational structures of the meta-system used to execute a program, i.e. structures that are implicit for that program. So, what is explicit for the meta-program is implicit for the object-level program. As already mentioned before, a system' s implementation itself also absorbs and reifies certain aspects of the implemented system (cf. the different meta-circular interpreters of the previous section). This obviously puts a limit on what is possible with open implementations.

In conclusion we can say that by opening up the implementation of a computational system it is possible to have, to a certain degree, both efficiency in expression and generality. In order to open up the implementation we need to identify a base and a meta-level interface. The meta-level interface is used to alter and/or extend the behaviour of the system. An open implementation can be used to construct a particular kind of meta-level architecture, in which the meta-system can be modified in a controlled manner.

We considered the special case of *open implemented programming languages* i.e. programming languages that have an open implementation. For an open implemented programming language we observed that they implement not one but many different languages, according to how the meta-level interface is used. These are called the languages *engendered* by the open implementation.

2.5 Reflection: Accessing One's Own Meta-system

The above meta-level architectures have in common that one computational system acts upon the meta-system of another, in any other way unrelated, computational system. Meta-level architectures of this kind have their practical applications, even in the area of programming languages. For example in [Kiczales93] a CLOS open implementation for a Scheme compiler is briefly mentioned.

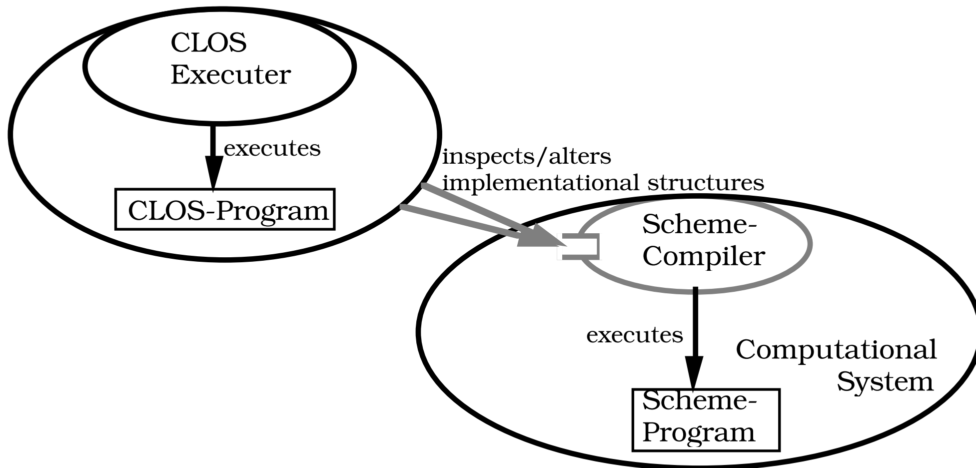


Figure 2.6

A more specific kind of architectures can be studied, i.e. that of *reflective systems*. Since a program is turned into a computational system by a meta-system, we can ask ourselves the question how and under what conditions a computational system described by some program processed by a meta-system can be given access to, use and alter the behaviour of its own meta-system (figure 2.7).

We will consider three forms of access to the meta-system: 1) access to the base-level interface of the meta-system, 2) implementational access to the meta-system, and 3) access to the meta-level interface of an open implemented meta-system. For the special case of programming languages the first will result in a special kind of meta-programming, the second in programming languages with *reflective facilities*, and the third in programming languages with a *reflective architecture*. The last architecture in this list being the most interesting one.

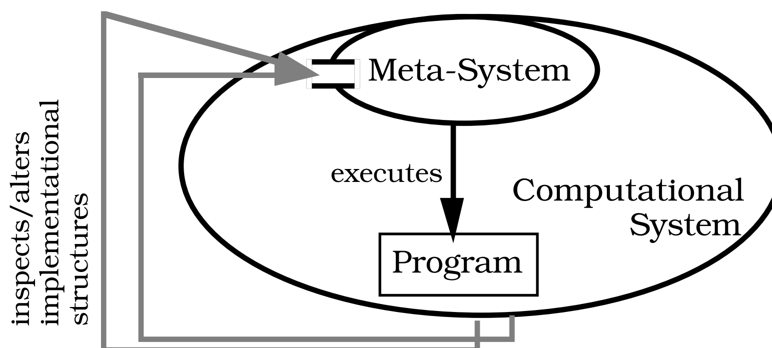


Figure 2.7

Not every open implementation is suitable as the basis for a reflective architecture. We will consider the conditions that must be met. This will lead us to the definition of *open implementations with reflective potential*.

The question how this access can be given can be answered in general. A program can be given access to its meta-system by extending the programming language with the necessary *reflection operators*. Reflection operators are language facilities, offered by the programming language, that allow programs to access the meta-system with which they are executed. A language that is extended with a set of reflection operators, can be called a *reflective programming language*.

A programming language must be *extended* with reflection operators. The special case can be identified where the open implementation of the programming language is powerful enough to formulate this extension. The open implementation of Agora that will be given will be of this kind. In all other cases reflection operators must be added to the programming language in an ad hoc fashion.

A program that uses reflection operators to access its own meta-system can be called a *reflective program*. A reflective program is both meta-program and object-level program at the same time. As mentioned before, the meta-program can explicitly handle implementational structures that are implicit for the object-level program. Collapsing meta-, and object-level program into one reflective program may lead to *reflective overlap*, i.e. implementational structures that are both explicit and implicit in the same expression. Reflective overlap is a phenomenon that can not be observed in ordinary meta-level architectures. Sometimes reflective overlap is undesirable. We will discuss a technique to manage reflective overlap.

An issue related to reflective overlap is that of *meta-regression*. A reflective program is said to *reflect* when it actually uses reflection operators to access its meta-system. The part of the program that reflects must, by definition, be a meta-program. On the other hand in a reflective program, not only the object-level program can reflect, but also the meta-program itself can reflect since it is executed by the same meta-system as the object-level program. This process of reflection can go on ad infinitum. If so, the program is said to *regress infinitely*. We will talk about *static reflection* when the maximum number of levels the program regresses can be statically determined. When the number of times the program regresses is dynamically determined, the program is said to exhibit *dynamic reflection*.

2.5.1 Reflective Architectures

If the meta-system has an open implementation, then a program processed by this meta-system can be given access to the meta-level interface of the meta-system with the intention of altering its behaviour. This gives rise to *reflective architectures*.

For example, in the case of the above "meta-function" open implementation of Scheme, access to the meta-function (i.e. to the meta-level interface) can be given under the form of reflection operators that allow the "installation" of extensions to the dispatcher (see [Simmons II&al.92] for an actual example).

The conditions under which access to the meta-level interface of an open implementation can be given, are reminiscent of, but fundamentally different from, meta-circular language processors.

In an open implemented programming language, two (kinds of) languages are of importance. First, the language in which the open implementation, and consequently all code that is added to this open implementation by means of the meta-level interface, is expressed. And second, the languages that are being implemented (not one but many, according to how the meta-level interface is programmed). We will call the former language the meta-level language of the open implementation, and the latter will be called the languages engendered by the open implementation. The meta-level language and the engendered languages need not be related. Even in practice examples can be found where it is advantageous to have a meta-level language that totally differs from the engendered languages. For example in the above CLOS open implementation for a

Scheme compiler these languages are different.

Since programs are processed by the open implementation, they are expressed in one of the engendered languages, and since the meta-level interface is coded from within such a program, a class of 'special' open implemented programming languages needs to be identified. This is the class of open implemented programming languages for which the meta-level can be programmed in *any* language engendered by this same open implementation. This class will be called the class of *open implemented programming languages with reflective potential*.

How can we construct such 'special' open implementations ? Notice that, in contrast with e.g. a plain evaluator, it is not possible to talk about a meta circularly implemented open implementation, exactly because an open implementation can be used to engender many different languages[8] (whereas a plain evaluator engenders only one). A meta circular open implementation would have to pick one engendered language as being preferred, excluding all the rest for programming the meta-level. For a reflective programming language this would mean that only the 'vanilla variant' of the programming language can be used for reflective programming, excluding all languages engendered by reflective programming, themselves, to be used for reflective programming. This latter ability, however, is considered as an essential characteristic of reflection [Smith82].

What is needed to construct an open implemented programming language with reflective potential is that all first class values (primitive values, functions, objects, …) can freely travel between implementation language and engendered language, and that both languages can transparently use each others first class values. Such a construction is called a *linguistic symbiosis* [Ichisugi&al.92] of the implementation language of the open implementation and possible engendered languages. A detailed description of such a construction for Agora will be given in a subsequent section.

2.5.2 Reflective Facilities

In the weaker case where the meta-system only allows access to its implementational structures (and is not a full-fledged open implementation), the system can only be extended with *reflective facilities*.

Reflective facilities usually take the form of two sets of operators. One set of operators to read the implementational structures of the meta-system. And one set of operators to overwrite the implementational structures of the meta-system. In the former case one speaks of *reification* [Friedman&Wand84]; in the latter the term *absorption*, or *deification* is used. Typical examples include reflective facilities to get access to and alter the variable binding environment or the control stack. See [Jagannathan&Agha92] for a fairly complex example of the usage of reflective facilities.

Reflective facilities often give rise to *reflective overlap*. Reflective overlap occurs when a part of the implementational structures of the meta-system is both reified (explicit) and absorbed (implicit) at the same time. Take for example a language with reflection operators 'get-environment' and 'put-environment' to reify and absorb environments (i.e. the reflective variants of the operators for implementational access from the previous section). Here, reflective overlap is most noticeable when environments are reified in a causally connected way, i.e. in a way such that modifications to the reified environment have an effect upon the

8 This is in fact a manifestation of what is called the causal connection requirement [Smith82].

executing program' s implicit environment. Stated otherwise, the environment that is made explicit is shared by the program in which it is made explicit and the meta-system. Environments that are reified are both explicit to the program and implicit in the meta-system. In such a case one speaks of reflective overlap. In our example this is apparent by the fact that the variable that holds the environment is also part of that environment.

This sort of reflective overlap can be disturbing. In the above example the programs that want to manipulate and alter environments must be careful not to destroy or alter their own variables for example. Reflective architectures often provide better mechanisms to control reflective overlap. It must be stressed, however, that not all cases of reflective overlap need to be avoided.

2.5.3 Meta-Programming

In practice programs are not executed by assembling expressions by hand, and then making an explicit call to the evaluator. Rather, an entire set of programs (or to be correct: computational systems) is available to construct and execute programs.

Programs that describe computational systems that manipulate other programs are often termed meta-programs. The program of our meta-system is such a meta-program. It is often desirable to construct one' s own meta-programs. Constructing a programming environment is an example.

To support meta-programming the base-level interface of the meta-system can be made available to programs executed by the meta-system. This enables the construction of meta-programs such that the processing of programs is absorbed, i.e. the language processor is not explicitly encoded. The difference[9] with the case where the language processor is explicitly encoded as a meta-circular processor, is that in that case we are using two *different* meta-systems; even though they have a possibly similar representation (i.e. one is meta-circularly defined in the other).

In our Scheme example access to the base-level interface, means that a Scheme program can explicitly invoke the underlying evaluator. A feature that is available in most Scheme implementations.

Finally note that in a reflective architecture, the code that is used in programming the meta-level interface is typically a meta-program, manipulating pieces of its own program, and applying the evaluator upon these program pieces. Meta-programming and reflective programming often go hand in hand.

---

[9] In a more general account of reflection this distinction would be made on the basis of causal connection, i.e. the difference would be made on the basis whether programs are executed by an executer program that is causally connected to the meta-system or executer programs that are not causally connected to the meta-system. Given our modest goals, our differentiation between the two is based on our consequent distinction between programs and computations.

2.6 Managing Reflective Overlap and Tower Architectures

What differentiates access to the meta system through reflection operators from 'ordinary' access to the meta-system is its dynamic character. The meta-system is accessed from within an executing program, and therefore meta-program and program possibly execute in the same execution environment. On the other hand, since meta-programs may actually reify and act upon this execution environment this may lead to reflective overlap. It is exactly this aspect that is the most difficult to manage in a practical setting.

A general recipe to avoid reflective overlap is closely connected to the notion of tower architectures [Smith82]. Conventionally, tower architectures are based on the notion of towers of meta-circular processors. Since our discussion on reflection is based on open implementations rather than meta-circular processors, we will briefly discuss the notion of towers of open implementations, and how they can be used to avoid reflective overlap. A more thorough discussion can be found in [De Volder&Steyaert94] (also in appendix).

The general idea is to provide a vantage point on which to stand when reasoning about one's own meta-system. Moreover, a vantage point that is similar in nature to the vantage point one system has when reasoning about another system (as in figure 2.5).

Reflective tower architectures mimic the fact that meta-programs execute in their own execution-environment. Actually, a tower of execution environments is needed rather than a single one, since in a true reflective system, not only is it possible to reflect on the meta-system of 'ordinary' programs, but also is it possible to reflect over the meta-system of meta-programs, and so on. What is actually mimicked is the infinite ascending chain of figure 2.8.

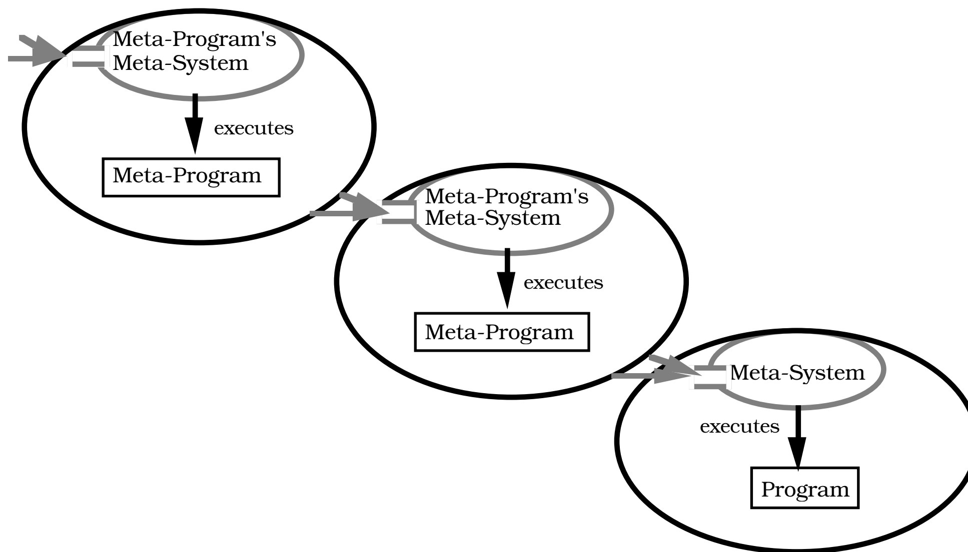


Figure 2.8

A couple of notes should be made here. The first is that in an actual implementation this infinite ascending chain can not be realised literally. In an actual implementation all meta-systems in the chain are one and the same (i.e. the open implementation with reflective potential), only a chain of execution states is realised. This chain of execution states conforms to those parts of the reflective program that can be identified as meta-programs. This leads us to the second remark. What is the nature of meta-programs, and how can they be

identified, if they are part of the object-level program ?

In the conventional tower model meta-programs mostly take the form of meta-circular processor programs. In the tower model based on open implementations, meta-programs are generally programs that use the base and meta-level interface of the open implemented meta-system. A typical meta-program is a program that first uses the meta-level interface to alter the behaviour of the meta-system, and then uses the base-level interface of this altered meta-system to execute a program. Whether meta-programs can be identified as such depends on the exact nature of the reflection operators. It should be kept in mind however that in order to avoid reflective overlap, meta-programs should be distinguishable as separate entities in a reflective program. Examples will be given in later sections.

2.7 Computational Systems with an Open Design

In the previous section we saw that open implementations can be used as a basis to construct reflective systems. In this section we argue that for programming languages mere open implementations are not enough. We will analyse what is wrong with open implementations as a basis for safe and fully expressive reflective programming languages and introduce the improved concept of *open designs*. We will analyse the question of *abstract representations* in programming languages. The discussion on how to construct open designs is deferred till the section on object-oriented frameworks.

A computational system with an open implementation does not define a single system but an entire design space of (related) systems. The behaviour of a system with an open implementation can be altered through the meta-level interface. Although the meta-level interface puts constraints on the extent to which the behaviour of the system can be altered, merely opening up a system' s *implementation* gives no guarantee that the so created design space is coherent or is the intended design space.

For an open programming language, for example, we could want such a design space to cover all languages belonging to the same programming paradigm. Merely opening up the implementation of a particular programming language does not guarantee that the intended design space is covered, nor does it guarantee that no languages out of the intended design space can be reached. The former obviously is important, having to do with expressivity, but also the latter, having to do with safety and the ability to reason about programs.

Take for example the design space of pure functional languages. In opening up a particular programming language we need to make sure that we make explicit the *important* language concepts. In case of our functional programming language obviously what needs to be made explicit is the concept of a function. In an actual implementation of a functional programming language this notion need not be explicitly represented. The implementation is not necessarily a good source for finding the important language concepts.

Furthermore it is equally important to identify the *constraints* surrounding the language concepts that are made explicit. Take for example an open

implementation of a pure functional programming language, where one can provide one' s own function representation. Here, an important constraint is that functions stay pure, i.e. that one does not introduce function representations that can be used to construct functions that are not referentially transparent. Certainly such constraints will not be found in an implementation, they may not even be enforceable by an implementation.

Finally, one must see to it that the concepts that are made explicit are represented *abstractly* enough. Or vice versa, that they are not represented too operationally. Take the above example again. Functions might be made explicit as closures, i.e. as a record with three fields: formal parameters, body and lexical context. Clearly closures are a representation of functions highly inspired by a particular implementation. Obviously such a representation does not conform to the truly abstract notion of a function, i.e. something that uniquely associates each input argument to an output argument. In case where the important language concepts are too operationally defined, it is possible that not the entire intended design space is covered.

A system with an open design differs exactly in the above points from a system with an open implementation. Rather than elaborating on these issues in general terms, we will explore them in more restricted settings.

The notion of abstractness will be explored in the particular setting of programming languages. We will see that in the area of programming language semantics, these issues have already been investigated, and that a set of objective criteria to test abstractness has already been developed for semantic definitions. These criteria will be adopted.

The questions how to make explicit the important design issues and what forms the constraints can take, will be discussed in the particular area of open object-oriented programs. As already said, this discussion is deferred to the section on object-oriented frameworks.

2.8 Full Abstraction and Compositionality in Programming Languages

> *"Programs are not text; they are hierarchical compositional structures and should be edited, executed and debugged in an environment that consistently acknowledges and reinforces this viewpoint."*
>
> **Teitelbaum & Reps (1981)**

> *"The meaning of a sentence must remain unchanged when a part of the sentence is replaced by an expression having the same meaning."*
>
> **G. Frege (1892)**

As said in the previous section, criteria are needed to test whether an implementation or design is defined abstract enough. For programming languages, and in particular in the domain of programming language semantics, such criteria have been defined. They are called full abstraction and compositionality. Full abstraction and compositionality are two complementary constraints. The latter ensures an abstract representation of expressions, the former an abstract

representation of the first class values of the programming language. In this section we will discuss how these mechanisms can be adapted to the context of programming language implementations.

2.8.1 Full Abstraction and Compositionality in Semantics of Programming Languages

Although different kinds of semantic definitions exist for programming languages, it is possible to identify a common set of evaluation criteria. Semantic definitions are primarily judged by their soundness and completeness and the possibility to prove this. Any semantical description that lacks one of both is very questionable. Other evaluation criteria for semantic descriptions include formality, mathematical rigour and intelligibility. Although all of the former criteria are important, they are of lesser interest to us since they give no constructive indication on how semantical descriptions should be structured, and these criteria are not directly applicable to the implementation of programming languages. However, we will discuss two other criteria, compositionality [Frege92] [Tennent91] and full abstraction [Tennent91], that can be interpreted in the context of implementation of programming languages and can be used in a constructive way.

Throughout this discussion, and to illustrate it, we will adopt a denotational style of semantics. In general we define the semantics as a function that maps elements of the syntactic domain to the semantic domain. Each expression is mapped to its “meaning”.

```
μ: Syntactic Domain -> Semantic Domain
```

Compositionality
A semantic description is compositional if the meaning of composite expressions is expressed as a function of the meaning of its immediate subexpressions. For a compositionally defined semantics one can say that, in a composite expression, subexpressions can be substituted by semantically equivalent subexpressions without changing the meaning of the composite expression.

> ***Compositionality***: *A semantic definition is compositional if two semantically equivalent expressions X and X′ (i.e.* $\mu(X) = \mu(X')$*); in each program context (..._...) where X can be used, X′ can be substituted such that:* $\mu(\ldots X\ldots) = \mu(\ldots X'\ldots)$. *[Tennent91]*

A compositionally defined semantics should not be confused with a semantics that is defined in a recursive compositional style (e.g. [Smith82]). In the latter the meaning of a composite expression is defined as a function of its immediate subexpressions, and not necessarily of the *meaning* of the subexpressions. Although in this latter kind of semantics the compositional nature is still an important issue, it does not have the above discussed property of substitutability of semantically equivalent expressions. An example of a semantics that is defined in a recursive compositional style but is not compositionally defined (example taken from [Smith82]) is the semantics of a Lisp-like language with a (one argument) quote expression, in which the meaning of this quote expression is the quoted expression. Clearly the semantics of the quote expression is not a function of the *meaning* of its subexpression.

Compositionality is crucial in proving properties of programs. It allows inductive reasoning about the structure of programs, i.e. to prove a property one proves the property for all non-composite (primitive) syntactic structures first, then the

property can be inductively proven for each composite expression on the hypothesis that the property holds for the immediate subexpressions.

Full Abstraction
Semantic definitions can be classified according to how operational, or vice versa, how abstract they are. A "too operational" semantic definition is one that makes too much distinction in assigning meaning to expressions; abstractly equivalent expressions are assigned different meanings. An abstract semantics maps abstractly equivalent expressions to the same meaning.

*A **fully abstract semantics** is a semantics that considers those expressions as equivalent that are indistinguishable in any program context. Consider two expressions X and X'. X and X' are indistinguishable if for all program contexts (…_…) it can be observed that:* $\mu(\ldots X\ldots) = \mu(\ldots X'\ldots)$*. A semantic description is fully abstract if for all indistinguishable X and X' it is true that* $\mu(X) = \mu(X')$*. [Tennent91]*

The difference between a semantics that is too operational and an abstract semantics is best illustrated with an example. Consider defining the semantics of an object-oriented programming language that supports strong encapsulation of objects.

On the programming level, strong encapsulation means that if we have two objects with the same behaviour, then these two objects are indistinguishable, according to our definition of indistinguishability above, regardless of how the behaviour of both objects is realised. For example two strongly encapsulated objects can have different private attributes and still be indistinguishable.

The semantic domain will consist mainly of a representation of objects. A semantic description where objects are, for example, represented as a couple (public methods, private instance variables), would be called too operational. Objects in this case can be considered semantically different on the basis of their private attributes. In contrast, a fully abstract semantics must consider all objects with the same behaviour as semantically equivalent.

The question of abstractness of the semantics boils down, in this case, to the question of how well the semantical representation of objects supports the notion of encapsulation. The question of whether such semantics exist remains open, however, and certainly lies beyond the scope of this work.

As Tennent [Tennent91] points out, the practical significance of full abstraction is that: if the semantics unnecessarily distinguishes the meaning of expression P and P', an axiom asserting the equivalence of P and P' could not be validated by the semantics, and an axiom asserting that P and P' are *not* equivalent might incorrectly be regarded as sound. This means, as can also be observed in the above example, that the semantics is too fine-grained in distinguishing values.

Full abstraction and compositionality are two complementary constraints. The latter assures an abstract representation of expressions, the former an abstract representation of the semantic values.

2.8.2 Full Abstraction and Compositionality in Implementation of Programming Languages

The notions of full abstraction and compositionality are informally applicable to the implementation of programming languages. First remark that an implementation of a programming language in general involves more than the simple execution of programs. The different components, i.e. compiler, evaluator, program browser, program debugger, type checker, … of an entire programming environment must be taken into consideration. Although we will focus on the execution of programs, it is important to keep in mind that all that will be discussed is part of a larger whole, i.e. the programming environment.

There are two major mechanisms to execute a program. The first is pure interpretation, i.e. the program is executed by direct inspection of the internal representation of the program. The second is pure compilation, i.e. the program is translated to a form that is directly executable by the hardware. Hybrid forms exist, whereby a program is first translated to some intermediate program code which is then interpreted (by a virtual machine).

In this text we concentrate on pure interpretation. The issue of compilation of open programming languages is outside the scope of this dissertation and is left untouched. We refer the reader to [Asai,Matsuoka&Yonezawa93] [Ruf93] [Kiczales&Paepcke93] for this matter.

Compositionality
A programming language evaluator takes a program representation as input and processes it to generate a result. It is a recursive process over the program representation that generates the result of evaluating the program. In an abstract form, it can be looked upon as a procedure, possibly involving side effects, taking an expression argument and returning a result from some type of answers:

```
Eval: Expression -> Answer
```

The compositionality criterium, as defined for the semantics of programming languages, can be adopted for programming language evaluators, albeit in an informal way.

An evaluator is compositionally implemented if for each composite expression this evaluator is implemented by means of the application of the evaluator to its subexpressions and its result depends only on the result of the application of the evaluator to the subexpressions. The evaluator may not depend on any other properties of the subexpressions.

The role of compositionality in the implementation of evaluators is extensibility. An evaluator can be extended for each new expression type that is added to the programming language in an incremental way.

Full Abstraction
In implementations in general it is hard to devise a criterium for what is a more abstract implementation and what is a more concrete implementation. In the case of an evaluator, however, the notion of full abstraction is such a criterium. In analogy with semantic definitions, the notion of full abstraction can be used as a criterium for the implementation of evaluators. Moreover this notion conforms to the notion of abstractness in implementations in general; i.e. in a fully abstract interpreter implementation details of the values that are manipulated by that interpreter are hidden.

An evaluator is a recursive process over the representation of a program. The result of evaluation is a value of some data type of values. It can be represented abstractly as a function of expressions to values. In analogy with full abstraction in semantic definitions, an interpreter is fully abstract if indistinguishable expressions X1 and X2, evaluate to indistinguishable values. Indistinguishability of expressions has already been discussed. Whether two values are indistinguishable depends, of course, on how these values are expressed (e.g. as data types). In general however this amounts to simple equality or some sort of behavioural equality (all operations that are applicable on the value type give the same results on indistinguishable values).

The importance of full abstraction at the implementation level lies, again, in the incremental extensibility of the implemented system. An example is indicated.

Consider implementing a functional programming language. In the implementation of the evaluator the question arises how functions are going to be implemented. We can make two apparent choices. The first is to implement a function as a record with three fields, i.e. the formal parameter names, the body and the lexical context of the function. Or we can implement our language level functions directly as functions that are available at the implementation level, i.e. as a function that takes a list of actual arguments. The formal parameter names, the body and the lexical context are encapsulated in this function. Notice that no circularity is involved here.

The second implementation is fully abstract: at the programming level functions can only be distinguished by observing their effect on all possible arguments, at the implementation level also functions can only be distinguished by observing their effect on all possible arguments. The first implementation of functions is not fully abstract. Functions can be distinguished at the implementation level by comparing their formal parameter names, whereas at the programming level the formal parameter names have no effect on a function's input-output behaviour.

The abstract implementation is more suited for extension. Consider adding a function type that is extensionally defined, i.e. a function is explicitly defined as a mapping of input values to output values. In the abstract implementation this function type can be added without modifying the implementation of function calling. In the non-abstract implementation the interpreter must be adapted to take into consideration this new function type.

It should be clear that the potential for constructing an abstractly implemented interpreter largely depends on the abstraction capabilities of the implementation language.

2.9 Conclusion

A program expressed in some programming language is turned into a computational system by means of a language processor (the meta-system). Not all aspects of the resulting computational system are explicit in the program. Some aspects are absorbed by the programming language concepts. We showed that due to this, there is trade-off between generality and efficiency in the description of computational systems.

Computational systems that allow implementational access try to improve on this trade-off by allowing a program to leave certain aspects implicit until they are needed. When needed, these aspects can be made explicit, modified and given back to the system to be absorbed. Computational systems that have an open implementation allow structured access to their implementation. They have an explicit meta-level interface by which a substantial part of the implementation, and accordingly, the behaviour of the system can be influenced. Language processors that have an open implementation can be used to construct a particular form of a meta-level architecture whereby the behaviour of the language processor can be customised prior to executing a program. Programs that alter the behaviour of a meta-system that executes a base-level program, are called meta-level programs.

Architectures in which programs can access their own executing meta-system have been studied. Different flavours of such architectures can be identified according to what aspect of the meta-system can be accessed. The particular kind where the meta-level interface of the meta-system can be accessed was called a reflective architecture. Open implementations with reflective potential allow the construction of reflective architectures.

Finally we discussed the difference between open implementations and open designs. The role of compositionality and full abstraction was discussed in the representation of aspects of programming languages.

In the next two chapters we will develop an open design for object-oriented programming languages. In chapter 5 we will come back to the issues of how to turn an open design into a reflective system.

Chapter 3

A Framework for Object-Based Programming Languages

3.1 Introduction

In this and the next chapter we consider all the different aspects of a framework for the implementation of object-based and object-oriented programming languages (OOPL). As is the case for frameworks in general, a framework for object-oriented programming languages, is a skeleton implementation of an object-oriented programming language. It represents a theory for how to design and implement object-oriented programming languages.

A framework is more than a mere implementation. Extensibility must be taken into account. The framework reifies the important constituents of an OOPL, such as objects, in an abstract way. Furthermore during the design of the framework we were led by the above discussed principles of compositionality and full abstraction.

A framework for a programming language in general, and for an OOPL in particular, must incorporate, in principle, all the different aspects associated with the implementation of that programming language. The different components, i.e. compiler, evaluator, program browser, … of an entire programming environment must be taken into consideration. We restrict our attention to the representation of programs and their evaluation. Since for object-oriented programming languages the result of evaluating a program is an object we will also consider the representation of objects.

The proof of the pudding is in the eating. This is especially so for object-oriented frameworks. We will test the framework by specialising it to different OOPL and by incorporating different language features. In this chapter we will confine ourselves to a very simple object-based programming language. Extension to a full-fledged object-oriented programming language will be done in the next chapter.

We will proceed as follows. We will first analyse the different design issues concerning OOPL. For our purpose we need a coherent framework of concepts that clearly delineates a design space of languages that can be called object-oriented. As we said in the introduction, examples can be found of different coherent frameworks of language concepts that are each called object-oriented. Our point of departure will be the notion of strongly encapsulated polymorphic objects that have a well-defined behaviour and interface. Inheritance will be briefly discussed, just enough to motivate that a thorough discussion on inheritance can be deferred to the next chapter. The argument for this approach is that at least for one particular kind, inheritance can be fully encapsulated in the internal implementation of objects. Furthermore this allows us to build a framework based on objects alone (no classes, no inheritance, no delegation), that can be specialised later on to include inheritance.

The next step will be the introduction of object-oriented frameworks. We will discuss what language concepts in the object-oriented paradigm support the construction of reusable programs and what must be added to construct open system. Reusability in object-oriented programs is partly based on inheritance. Since we deferred a thorough investigation of inheritance to the next section, only a simple form of inheritance will be used in our treatment of reusability.

A simple object-based programming language (Simple) is then presented for which the semantics is given in the form of a calculus for primitive objects. This calculus is discussed. It conforms to all of the previously adopted criteria. Although not fully formalised yet, it is mature enough to serve our purposes here. Similar calculi are being proposed in the literature [Abadi&Cardelli94] [Dami93] .

An initial set of abstract classes that form a framework is proposed. This initial framework incorporates our strong notion of objects. It is used to implement Simple. While trying to extend Simple, the limitations of this initial proposal are shown. The framework is adapted accordingly.

3.2 Design Issues in Object-Oriented Programming Languages

In the object-oriented research community there is still confusion as to what is essential for object-oriented programming, or even what it means to be object-oriented. There is no consensus about the central notion of what an object is, or should conform to. Different camps can be identified.

The Operations and State Camp

Probably the most popular conception of an object is that of a collection of operations that share a changeable state. The set of operations defines the interface of the object, and an object can only be accessed via the interface. The

state of an object is hidden for all users of the object, but shared by all operations of the object, i.e. state changes by one operation may be seen by subsequent executed operations. The operations are called methods, the hidden state is mostly realised by instance variables. The concept of a hidden state is a particular form of the concept of encapsulation.

An important correlated notion is that of *object identity*. In an OOPL that supports identity each object is assigned a unique identity. This identity is independent of the values of the object's hidden state. Identity is kept over state changes and can be used to uniquely refer to an object. Given two objects it can be tested whether they have the same identity. Sometimes objects can explicitly ask to change identity without changing state.

The definition of objects employed in this camp is a very operational one. It is too much directed towards imperative programming languages. The more abstract definition that will be employed in our work captures it as a special case.

The Classes+Inheritance Versus the Objects+Delegation Camp

One of the most debated issues of object-oriented systems is *inheritance*. From a *practical* point of view inheritance may be considered one of the most important contributions object-orientation has made. The whole notion of *software reuse* and of *incremental definition* of software systems has gained wide-spread acknowledgement due to the concept of inheritance.

Inheritance and classes are closely linked since in most languages only classes (rather than objects) can be inherited from. This has led to the almost general belief that object-oriented = objects + classes + inheritance [Wegner87], i.e. a distinction is made between *object-based languages*, that do not have classes, *class-based languages*, that do have classes but no inheritance and *object-oriented languages* that have both classes and class-based inheritance.

Classless languages generally employ a *delegation* mechanism rather than an *inheritance* mechanism. If inheritance specifies behaviour sharing at the level of classes, then delegation specifies behaviour sharing at the level of objects. Delegation mechanisms may vary in the amount of flexibility that is supported, i.e. with pure delegation the sharing pattern may dynamically vary after an object has been created, in a more restricted form the delegation structure is fixed after an object has been created.

Although some work has been done to harmonise classless delegation and class-based inheritance [Stein,Lieberman&Ungar89] and proposals are made for integrating them [LaLonde,Thomas&Pugh86] [Stein87], both are still considered as fundamentally different language concepts. As we will illustrate further on, at the semantic level the differences between pure delegation and inheritance seem to confirm this feeling. Moreover we will show that neither satisfies our design criteria, and that a new notion of inheritance needs to be developed. We will show how a particular form of object-based inheritance, on the basis of mixin-methods, is the right compromise between class-based inheritance and classless delegation.

Finally note that for people in the classes+inheritance camp, the concept of objects is a very broad concept. All that is needed is that it must be possible to support some notion of inheritance. OOPL are not classified according to the nature of objects, but rather they are primarily classified according to the presence and the nature of classes and inheritance. Given the large variety of kinds of objects, all with sometimes fundamentally different properties (as

will be illustrated below), inheritance is at least a suspect property for a primary classification.

The Polymorphism Camp

Objects are often associated with polymorphism. If we define the protocol of an object as the set of operations that can be applied to that object, then an object-oriented language is polymorphic if each object can be transparently substituted by all other objects that have at least the same protocol. We say that an object can be transparently substituted by another object if the latter object can be used in any program context in which the former object can be used without modification of neither objects, nor the program context. In a polymorphic programming language any object can be filled in a particular program context, if it has a protocol that subsumes the expected protocol. Take for example an operation that expects an object with a particular protocol. According to the polymorphism camp, this operation must be transparently applicable to all objects that implement this protocol.

Note that this form of polymorphism only says something about being able to substitute objects with a comparable protocol, it does not say anything about whether such substitution is *meaningful*. To decide whether a substitution is meaningful the behaviour (i.e. the abstract description of the effect of the applicable operations on an object) of the substituted objects must be taken into account. We will have more to say on this later.

The above form of polymorphism is called inclusion or *late-binding polymorphism*. In contrast with for example parametric polymorphism it is independent of type issues. With parametric polymorphism an operation can be applied to arguments of different types but typically has to rely on case analysis on the actual types of the arguments to perform its action.

Late-binding polymorphism is obtained in different ways. The first and foremost way is by means of message passing. An operation is not applied to an object (as for example in applying a function to some value), but rather an object is asked to perform an operation by sending it a message. It is the responsibility of the object itself to select the corresponding operation to perform. Obviously objects that understand the same messages can be substituted for each other.

A second popular way to achieve late-binding polymorphism is through overloaded functions, as exemplified by languages based on multi-methods. With multi-methods, all operations are overloaded with all the objects to which they are applicable, i.e. with one abstract operation (identified, for example, by its name) different concrete variants of the operation are associated. When an operation is applied to an object the correct variant is automatically selected. An object' s protocol consists of all those operations in which the object is used as a determinant for overloading (things might be a bit more complicated since an operation may be overloaded on different arguments, but we will ignore that for the moment). Again, objects that share the same protocol can be substituted freely.

The Data Abstraction Camp

As already mentioned the idea of an object as a set of operations that share a state is a very popular one. This view on objects is a particular case of the more general idea about objects as data abstractions, i.e. data-representations that are only accessible through a separately defined set of operations. The particular form of data abstraction used in object-oriented programming

languages is also called encapsulation.

The problem here is that many different forms of data abstraction exist. As will be discussed data abstraction, as found in abstract datatype programming languages, and encapsulation, as found in object-oriented languages, are different forms of data abstraction. Furthermore, within the object-oriented paradigm itself, different forms of encapsulation exist. The two most notable are *class-based encapsulation* such as can be found in for example C++ [Ellis&Stroustrup90] and *object-based encapsulation* such as can be found in for example Smalltalk [Goldberg&Robson89]. We find it important to make a clear distinction between them.

The Object-Oriented Typing Camp

Type-systems are an important issue in understanding object-orientation. They are prototypical for what kind of static information can be attached (in case of type checking) or gathered (in case of type inference) to object-oriented programs.

Almost all, if not all, type-systems for object-oriented languages are based on object interfaces (or protocols), i.e. objects are annotated with interface specifications. It is primordial for the typing of object-oriented programs that it can be checked whether an object conforms to its formally declared interface specification. Therefore the interface of an object must be explicit in its definition, i.e. the interface must be formally derivable from an object definition. This must also be true when inheritance is introduced. The inheritance mechanisms must be such that the interfaces of newly created objects can be derived.

Interfaces are not typical for object-oriented programming alone. They play an equally important role in abstract datatypes. Nor are polymorphism and data encapsulation alone typical for object-orientation. It is typical for OOPL that explicit interfaces are intimately connected with late-binding polymorphism and object-based encapsulation to form the intuitive notion of an object as a self contained entity that has a well-defined behaviour and responds to a well-defined set of messages. It is this notion of objects that will be explored.

This section is an overview of the design issues involved while designing an OOPL. This is not an easy task. There have been an abundant amount of proposals for object-oriented languages and language features since the conception of object-orientation.

We will use explicit interfaces as a first yardstick. We feel that although the notions of classes and inheritance are important, the emphasis should be put on objects and on how new objects can be derived from old ones in a fashion that interfaces are derivable also. Up to the present there is a dichotomy between class-based languages that use inheritance (interfaces can be derived) and prototype-based languages that use delegation (interfaces can not be derived) as code reuse mechanisms. We will explore the essential differences between the two and formulate an alternative, object-based inheritance, for which interfaces are derivable and explore the consequences.

Encapsulation and polymorphism will be used as a second yardstick. The distinction between object-based (typical for OOPL) and module-based (typical for abstract datatypes) encapsulation will be elaborated upon. We will explore some variations on encapsulation.

3.2.1 Objects, Interfaces, Messages and Encapsulation

The goal of this section is to clearly delineate our notion of an object. It is the intention to give a more elaborate definition of the intuitive idea of an object as a self contained entity that has a well-defined behaviour and responds to a well-defined set of messages. We will show that the notions of *explicit interfaces*, *object-based encapsulation* and *late-binding polymorphism* are intimately connected. This informal definition of objects will then be used as a yardstick to evaluate possible design decisions in the construction of an object-oriented language. The definition of objects employed in this text is:

> **Object**: Objects can only be operated on by sending messages. An object responds to a finite set of messages. A message consists of a receiver and a selector. Selectors are abstract distinguishable entities. The result of sending a message to an object is again an object. Message passing is an atomic operation: for the sender of a message the result only depends on the combination of the receiver object and the selector, and the way a receiver implements its response to a message is entirely hidden. For each given object the set of messages it responds to is known. Two objects are equal when they respond to the same messages with the same results.

Objects that conform to the above definition will be called *substitutable objects*. In the remainder of this section we illustrate and elaborate on this definition. We will elaborate on the notion of selectors, the atomicity of message passing, the way objects implement their response to a message and how this all goes together with the notion of encapsulation.

The next figure shows an example person object in a graphical notation. Each node corresponds to an object, the messages an object responds to are represented as outgoing arrows, the selector of the message is placed above the arrow, the value is the target of the arrow. In a textual form message expressions are represented as 'o.x' or simply 'ox' or 'o|x' for sending a message with selector 'x' an object 'o'. It should be noted that the graphical notation is an informal notation of a person object.

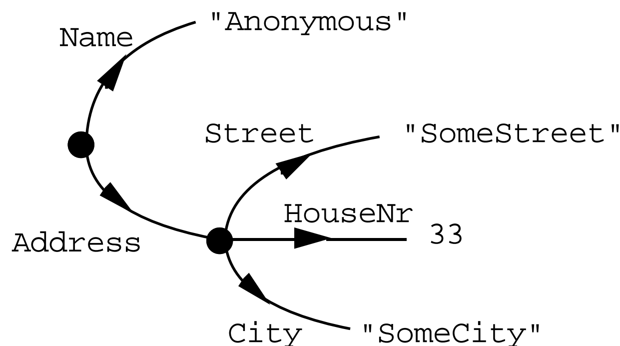


Figure 3.1

Selectors are abstract (syntactic, noncomputed) distinguishable entities. Selectors are not first class values (or objects). Stated in programming language terms the message part (the x in the message 'o.x') of a message expression is not evaluated. Selectors need to be distinguishable since they are used to identify the different messages an object responds to.

The collection of messages an object responds to is traditionally called its *interface*. It is important that objects have an *explicit interface* that is part of

their definition. That is to say, the interface of an object is determined totally by the object's definition and an object should always respond to the same messages in a given context. An object's interface should not be determined by the context in which the object is used.

Two objects are *extensionally equal* if they both respond to the same messages with the same results. The interface of an object, together with a description of how an object responds to the messages that are sent to it, is commonly referred to as an object's *behaviour*. We can restate the above definition as: two objects are extensionally equal if they both have the same behaviour.

Objects can be extensionally defined as a finite collection of named attributes, i.e. mappings of selectors (the name of the attribute) to objects (the value of the attribute). Message passing then corresponds to attribute selection. The result of a message is the object associated with the attribute with the name that corresponds to the selector of the message. Extensional definitions of objects are reminiscent of the more conventional record structures rather than our intuitive idea of objects. It is no surprise that records are extensively used to model certain features of object-oriented programming. Examples can be found in type systems of OOPL and modelling of inheritance in class-based OOPL. We will come back to this in the next section.

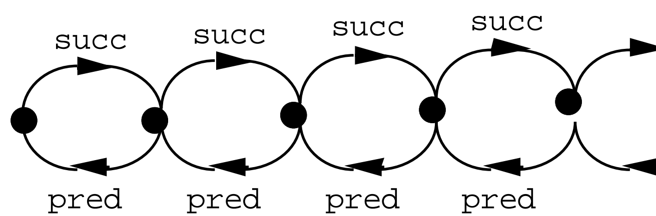


Figure 3.2

An extensional definition of an object is but a mere list of how each name in the objects interface is mapped onto its attribute value (that can be an object definition again, of course). Since extensionally defined objects must directly map each message to a resulting object (without a computation for example), extensional definitions are limited in their ability to express the more interesting object structures. Essentially they are limited in the possibility to define recursive object structures (such as the natural numbers in the above figure).

Intentional definition of objects

An *intentional definition* of an object defines for each name how its value is computed. Attributes can be defined in different ways. An attribute can directly contain a stored value i.e. the attribute is defined by a simple value; or its definition can depend on other attribute values. The former are conventionally referred to as instance variables, the latter are usually referred to as methods.

Message passing is an 'atomic' operation. When a message is sent to an object we say that the corresponding attribute is selected. This means that the corresponding attribute's definition is looked up and evaluated and the result is returned. We will refer to the former as attribute lookup and the latter as attribute evaluation. For example, for a method this corresponds to what is conventionally called *method lookup* and *method invocation*. From the standpoint of the sender of a message method lookup and invocation are unobservable parts of the atomic message, i.e. they form an indivisible whole.

An attribute, either public or private, is selected by name. The result of this can

be, among others, a computed value or a side effect (e.g. in the case of a method) or simply a stored value (e.g. in the case of an instance variable) or a combination of these, all depending on the type of attribute. Each type of attribute can have its own *attribute selection rule*.

Encapsulation
The way and the means to compute an attribute's value, or even the fact that it is a computed value rather than a 'stored' value, remains hidden for the sender of a message. This is what is commonly referred to as encapsulation. We will refer to it as *object-based encapsulation* to make a distinction with other forms of encapsulation. For example, some object-oriented languages employ a form of *class-based encapsulation*. With class-based encapsulation attributes can be declared private (as before), but in contrast with object-based encapsulation, all objects of the same class can invoke or access each others private attributes. Class-based encapsulation is more akin to the more general form of *module-based encapsulation*.

We will call all the objects that send messages to some distinct object, the *instantiating clients* of this distinct object (the terminology might seem bizarre here but will become clear later on). Object-based encapsulation then means that an object is free to use 'private resources' to realise its behaviour, that the instantiating clients have no access to. Conversely this also means that an object can only realise its behaviour by making use of its own attributes and private resources. The private resources available to an object will be called its *private or encapsulated attributes*.

Encapsulation of Acquaintances
Encapsulation is based on how an object can gain access to other objects. In the spirit of Actor languages [Agha86], we will call all the objects that a distinctive object has knowledge of, or can directly refer to, the *acquaintances* of that object. We will call an object the owner of its acquaintances. Two complementary aspects of encapsulation can be identified. The one side has to do with to what extent an instantiating client can gain access to the acquaintance of the object of which it is instantiating client. The other side has to do with how an object can gain new acquaintances.

Object-based encapsulation is the consequence of the fact that an object has total freedom in the way it realises its behaviour. This ensures that the implementation of an object remains hidden for its instantiating clients. This includes the fact that an object can use, in its implementation, other objects or acquaintances that are not directly accessible by instantiating clients.

With object-based encapsulation instantiating clients are given access to an object's acquaintances at the initiative of the owner object. An owner object gives access to one of its acquaintances (whether already in existence or newly created) as a result of message passing, i.e. by returning an acquaintance as result of a message. An owner object that does not grant access to any of its acquaintances can be called an *autistic object*, since no information can be retrieved from this object. An object that grants access to all of its acquaintances is a *nonencapsulated object*. In OOPL where acquaintances are stored in named instance variables, encapsulation is tantamount to restricting (read) access to these instance variables. Notice however that this does not exclude *public instance variables* as long as a distinction can be made between instance variables that are public, and instance variables that are encapsulated.

The complementary sort of encapsulation is based on how an object can gain

references to other objects. Once again the term autistic applies to objects that can not gain any new acquaintances, and nonencapsulated applies to objects that can gain new acquaintances without restriction. The set of acquaintances of an object can be restricted such that it is determined at any time by the initial set of acquaintances (the ones that where available when the object was created) and the acquaintances gained as arguments of message passing.

Obviously, it is not a violation of encapsulation that an object can gain new acquaintances by creating new objects on the one hand and by sending messages to its acquaintances and retrieving the result on the other hand. The other way around, an object must be given an initial set of acquaintances when it is created. The initial set of acquaintances is given by the object that creates. Similarly, new acquaintances are gained during message passing. An instantiating client can explicitly pass a set of acquaintances as arguments of a message.

In contrast with the former sort of encapsulation, this latter form is generally not accounted for in present day OOPL. A simple case where this encapsulation is violated is in OOPL that have global variables. In that case the objects contained in the global variables are acquaintances of all the objects present in the system. It is obvious that the set of acquaintances of a particular object can unrestrictedly change.

Encapsulation of Methods
It is obvious that apart from the ability to encapsulate acquaintances, it should also be possible to encapsulate methods. Again it is important to make a distinction between module-based and object-based encapsulation of methods. Present day OOPL either lack the possibility to encapsulate methods (Smalltalk [Goldberg&Robson89]) or employ a form of module-based encapsulation of methods (Self [Ungar&Smith87], C++ [Ellis&Stroustrup90]). The latter usually takes the form of privacy attributes that are attached to methods. All objects of the same class can invoke each others private methods, objects of different classes can only invoke each others public methods. So called private methods and their usage will be discussed in the section on scoping.

It is possible to employ object-based encapsulation for methods. Questions that must be answered are: how are these methods invoked, can private methods be overridden, what about visibility of encapsulated methods that are declared in an ancestor ? These questions will be answered in the section on scoping.

Languages that provide objects with a uniform access to both state and behaviour are called *slot-based languages* or also *languages that blend state and behaviour* [Ungar&Smith87]. Such languages typically feature uniform access to private and public methods and private and public instance variables; all of which are accessed through message passing.

Encapsulation of Inheritance
Not much has been said about inheritance yet. The reason is that the fact whether an object has been created as an instance of a class and whether this class inherits from another class or whether an object is created as a copy of another object or whether it directly inherits or delegates to another object should be encapsulated in that object. For an instantiating client only the behaviour of an object is important, not how this behaviour is realised. In subsequent sections a second kind of clients will be introduced. Clients that *do* care about how an object' s behaviour is realised. These are the so called *inheriting clients*, i.e. classes or objects that inherit from the class or object they are client of.

Late-binding Polymorphism
Late-binding polymorphism is the result of the fact that the same method name can be used by different objects. Each object can associate a different body (definition) to this method name. When a message is sent the appropriate method body is selected according to the receiver-object. This leads to polymorphic code since objects that implement the same protocol can be intermixed.

Late-binding polymorphism is inherent to the objects as defined above. Since each object has total freedom in the way and the means to compute an attribute's value (encapsulation), in case of methods, each object can associate different method bodies to the same method name. So, in our definition of objects, polymorphism and object-based encapsulation are two sides of the same coin !

3.2.2 Alternative Object Models

In order to further delineate our view on OOPL we will sketch four alternative approaches to object-oriented programming and we will show in what respect they do not correspond to our notion of objects.

Multi-Methods
Multi-methods form the basis for a popular class of object-oriented programming languages including CLOS [Moon89]. Multi-methods depart from the idea that messages are passed to a single distinct receiver. The goal is to construct a more powerful form of message passing where multiple 'objects' can participate in method lookup. Multi-methods were introduced, at first, to integrate object-oriented concepts into a functional language (Lisp), but were also inspired by the observation that message passing to a single distinguished receiver is in some cases awkward. This is the case with e.g. most hybrid binary arithmetic operations.

In a typical language employing multi-methods an object is defined by giving a list of its instance variables. Methods are declared separately. They are defined as (runtime) overloaded functions [Ghelli91]. With each method name different definitions can be associated, each distinguished by the number of formal arguments and a specification for each formal argument on which collection of objects a particular method definition is applicable. So, with each method name different method definitions can be associated. Which exact definition is to be used when a method is invoked depends on the actual arguments.

```
CartesianPoint = Object x,y:Real EndObject;
PolarPoint = Object rho,theta:Real EndObject;

Method sum (p1:CartesianPoint; p2:CartesianPoint)
  ^CartesianPoint(p1.x + p2.x, p1.y + p2.y)

Method sum (p1:PolarPoint; p2:PolarPoint)
  ^PolarPoint(…)

Method sum (p1:CartesianPoint; p2:PolarPoint)
  ^…

Method sum (p1:PolarPoint; p2:CartesianPoint)
  ^…

cPoint:CartesianPoint(1,1) ; pPoint:PolarPoint(2, pi);
sum(cpoint, pPoint)
```

At first sight multi-methods seem to be a generalisation of objects and message passing. If we were to use a function notation for message passing, then the function name would correspond to the message name and the first parameter of the function call to the receiver of the message. Furthermore, the receiver would play a special role since it is used to determine the exact function or method definition to be used. The overloading that goes with multi-methods and the late-binding polymorphism that was discussed in our definition of objects above are seemingly similar notions. Whereas with pure message passing only the receiver determines the exact method to be selected, with multi-methods all parameters can be used in determining the exact method definition that is to be selected. This gives multi-methods a gain in expressiveness. With the gain in expressiveness, however, comes a loss in object-encapsulation.

In contrast with the single receiver approach where each method truly belongs to one object, each method in the multi-method approach belongs to all of its arguments. As such the implementation details of none of the arguments is hidden in a method's definition. As a result programming languages supporting multi-methods typically do not support encapsulation. One notable counter-example to this is Cecil [Chambers92]. The solution, in fact, exists in combining overloading with encapsulation. All arguments in the multi-method that are overloaded (i.e. that play a role in determining the exact method-body that will be invoked) are regarded as non encapsulated for that method; all other arguments are encapsulated. So the encapsulation problems with multi-methods can be amended.

The lack of support for explicit interfaces is a more fundamental difference with the model of objects as put forward in the previous section. Intuitively, objects defined in a language employing multi-methods do not have the flavour of self-containedness typically ascribed to objects. This stems from the fact that in such languages the set of messages an object responds to is determined by the context in which an object is used, rather than by the object's definition.

Consider the point example from above. It is easy to construct an alternative example where two entirely different sets of methods are defined that are both applicable to e.g. polar points. Each set of methods is defined in a different context (e.g. in some local scope or for example in a 'package' or 'module'). Depending on the context in which a certain polar point is used it will respond to two entirely different sets of messages.

Although multi-methods are certainly useful, they do not conform to our notion of object-orientedness. The lack of support for encapsulation has already been noted, but appropriate solutions to provide encapsulated multi-methods have been provided in other work [Chambers92]. Essentially the lack of explicit interfaces, or rather that the interface of an object is not determined by the object's definition but by its surrounding context, seems to us a more profound difference with what is intuitively called object-oriented.

Objects with an Explicit Method Dispatcher

Objects with an explicit *method dispatcher* can be found in languages for concurrent object-oriented programming [Agha86][America87], but also (and again) in attempts to embed object-oriented features into Lisp-like languages [Abelson&Sussman84]. The central idea is that each object has a 'main' part or body. This main part is responsible for 'method dispatching'. Message passing has more or less the conventional form, that is, a message expression is composed of a distinct receiver and a message. The receiving side, however, takes quite a different form.

The programmer of the receiving object is responsible for deciphering the messages that have been sent and for deciding what action to take accordingly. A process that is normally part of the implementation of objects, and thus invisible for the programmer. The part of the receiving object that is responsible for this deciphering is usually called the 'method dispatcher'. For the purpose of method dispatching a message is explicitly constructed out of a message name and arguments. A typical method dispatcher is nothing but a simple procedural implementation of a mapping of a message name to an associated internally declared procedure (for example a simple Pascal-like case statement).

```
(define (CartesianPoint x y)
  (define (sum p1 p2)
      (CartesianPoint (+ x (p2 'x)) (+ y (p2 'y))))
  (lambda msg
    (case (car msg)
       ('(sum) (sum (cadr msg) (caddr msg)))
       ('(x) x)
       ('(y) y))))

(define p1 (CartesianPoint 1 2))
(define p2 (CartesianPoint 3 2))
(p1 'sum p2)
```

Objects can have encapsulated variables and methods. In the example encapsulation is achieved by making use of the specific scope rules of the underlying language.

Objects with an explicit method dispatcher don't have an explicit interface. Since the method dispatcher is algorithmically defined it is not possible, in general, to determine an object's interface on inspection of its definition.

Associated with explicit method dispatchers is the phenomenon of first-class message names (although this is not always the case). In the above example it is apparent that message names are first-class values, they can be passed as arguments for instance. In the example this is essential for the method dispatcher to work since the dispatcher needs to compare message names.

Restrictions can be put on the form of the method dispatchers (such as in the above example where the dispatcher is essentially a case statement) and message expressions. All such restrictions will have the goal of making the accepted message interface more explicit.

It is trivial to notice that for example object-oriented type checking[1] of this sort of objects is impossible. It is, in general, not possible to determine whether a message sent to an object will result in an error or not. In fact it is in general not possible to determine what message is sent to an object. Neither is it possible to determine whether two objects are substitutable.

1 It is possible to type check objects with explicit dispatchers, but type checking will reduce to checking e.g. function domains of the functions that represent the objects, which is obviously not what is intended in this case.

Lambda calculus with records
Lambda calculus extended with a record datatype can be used to model certain features of object-oriented programming. Objects are simply modelled as records. Once again the particular scope rules of lambda calculus can be used to achieve encapsulation. Instance variables are modelled as record fields containing a value, or directly as variables in lambda calculus. Methods are modelled as record fields containing a lambda function. In the example we use the notation {x1.e1 ; … ; xn.en} for a record with field labels x1 to xn and values e1 to en.

```
(define (CartesianPoint x y)
  {x.x ;
   y.y ;
   sum.(lambda (p2) (CartesianPoint (+ x p2.x) (+ y p2.y)))})

(define p1 (CartesianPoint 1 2))
(define p2 (CartesianPoint 3 2))

(p1.sum p2)
```

Message passing, however, is not an atomic operation. The fact whether an attribute of some object is implemented as a stored value versus a computed value can not be hidden. Whereas a message retrieving a stored value is mere record selection, a message that invokes a method must be explicitly constructed as a combination of record selection and function application.

The fact that this construction must be made by the sender of the message also compromises object encapsulation in a way. It is possible to retrieve a method without invoking it. This method can subsequently be stored for later invocation gaining direct access to an object’s encapsulated parts without passing through its interface. It is not so much the fact that this is possible but rather that the means provided to define methods can not preclude this.

Modules and Abstract Datatypes
OOPL share their concern about interfaces and encapsulation with abstract datatypes (ADT). The essential difference is that encapsulation in ADTs is a result of the type system employed. The fact that the implementation of an ADT can only be accessed through its procedural interface is enforced by the type system. This leads to a different sort of encapsulation (i.e. module-based encapsulation).

The essential difference with ADT encapsulation is that a function in an ADT’s interface can access the implementations of all the ADT’s elements it has knowledge of, e.g. a function that is part of an ADT, that has two arguments that are both typed as elements of the ADT, has direct access to the implementation details of both arguments (see the sum operation in the point example below). This leads us to say that with ADTs the implementation details are mutually visible to all members of the ADT. In contrast a method of some object has only access to the encapsulated part of ONE object: the receiver.

Even if a method has indirectly obtained a reference to the receiver object it can not access the receiver’s implementation details via this reference. To put it otherwise: an object has no direct access to the implementation details of any of its acquaintances, even if it has itself as acquaintance. With ADTs a function activation of a function in the ADT has direct access to the implementation details of all the elements of the ADT that are visible in that activation.

```
Module CartesianPointModule
  Interface
    Type CartesianPoint

    Function sum(p1, p2) Result CartesianPoint

  Implementation
    Type
      CartesianPoint = Record x:Real y:Real End

    Function sum(p1, p2) Result CartesianPoint
      c:CartesianPoint
      Begin
        c.x := p1.x + p2.x
        c.y := p1.y + p2.y
        ^c
      End
End
```

Traditionally, ADTs also lack the late-binding polymorphism that is associated with objects. Polymorphism can be added orthogonally to ADTs.

The essential difference between object-based encapsulation and ADT encapsulation is in all aspects similar to the two complementary encapsulation mechanisms of procedural abstraction and type abstraction as identified by Reynolds in [Reynolds75]. In a similar vein the differences between objects and ADTs have been amply discussed in [Cook90]. Although the terminology used (Cook discusses the difference between what he calls abstract datatypes and procedural data abstraction) is different, and we do not share his opinion that the essence of object-orientation is procedural data abstraction, we agree with his argumentation on how ADTs support a different notion of encapsulation and are restricted in their overloading capabilities.

3.2.3 Operations on Objects

In practical object-oriented programming languages objects are defined in different ways. In class-based languages all objects are instantiated from a set of templates — called classes — in a cookie cutter way [Stein,Lieberman&Ungar89]. In prototype-based languages idiosyncratic objects or new objects can be made by copying old objects.

Furthermore different mechanisms exist to construct new objects out of (a combination of) old objects. In general this takes the form of inheritance, which is an incremental modification mechanism on classes, or delegation [Lieberman86], which is a control structure similar to message passing, or a more modular mechanism where two classes or objects are combined with a primitive operator to form new objects [Bracha92][Bracha&Lindstrom92].

The emphasis of this and the next four sections is, on the one hand, on how new objects can be constructed so that interfaces are derivable, and on the other hand, on how we can derive *objects* from already existing *objects* (with the emphasis on *objects*). Unrestricted unanticipated delegation is unacceptable in this context. In the section on prototypes it is shown that unrestricted unanticipated delegation is in contrast with the constraint of explicit interfaces. Pure class-based approaches are unacceptable in their need to have an explicit class construct in order to derive new objects. Class-based inheritance is not an operation on objects.

The purpose is to come up with an acceptable form of object-based inheritance, whereby object-based inheritance is an inheritance mechanism for objects (such as delegation) in which interfaces are derivable. We will first show how classes

can be represented as generator functions. Exactly these generator functions will form the basis of a novel mechanism of object-based inheritance. This mechanism is based on mixin-methods. Note that classes will not be discarded entirely, in the section on prototypes versus classes we will show how classes can be reintroduced in an OOPL that employs object-based inheritance.

The second goal is to show that at least one other operator exists to derive new objects. We will show an operator that derives a new object by encapsulating one object in another. It will be shown that essentially two forms of this encapsulation operator exist. Furthermore an example will be given to show that one of both forms is a possible candidate as a basis for introducing user defined control structures in an OOPL in replacement of the often used higher order functions.

In spite of the considerations that were made in the previous section, objects will be modelled as records. Much of what follows in this and the next four sections is derived from work in denotational semantics of object-oriented programming languages [Cardelli88] [Cook89] [Bracha92] [Hense92] [Ghelli90] [Kamin88] in which records are almost unanimously used to model objects. It is our opinion that the argumentation is, to a sufficient extent, independent of the employed object model.

3.2.4 Classes and Class-based Inheritance

Design Issues in class-based languages
In class-based languages objects are grouped into classes. The class concept is heavily overworked. Classes also play many different roles in different OOPL.

A class is usually defined as a *template* that contains both instance variable definitions and method definitions. This template can be *instantiated* to create new objects, called *instances* of the class or template, that conform to this template. An object conforming to some template has exactly the number and names of instance variables as defined in the template and responds to messages according to the appropriate method definitions from the template. The exact value of the instance variables can vary from object to object.

Class-based inheritance is usually characterised by its *strict use of templates*. That is to say, objects instantiated from some template will stay conform to this template during their entire lifetime. This means, for example, that an instance of some class can not be extended by adding extra methods.

Another characteristic property of class-based languages is the *strict separation of templates and instances*. Concretely this means that the operations allowed on templates are restricted to instantiation and on the other hand instances can not be used as templates. Furthermore, class-based languages require in general that all objects are instance of some class, i.e. no other means are provided to create objects than by class instantiation.

Classes can be organised into class hierarchies. We will see that *class-based inheritance strictly involves incremental modifications of templates*. This view is supported by the fact that objects in a class-based language have *strong identity* (as discussed in the section on object identity).

Classes as a classification mechanism
Objects are classified by classes; classes are classified by class taxonomies. Keeping this in mind, an object belongs to more than one class: the class from which it is instantiated and all the superclasses from that class (or more formal all the classes into which this class is classified). We will call the class from which an object is instantiated the object's class. In traditional class-based languages an object has exactly one class and an object can not be reclassified dynamically. Languages exist, though, where an object can have more than one class [Hamer92] and where objects can be reclassified dynamically. We will have a closer look at the latter case.

Two different sorts of *dynamic reclassification* exist depending on the 'feasibility' of the reclassification. An object is closely linked to its class since a class describes the template ('layout') of all the objects that are instantiated from it (and in fact dynamic reclassification is more or less a violation of the 'strict' use of templates that is typical for class-based languages).

An object can not easily be reclassified to a class that defines a totally different template for its instances. Taking this into consideration, the 'feasible ' reclassifications of an object are those where an object is reclassified to a superclass of the object's class or one of the possible subclasses of the object's class. The 'unfeasible' reclassifications of an object are those where an object is reclassified to a class that is unrelated to the object's class. Although the latter sort of reclassification has been studied in the context of for example object-oriented databases the former sort of reclassification seems more useful as a concept in programming languages.

Reclassification is called monotonic if an object is reclassified to one of the possible subclasses of its class. *Monotonic reclassification* is interesting since it allows an object to gain attributes during its lifetime.

Classes as a module mechanism
Classes are in some cases used to introduce an extra form of encapsulation. Existing languages mostly employ an object-based encapsulation, i.e. the per object encapsulation of private attributes that is inherent to object-based programming. Classes can be used to introduce a sort of *module-based encapsulation* [Ungar,Chambers,Chang&Hölzle91]. The idea is to let all objects belonging to the same class have privileged access to each other. This is reminiscent of, the already discussed, abstract datatype encapsulation.

In its ideal form this sort of module-based encapsulation should be an extra form of encapsulation on top of the already existing object-based encapsulation, this is not always the case however (e.g. C++ exclusively uses module-based encapsulation). Module-based encapsulation is realised by declaring in each class which of the attributes are visible only for the objects belonging to the class or vice versa, which of the attributes are visible for all objects not belonging to the class. In its ideal form this extra restriction on the visibility of attributes only applies to those attributes that are not made invisible by the object-based encapsulation.

Object-based encapsulation alone is in some cases too restrictive. Examples where module-based encapsulation is desirable overlap largely the examples where multi-methods are desirable, e.g. arithmetic operations. Module-based inheritance is indeed useful, but it has been rightfully argued that classes and modules are separate concepts [Szyperski92], i.e. that this sort of encapsulation should not be strictly coupled to classes but rather that modules should be provided as an explicit language construction.

Similar observations to the above can be made concerning the use of 'shared' or global variables. Most class-based languages associate with a class a set of variables that are shared by, and directly visible to, all instances of that class. These so called *class-variables* play the role of global variables and can be used to share information among instances of one and the same class. Apart from these class-variables other kinds of shared variables are provided (in Smalltalk for example there are so called global variables, pool variables and class variables). Here again one can wonder whether the use of shared variables should be strictly related to classes and if not whether it should perhaps be better to provide a separate mechanism to share variables.

The use of *nested classes* is related to both of the above issues. Depending on the language nested classes are used to implement shared variables or to provide a restricted form of a module mechanism. This will be elaborated upon in the section on scoping.

Meta-Classes
Classes themselves can be considered as first-class objects. The advantages are that classes can be manipulated as any other object and that instantiation can be realised by message passing. An object is created by sending an instantiation message to the class of which an instance is required. This has several advantages. First, no special operation is required for instantiation; secondly a class can now provide different instantiation methods. The task of an instantiation method is to create a new instance and to do the necessary initialisation.

In a pure class-based language (one in which every object must have a class) it is impossible to consider a class as a first class object without then considering the meta-class of this class (i.e. the class of which an object's class is an instance) and its meta-class and its meta-class and so on … .

It is apparent that the *class concept is heavily overloaded*. This is also apparent in the relatively large number of proposals to unravel the different functionalities covered by the class concept. Witness of this are proposals to differentiate classes from types (interfaces), classes from modules and to provide sharing mechanisms that are independent from the class construct.

Class-Based Inheritance as an Incremental Modification Mechanism
The idea of class-based inheritance is that a new class is defined by specifying how it differs from an already existing class. This latter process is also referred to as *incremental modification* [Wegner&Zdonik88]. Incremental modification is considered to be one of the most innovative concepts of OOPL. The goal of it is to realise small system changes by small specification changes, or, to make extensions or modifications to a system's behaviour without modifying existing code but rather by adding new code.

Viewed as an incremental modification mechanism inheritance takes the form of a parent P (the superclass) that is transformed with a modifier M to form a result R = P + M (the subclass); the result R can subsequently be used as a parent for further incremental modification.

In our model of objects the parent, result and modifier are collections of named attributes. From the viewpoint of the result R, the attributes defined in the parent P are referred to as the *inherited attributes*, attributes defined in the modifier M are referred to as the *proper attributes* of R.

The result R is truly an extension of the parent P (in contrast with e.g. aggregation). Access to the inherited attributes in R is exactly the same as access to the proper attributes of R, though the proper attributes of R take precedence over the inherited attributes in case of name clashes.

The above incremental modification model of inheritance is a simplification. In most common object-oriented languages, modifiers themselves, also, have access to the attributes of the parent with which they are composed. For example, a subclass can invoke operations defined in the superclass (hereafter called parent operations). To model this, a modifier M is parameterised by a parent P that can be referred to in the definitions of the attributes of M. The actual parent is supplied to a modifier when a modifier is composed with a parent. Composing a parent P and a modifier M now takes the form P Δ M = P + M(P), where the modifier M is no longer a simple set of attributes, but is now a function from a parent to a set of attributes.

Modifiers can be made concrete in lambda calculus. In the following example we assume that collections of attributes are represented as records. We assume that records can be added by means of the "+" operator. As an example a modifier is given that extends a point object with a `sumtwice` method. The point object is assumed to understand a sum message.

```
PointModifier =λsuper.{sumtwice.λp2.super.sum(p2.sum(p2))}
ExtendedPoint = Point Δ PointModifier
```

It is obvious from the previous that inheritance as modelled above is an asymmetric operation. Modifiers are different from objects. An object is combined with a modifier to form a new object. It is not possible to combine two objects directly. It is possible, however, to define an operator that combines two existing modifiers into a new modifier:

```
M1 & M2 = λP. (P Δ M1) Δ M2
```

Further argumentation for the asymmetric treatment of inheritance will be given in the section on multiple inheritance.

Method Overriding, Late Binding of 'self'

The above model of inheritance is still a simplification. No treatment is given of recursion or self-references. The expressiveness and the modelling power that is ascribed to inheritance owes much to the special treatment of recursion in OOPL. Recursion emerges when, in response to some message, the invoked method refers to the receiver object. For this purpose a so called *'self' pseudo-variable* is provided. The self pseudo-variable contains a reference to the receiver object. It can be used in the implementation of an object to define recursive methods, for example.

Recursive method invocations take a special form when the base class that invokes one of its methods recursively is modified with a modifier that overrides the method under consideration. In this case the recursive call will result in an invocation of the *overriding method* definition. This is illustrated in the following picture.

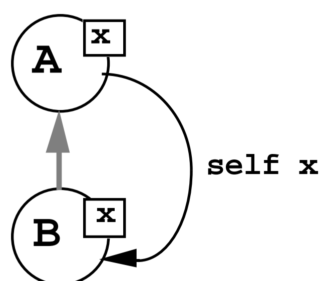


Figure 3.3

The special treatment of recursive invocations in the inherited attributes is referred to as *late binding of self*. The fact that inherited attributes from a parent are more essentially part of inheritors than attributes that are merely invoked can be explained entirely by the late binding of self in recursive invocations in inherited attributes.

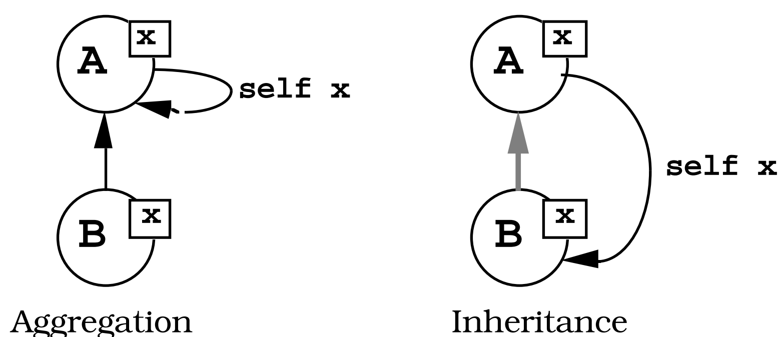


Figure 3.4

Late binding of self is the source of much of the power of inheritance. It is also part of the good programming practice that is associated with inheritance. It allows code to be factored out in a common parent. The parent can rely on its possible inheritors to 'fill in' the unspecified details. A parent that invokes methods that are to be defined in inheritors is called an *abstract parent* [Bracha&Lindstrom92]. Abstract parents are an essential part in structuring inheritance hierarchies and creating *object-oriented frameworks*. We refer to the section on that topic.

Generator Functions and Classes

The most widely accepted definition of inheritance is given in terms of its operational semantics. The operational semantics describes the way a method is looked up in the inheritance chain of a receiver object [Goldberg&Robson89]. This process is referred to as *'method lookup' semantics*. A first denotational semantics in which the importance of recursion is recognised, was given in [Reddy88]. A denotational semantics in terms of fixed points is given in [Cook&Palsberg89][Cook89] and has been used by several other authors [Bracha92] [Hense92].

This analysis is based on the fact that recursive (function) definitions, in a mathematical treatment, can be expressed as fixed points. For example the factorial function can be expressed (in lambda notation) as:

```
fac = FAC(fac)
where FAC = λf.λx.if x=0 then 1 else x*f(x-1)
```

The function FAC of which the factorial function is the fixed point is called the

generator function. The self-reference in the factorial function is explicitly captured by the first argument to this generator function (i.e. ' f' in the above). The fixed point of a *generator function* is constructed by applying a fixed point operator to it. The standard fixed point operator in lambda calculus is the Church or Y fixed point combinator [Revesz88], i.e.:

```
fac = Y(FAC) where Y(f)= f(Y(f))
```

An example of how a point object can be modelled using fixed points is given below. The point object is self referential in its `distFromOrigin` method. The `distFromOrigin` method needs access to the point object itself. This self-reference is resolved by constructing the resulting point object as the fixed point of a generator function that expects a point object as argument. This generator function generates objects in which this self-reference is resolved. The idea is of course that the argument of the generator function is bound to the object that is being generated: a point object is the fixed point of the generator function. The `INST` function, in the example, is the same as the Church or Y fixed point combinator.

```
pgen(a,b) =
  λs. { x.a,
        y.b,
        distFromOrigin.√(s.x² + s.y²),
        closerToOrigin.λp.(s.distFromOrigin < p.distFromOrigin)
      }

p = INST(pgen(1,2))
p.distFromOrig

INST(x) = x(INST(x))                    -- INST(x) = Y(x)
```

Another way to look at generator functions is that they are objects in which the self is not yet bound. In that respect they are comparable with classes. Taking the fixed point of a generator function is the equivalent of instantiation. Hence the name of the fixed point combinator in the example.

Inheritance can be modelled as an operator on generator functions. The fact that in a generator function, the self is not yet bound provides exactly the necessary functionality to model the late binding of self. This is illustrated in the following example. Manhattan points are constructed by overriding the `distFromOrigin` method from point. Generators can be combined to form new generators with the `INHERIT` combinator. The result of the `INHERIT` combinator distributes its self argument (the result is a generator) over its two constituent generators. The result of applying the two constituent generators are two objects that can be combined with mere record combination. Instantiation of a Manhattan point is done by taking the fixed point of the combined generator.

```
mpgen = λs.{distFromOrigin.(s.x + s.y)}

mppgen(a,b) = INHERIT(pgen(a,b), mpgen)

mpp = INST(mppgen(3,4))

INHERIT(G1,G2) = λs.G1(s)+G2(s)
```

The techniques found in this section and the ones found in the previous section can be combined. This gives a full account of inheritance as found in class-based languages.

3.2.5 Classless Delegation

In *prototype-based languages* (also referred to as *delegation-based languages*) objects are not organised into classes. The basic mechanisms provided in prototype-based languages are object creation and delegation, apart from message passing, of course. This results in a much simpler view of object-oriented programming. Still some design issues remain to be discussed. The discussion below follows to a certain degree the discussion of prototype-based languages in [Dony,Malenfant&Cointe92].

Object Creation

In the absence of classes, objects must be created by another means than template instantiation. Two alternatives exist. One is that objects are created ex nihilo; the other alternative is that objects are created by *cloning* (copying) an already existing object.

Objects that are created *ex nihilo* can be either created as empty objects that must be 'filled up' afterwards, or an object is created by listing its public and private attributes (both methods and instance variables) and the initial values for instance variables. For similar reasons as in [Dony,Malenfant&Cointe92] the former option is ruled out. Creation of empty objects presumes the existence of primitives to dynamically change the structure of an object, i.e. to add instance variables and methods. This is not only a dangerous feature but is also in contradiction with the fact that an object's interface should be explicit in its definition.

The second alternative to make new objects is by cloning existing objects. It is apparent that this is a less primitive way to create objects. With a cloning primitive alone no objects with a new structure can be constructed. In fact cloning closely resembles instantiation in class-based languages. Cloning can be interpreted as taking an existing object as template to create a new object. Additionally to the creation of a new instance, which is the purpose of instantiation, this new object has an initial state that is a copy of the state of the object used as template. Different sorts of cloning exist according to the amount of control a programmer has over the copying process with respect to which part of the state of an object has to be copied (e.g. deep copying versus shallow copying).

To sum up: whereas in class-based languages objects with an idiosyncratic structure are created by first defining a template (ex nihilo) and then instantiating this template. And whereas in class-based languages objects of the same kind are created by taking different instances of one and the same template. We can say that in prototype-based languages, on the contrary, idiosyncratic objects can be created directly (ex nihilo) and objects of a same kind are created by cloning an existing object with the extra advantage that state, also, is copied into the newly created object.

Delegation

The second characteristic mechanism for prototype-based languages is delegation. Different forms of delegation (according to the flexibility and anticipation of the delegation structure) exist. After discussing delegation in its purest form, i.e. explicit unanticipated delegation, we will review some variants of delegation.

Delegation allows the behaviour of an object to be defined in terms of the behaviour of another object. In its most general form delegation was introduced [Lieberman86] as a message forwarding mechanism. A message that can not be handled directly by the receiving object is forwarded (or delegated) to another object that responds on behalf of the delegating object.

Delegation is a special form of message passing. A message is delegated to an object. What differentiates delegation from ordinary message passing is the interpretation of recursive method invocations (i.e. the interpretation of the self pseudo variable). As illustrated, the difference between message passing and delegation is similar to the difference between aggregation and inheritance. The difference between inheritance and delegation is that the inheritance structure of an object is fixed, whereas the delegation structure can vary.

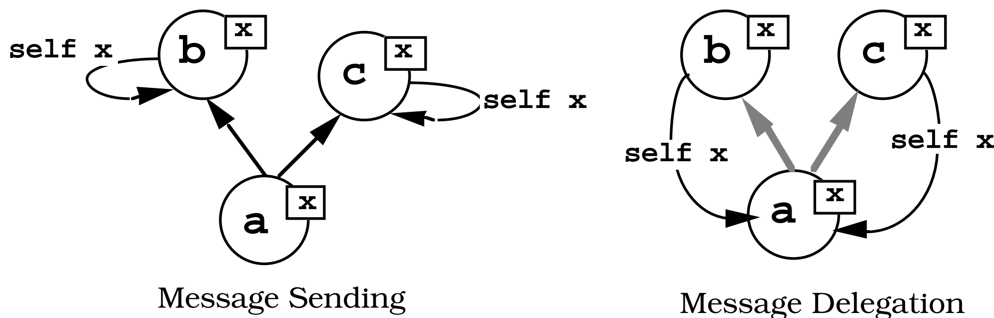


Figure 3.5

In the above model of class-based inheritance objects were modelled as fixed points of generators. An object' s self was bound at instantiation time. In a model of classless delegation all objects must have an unbound self. At any point in time a message can be delegated to an object forcing all self-references in that object to be redirected to the delegating object. An example can be found below.

```
makePoint(a,b) =
  { x.a, y.b,
    distFromOrigin.λself.√(self.x² + self.y²),
    closerToOrigin.
      λself.λa.(self.distFromOrigin(self) < a.distFromOrigin(self)) }

makeManhattan(p)
  = p + { distFromOrigin.λself.self.x + self.y) }

p1 = makePoint(1,2)
mp1 = makeManhattan(p1)
p2 = makePoint(2,2)
mp2 = makeManhattan(p2)

p1.closerToOrigin(p1)(p2)            --- true
mp1.closerToOrigin(mp1)(mp2)         --- true
mp1.closerToOrigin(mp1)(p1)          --- false
```

In this encoding of objects all methods have access to the receiver as an explicit argument. Each time a message is sent the receiver must be passed along explicitly as message argument. This allows considerable freedom. Delegation is characterised by the fact that the receiver that is passed as argument is not the same as the receiver of the delegated message. Consider the following example where ' mp1' delegates a message to ' p2' . ' p2' is the explicit receiver of the delegated message, but all self-references are directed to ' mp1' .

```
p2.distFromOrigin(mp1)               --- √3
```

Variants of Delegation
A first design alternative to be considered is whether delegation must be explicit or implicit. With *explicit delegation* an object can explicitly delegate a message (an operation that syntactically resembles message passing but has a different meaning in its late binding of self) to any other object it has knowledge of (e.g. instance variables). Since an object has to decide on a message per message basis whether this message has to be delegated, explicit delegation is only relevant for objects with an explicit method dispatcher. So, although explicit delegation provides great flexibility it is in contradiction with the notion of explicit interfaces. Therefore implicit delegation is favoured.

With *implicit delegation* an object can explicitly designate another object as its parent. The parent is used to delegate messages to that are not found in the interface of the receiving object. Once again two design choices can be made. One in which the delegation structure can be dynamically changed, the other in which an object can not change parent. In the latter case the parent object must be assigned when the object is created and can not be reassigned during the object' s life-time. We adopt the terminology from [Stein,Lieberman&Ungar89]: the former is called *unanticipated delegation*, the latter is called *anticipated delegation*.

In prototype-based languages where the delegation structure can be dynamically changed the parent of an object is typically stored in some specially identified instance variable (e.g. Self [Ungar&Smith87]). This instance variable can be consulted and also modified, thus changing an objects parent. Dynamically changing an object' s parent is flexible (useful examples are given in [Ungar,Chambers,Chang&Hölzle91]), but again in contradiction with the notion of explicit interfaces.

Implicit anticipated delegation can also be interpreted as an incremental modification mechanism. Here again a parent (the parent object) is incrementally transformed with a modifier M (the definitions in the delegating object) to form a result R = P + M (the resulting object). This insight underlies the argument in [Stein87] in which the author argues that delegation is inheritance. Oversimplifying the main result one could say that the argument boils down to the fact that both delegation and inheritance are both in essence based on an incremental modification mechanism. Whereas in class-based languages inheritance involves incremental modification of stateless templates, implicit anticipated delegation can be considered the same as incremental modification of objects with state.

In the remainder we will call the form of implicit delegation where the delegation structure is anticipated *object-based inheritance* (versus class-based inheritance). This name stems from the fact that this sort of delegation is very similar to inheritance in class-based languages. The same term has been used in [Canning,Cook,Hill&Olthoff89] for a similar notion. Furthermore we will call an object that is being extended the *prototype* (the superclass in class-based terminology) and the extended object will be called the derived object (the subclass in class-based terminology).

3.2.6 Mixin-Method Based Inheritance

The difference between class-based inheritance and delegation can be recast in terms of to what degree the self of an object is encapsulated in that object. With class-based inheritance all objects have a totally *encapsulated self* — an instantiating client of an object can not reassign the self of that object. With delegation-based inheritance all objects have an *unencapsulated self* — all instantiating clients of an object can reassign the self of an object. The dilemma of object-based inheritance is that normal message passing requires objects that have an encapsulated self, and inheritance requires objects that have a nonencapsulated self.

Moreover, generator functions can be seen as a motivation for class-based languages. Since the late binding of self that is essential for inheritance is so intimately connected with generator functions, it could be argued that generator functions and consequently classes are an essential ingredient of object-oriented programming. Generator functions must be first class values to model class-based inheritance with the above approach.

In this section we will show that an alternative approach to inheritance can be devised that combines the advantages of class-based inheritance with the advantages of delegation and avoids the above pitfalls. The flavour of delegation that is so obtained corresponds to the already mentioned object-based inheritance. On the semantic level a new technique is proposed, based on *encapsulated generator functions*, to explain this new kind of inheritance.

Generator functions can be thought of as objects with an unassigned self. The essence of our solution to the above dilemma is that an object encapsulates a nonencapsulated version of itself and that an object must be asked to extend itself; and only an object can extend itself.

An object is, in this approach, a wrapper around its own generator function. This wrapper fixes and encapsulates the self for the wrapped generator function. Since an object has an explicit reference to its own generator function it can extend itself. An instantiating client of this object however has a view of the object in which the self is encapsulated. This is illustrated in the following figure.

```
mpgen = λg.λs.{distFromOrigin.(s.x + s.y)}
pgen(a,b) = λg. λs.
     { x.a,
       y.b,
       distFromOrigin.√(s.x² + s.y²),
       closerToOrigin.λp.(s.distFromOrigin < p.distFromOrigin)
       mp.WRAP(EXTEND(g,mpgen))
     }

p = WRAP(pgen(1,1))
p.distFromOrigin                              -- = √2
mp = p.mp
mp.distFromOrigin                             -- = 2

WRAP(x) = INST(x(x))
EXTEND(G1,G2) = λg.INHERIT(G1(g), G2(g))
```

Generator functions now have two arguments. One argument is used, as before, to contain a reference to the object itself. The other argument contains a reference to the generator itself. Instantiation takes a slightly different form. An object must be provided with its generator function when it is instantiated. This is done in the new instantiation function `WRAP`. Notice also how generator functions are

combined.

Objects can only be extended by selecting an appropriate attribute. In the example the point object is extended to a manhattan point by selecting the `mp` attribute. In anticipation of a thorough discussion on *mixin-based inheritance*, methods (attributes) that return an extension of their receiver object will be called *mixin-methods* (*mixin-attributes*).

The result is a different notion of prototype-based programming. In pure delegation-based languages any object can delegate to any other object it chooses. Object-based inheritance with encapsulated generators still has the notion of extending objects rather than classes. It also conforms to what is generally considered important for prototype-based languages in its nonstrict use of templates. However, it is not possible for an object to delegate messages to any other object it chooses. Each object has control over how it will be extended.

Delegation is more powerful — and, as was shown, too powerful — than object-based inheritance with encapsulated generator functions. Indeed the latter one can be mimicked with the former. It suffices to define a method for each corresponding mixin method that extends the receiver object by delegating to it. Object-based inheritance with encapsulated generator functions is in its turn more powerful than class-based inheritance in its ability to extend objects.

The above new inheritance mechanism puts inheritance in a different perspective. In class-based languages as well as in delegation-based languages, inheritance is realised by explicit operators to combine classes (respectively objects), making inheritance a fundamental operation. In contrast, inheritance as in the above proposal, is an operation that is internal to objects. Witness of this is that all generator functions can be encapsulated in objects.

The above model of inheritance is the basis of mixin-methods in *Agora*. We will not pursue the above line of reasoning in a formal way (the usage of encapsulated generators in a denotational semantics of mixin-methods is currently under investigation). The practical use of mixin-methods will be explored in the section on multiple inheritance.

Can classes and prototypes coexist ?

In the previous sections we showed that both classes and prototypes contribute to object-oriented languages. Still, we argued that each, in combination with respectively inheritance and delegation, has unacceptable drawbacks. An intermediate form — i.e. prototypes with mixin-based inheritance — was opted for. We now turn to the question whether we can reintroduce some of the advantages of classes into a possible OOPL featuring prototypes with mixin-based inheritance.

As argued in other work [Stein,Lieberman&Ungar89] the difference between classes and prototypes is a difference in flexibility. On the one hand classes and class hierarchies are an important structuring mechanism, on the other hand it is sometimes important to have objects with an idiosyncratic behaviour or to be able to temporarily extend an object's behaviour.

There are two notable attempts to integrate classes and prototypes. One attempt is the integration of class-like behaviour in a prototype-based language through the use of 'traits' objects in Self [Ungar,Chambers,Chang&Hölzle91]. The other is the introduction of objects with an idiosyncratic behaviour in a class-based language in a hybrid language [Stein,Lieberman&Ungar89]. Traits objects in Self have been shown [Dony,Malenfant&Cointe92] not to provide the right concept for

this task. The hybrid approach starts from classes as the key concept and handles prototypes as a 'special case' taking away much of the simplicity that is typical for prototype-based languages. It is our opinion that an integrated approach should start from prototypes and consider classes as an extra structuring mechanism on top of the already existing prototypes.

As was shown in the two previous sections the underlying principles of class-based languages are (ex nihilo created) templates, template instantiation and incremental modification of templates; the underlying principles of prototype-based languages are (ex nihilo created) objects, object cloning and incremental modification of objects.

The difference between, and consequently the integration of, classes and prototypes can now best be illustrated by seeing templates as a special case of objects. This view is supported by [Stein,Lieberman&Ungar89], where respectively templates and empathy are considered the fundamental principles underlying both inheritance and delegation. The main difference between classes and prototypes is how strictly templates are used and how strict the separation of templates and instances is.

To introduce classes in a language with only objects we must separate out special objects that will act as templates. In order to be able to regard these special objects as true templates some restrictions must be put on the use of these objects. We will call these special objects *template objects*, all other objects will be called *instance objects*.

First, a strict distinction must be made between those messages that are sent to create a new instance, called *instantiation messages*, and all other messages, called *'ordinary' messages*. In a class-based language the former ones are the instantiation messages defined for classes, in a prototype-based language these are the cloning messages defined on all kinds of objects. So, template objects should be restricted in such a way that they only respond to cloning messages, and vice versa, a strict use of templates implies that all instance objects should be restricted in such a way that instance objects don't respond to cloning messages. Note that since instance objects are cloned versions of some template objects, the above restriction truly is a restriction on the interfaces of both the template object and the derived instance objects. Both objects include the entire protocol (i.e. cloning and all other messages). Declaring an object as a class, however, implies that the protocol of this object is restricted to cloning messages. Making an object as instance of some class implies that the protocol of this object is restricted to all but the cloning messages.

Secondly, a restriction must be put on the incremental modification of instance objects. The incremental modification of template objects corresponds exactly to inheritance. A strict use of templates implies that instance objects can not be incrementally modified. In general it is difficult to enforce such a restriction. However, in the special case of mixin-method based inheritance where an object must be asked to extend itself, this restriction is, again, a restriction on the interface of objects.

Summing up: in order to introduce classes in a prototype-based language some objects must be restricted in the way they are used; we introduce two kinds of these restricted objects, i.e. template objects, that will play the role of classes (and can be called classes from now on!), and instance objects, that will play the role of instances of classes. Template objects can only be cloned (instantiated) and incrementally modified, instance objects can not be cloned nor incrementally modified.

Some important remarks should be made here. Not all objects must be either template objects or instance objects, i.e. it is still possible to have objects with an idiosyncratic behaviour that are not classes. That is to say, classes and prototypes can happily live together. Moreover, we can loosen the restrictions put on instance objects. In most class-based languages it is possible to copy instances (a closely related operation to cloning), it is in most cases not possible to incrementally modify instances. If we allow instance objects to be incrementally modified, then template objects will serve as *minimal templates* [Stein,Lieberman&Ungar89] rather than strict templates. Note that incrementally modifying an instance object is closely related to monotonic reclassification in pure class-based languages.

Finally note that, even when ignoring the above remarks, this approach does not lead to what has been called, in the section on classes, a pure class-based language. Not all objects are instances of a class: classes are not. Classes are objects with an idiosyncratic behaviour and class inheritance is expressed as incremental modification of objects. This is much in the spirit of [Stein87]. We find this a very satisfactory situation. Although meta-classes have some use, meta-classes are conceptually too complex. A symptom of this is the distinction between class-methods and instance-methods in pure class-based languages with meta-classes. The concept of class-methods (and also class instance variables) is mostly considered as concept that is ‘hard’ to understand. By modelling classes as objects with an idiosyncratic behaviour and instantiation as object cloning, this distinction between class-methods and instance methods, is nothing but a restriction on the protocols of the template object (the class) and the instance object (its instances).

3.2.7 Encapsulation as an Explicit Operation on Objects and Generators

Up until now nothing much was said about the form of the encapsulated attributes. In all of the above discussions encapsulation was a consequence of the nested scoping of the underlying lambda calculus and encapsulated variables were encoded as variables in the underlying calculus. Rather than being just a collection of unrelated attributes, the encapsulated attributes can be grouped into an object. Encapsulation then becomes an operation on objects where one object is encapsulated in another object, or alternatively an operation on generator functions. Although unconventional, encapsulation operators in this form are helpful in modelling private methods or nested objects for example. The combination with inheritance gives rise to a variety of different possibilities. This has much to do with the question of encapsulated versus nonencapsulated inheritance as will be discussed in the next chapter. We will not try to be complete in covering all possible forms of encapsulation operators. We will just point out some possibilities.

In its simplest form an encapsulated part is an object, but is defined locally to the generator function. This can be a good way to structure the encapsulated part of an object. For example the encapsulated part can inherit independently from the object of which it is part.

```
pgen =
  λs. let e = INST(λs.{x.0, y.1})
   in {distFromOrigin.√(e.x² + e.y²)
       closerToOrigin.λp.(s.distFromOrigin < p.distFromOrigin)}

p = INST(pgen)
```

The above form of encapsulating attributes fails in modelling private methods as can be found in current day OOPL. What is typical for private methods is that they inherit the ' self' from the object of which they are part, in contrast with the above example where the encapsulated part is an object that has its own self reference. The difference between the two is that in the latter case self-references in a private method are directed to the encapsulated part only, whereas in the former they are directed to the entire object. This can be solved with an encapsulation operator that works on generator functions. The idea is that the generator function of the encapsulated part must be used rather than its instantiated form. The self argument of this generator function can then be bound to whatever is necessary (e.g. it is bound to the entire object when modelling true private methods).

Below is an example of the usage of such an encapsulation operator on generator functions (ENCAPSG). This encapsulation operator encapsulates one object into another such that they share the same self-reference. In this example it is used to model a colour point that inherits from a point class in an encapsulated way. The x and y encapsulated point attributes are not visible to the methods defined in the colour point (encapsulated inheritance will be discussed later on in the text).

```
cpgen = λe.λs.{color.e.color}
pgen = λe.λs.{ distFromOrigin.√(e.x^2 + e.y^2),
               closerToOrigin.λp.(s.distFromOrigin < p.distFromOrigin)}

mppgen = λe1.λe2.INHERIT(ENCAPSG(e1,pgen),ENCAPSG(e2,mpgen))

p = INST(ENCAPSG(λs.{x.0,y.1},pgen))
mpp = INST(mppgen(λs.{x.3,y.4})(λs.{d.5}))

INST(x) =  Y(x)
ENCAPSG(e,x) =  λs.x(e(s))(s)
INHERIT(G1,G2) = λs.G1(s)+G2(s)
```

The encapsulation operator that works on generator functions serves the purpose of structuring the encapsulated part of an object mainly with respect to the inheritance hierarchy. We will come back on this later when discussing mixin-methods and their implementation.

Alternatively an encapsulation operator on objects (after instantiation) can be devised. Consider the following example. After instantiation a point object still expects an encapsulated part. In the example the expected encapsulated part of p is bound to a simple object containing x and y variables.

```
pgen =λs.λe.{ distFromOrigin.√(e.x^2 + e.y^2),
              closerToOrigin.λp.(s.distFromOrigin < p.distFromOrigin)}
egen = λs.{x.0,y.s.x+1}

p = ENCAPS(INST(egen), INST(pgen))

INST(x) =  Y(x)
ENCAPS(e,x) = x(e)
```

It should be noticed that the ENCAPS operator is one that encapsulates attributes into an object after its instantiation. It is an example of an operator that enables an object to gain acquaintances after instantiation. The most important advantage of an explicit operator for encapsulating one object in another, is that it allows to

define control structures in a pure message passing style, thus giving an interesting alternative to higher order functions as a basis for user defined control structures.

Consider the following example. The boolean objects true and false are defined. A conditional expression is constructed by encapsulating an object with two methods that correspond to the branches of the conditional into either the true or false object.

```
truegen =λs.λe.{if.e.true}
falsegen =λs.λe.{if.e.false}
true = INST(truegen)
false = INST(falsegen)

 … λb.λx.λy.ENCAPS({true.x, false.y}, b).if  …

INST(x) =  Y(x)
ENCAPS(e,x) = x(e)
```

Other very similar encapsulation operators can be envisaged depending on their interaction with the different inheritance operators. Especially the interaction with encapsulated generators could prove to be very interesting. Here again, we will not investigate further the different forms of encapsulation operators in a formal setting. Nor will we develop a complete set of operators that explores the different combinations between the more primitive inheritance and encapsulation operators.

We saw two forms of encapsulation operators. One that operates on generator functions and that is mainly used to structure the encapsulated part of an object with respect to inheritance. The other, more interesting one, operates on objects. One object can be encapsulated in an other object. This form of encapsulation offers an object to gain acquaintances after instantiation. It is thus comparable to argument passing. An interesting application of this encapsulation operator is user defined control structures. Both kinds of encapsulation operators will be illustrated in the remainder of this text. The encapsulation operator on objects will be used in the construction of a calculus of objects. An encapsulation operator similar to the one on generators will be used in the construction of a programming language with mixin-based inheritance where mixins can be nested.

3.2.8 Objects with State, State Changes and Object Identity

In most practical object-oriented languages objects have a state that can be changed. State changes are most often realised by having so called instance variables that can be assigned values. An instance variable is associated with an object.

It should be evident from above that, although we find state and state changes important issues in OOPL, we do not find state and state changes as one of the determining features for 'object-orientation'. That is, in our definition of what an object is, an object must not have state to be called an object. We value objects for their abstraction capabilities, more or less comparable to abstract datatypes. Encapsulation does not necessarily mean 'an encapsulated state' (although the notion of encapsulated state is by now part of the 'folklore' of the object-oriented community), but rather that an object can use its own private resources to realise its behaviour. Vice versa an object can also have a 'public state', i.e. variables that are part of the public interface of an object, provided that variables and methods are accessed in a uniform way (e.g. variable access with accessor methods).

Object Identity, Strong versus Weak Identity

An important correlated notion is that of *object identity*. In an OOPL that supports identity each object is assigned a unique identity. This identity is independent of the values of the object's attributes. Identity is kept over state changes and can be used to uniquely refer to an object. Given two objects it can be tested whether they have the same identity. Object identity also is an important issue in defining different copying (object cloning) strategies.

We will call an OOPL where there is a one to one mapping of objects to object identities an OOPL with *strong object identity*. Thus, in a system with strong object identity with each identity corresponds one object, and each object is assigned one identity. Class-based OOPL typically are languages with strong object identity.

The latter constraint is relaxed in OOPLs that allow some form of object-based inheritance (or delegation). With object-based inheritance an existing object — and consequently one that already has an identity — is extended to form a new object. The newly created object is assigned a new identity. Still, the object that is being extended is part of the extended object.

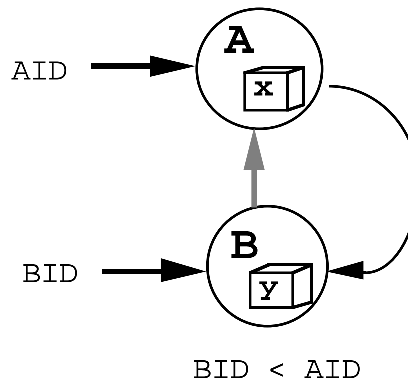


Figure 3.6

On the one hand the original object is a part of the extended object since its interface and implementation are used directly as part of the interface and implementation of the extended object. In this sense it is different than just any acquaintance of the extended object. The extended object can use acquaintances to realise its behaviour, but their interface and implementation does not become a part of the extended object.

On the other hand the original object is a part of the extended object since some of the changes to the extended object are identical to changes to the original object. This is so for changes due to messages sent to the extended object that are implemented in the original object. This phenomenon is normally referred to as inheritance of state. The state of the original object can be accessed through two paths: the identity of the original object and the identity of the extended object.

The above discussion leads us to say that OOPLs with object-based inheritance have objects with *weak identity*. There is a relation between the identity of the original and the extended object. The identity of the original object is a subidentity of the identity of the extended object, i.e. apart from an operator to test whether two objects have equal identity, it makes sense to provide an operator to test whether one object has an identity that is a subidentity of the identity of another object. Similar observations have been made in

[Dony,Malenfant&Cointe92] where prototype-based OOPLs are classified, amongst other criteria, according to the extent to which they support 'split objects'.

Changing Identity
An important question related to object identity is in how far an object can change identity or in how much the object that is associated with some identity can change. The latter can vary from being restricted to state changes only, to having an explicit operator that swaps the identities of two arbitrary objects (such as the become operator in Smalltalk).

It seems natural that due to the importance of an object's interface, identity changes should be restricted to objects with the same interface. This ensures that all objects referenced through one identity have the same interface.

On the other hand in some cases one can observe major changes in the behaviour of an object due to (small) state changes (i.e. a person object becomes a PhD person due to graduation). Practical solutions such as *dynamic reclassification* [Hamer92], unanticipated delegation or more recently predicate classes [Chambers93] have been proposed to cover such phenomena. It should be kept in mind, however, that these mechanisms have a substantial impact on the relation between an object and its identity.

3.3 Object-Oriented Frameworks

Inheritance is a powerful technique for structuring the code in one' s program. It allows common program-fragments to be factored out in superclasses. This ' factoring out' ultimately leads to skeleton classes that define only an abstract implementation. Such an *abstract class* can be reused to build a variety of concrete subclasses. A concrete subclass tailors the abstract class to its specific needs by filling in the methods used, but not implemented in the abstract class. *Object-oriented frameworks* grew out of the practical application of this sort of object-oriented techniques while building computational systems. They grew out of the observation that inheritance and late-binding polymorphism are powerful abstraction mechanisms, and that programs expressed in an object-oriented programming language can be reused by incrementally adapting them to different needs. Among the earliest examples of object-oriented application frameworks was the Smalltalk Model/View/Controller framework [Goldberg&Robson89].

According to [Wirfs-Brock90], an object-oriented framework is a skeleton implementation of an application or application subsystem in a particular problem domain. It is composed of concrete and abstract classes and provides a model of interaction or collaboration among the instances of classes defined by the framework. An important characteristic of a framework is that the methods defined by the user to tailor the framework will often be called from within the framework itself, rather than from the user's application code. The framework often plays the role of the main program in co-ordinating and sequencing application activity. This inversion of control gives the framework the power to serve as extensible skeleton. The methods supplied by the user tailor the generic algorithms defined in the framework for a particular application [Johnson&Foote91].

Frameworks are an emerging technique; there is still much to be learned concerning the design, configuration, and architecture-level description of frameworks [Opdyke92]. They have been investigated mainly in the context of design and code reuse. There is not yet an agreed definition of what exactly is a framework. Furthermore, most frameworks are defined informally; the constraints that are imposed on the components of a framework are informal constraints [Kiczales&Lamping92]. Work on formally expressing constraints can be found in [Holland92][Helm&al90].

Although frameworks are typically explored in the context of reusability, it is our intention to use frameworks as a means to express open systems. It has already been noted that: a framework is a way of representing the theory of how one should solve the problems in a particular area [Johnson90]. Therefore it shares much of the properties of an open system. The relation with open implemented systems has already been noted in [Holland92].

In this section we will explore the notions of abstract classes and object-oriented frameworks. More importantly we will show how a framework can be used to express an open system.

When using frameworks to express open systems two aspects of frameworks need to be emphasised. First of all the distinction between a *framework' s external interface* (corresponding to the object-level interface) and the *frameworks internal interface* (corresponding to the meta-level interface) must be made more explicit. Secondly special attention must be paid to those transformations on the framework that preserve the design of the framework. Particularly *refinement*, (partial) *concretisation*, and extensions to a framework will be discussed.

For the question how frameworks are developed by generalising a set of concrete applications, and vice versa, how they are tested and refined for reusability by building applications that use it, we refer the reader to [Opdyke&Johnson90] [Johnson&Russo91].

3.3.1 Reusability in Object-Oriented Programs

Programs expressed in an object-oriented programming language are, by the very nature of object-orientation, reusable. This does not mean however that every object-oriented program constitutes a full-fledged framework. Before discussing the properties that distinguish a framework from a run of the mill object-oriented program, we will first show in what ways reusability is supported by the object-oriented paradigm.

Late-binding Polymorphism

A polymorphic function or method is more reusable than a monomorphic one. It can be reused with different types of arguments. All arguments of a method in an object-oriented program can be used in a polymorphic way. Every object that conforms to the correct protocol can be used as actual argument. Due to late-binding, a method can rely on the fact that all messages sent to its arguments will invoke the right methods. A method is, as such, reusable for different types of arguments.

It should be noted that *late-binding polymorphism* is a more reusable form of polymorphism than the one that can be found in more conventional languages (e.g. ADA). In the latter polymorphism is more a typing issue, and polymorphic functions are less reusable since they must rely on case analysis of their polymorphic arguments to work. In this respect late binding is more comparable

to the overloading facilities that can be found in such languages.

Moreover, in object-oriented programming languages where bounded polymorphism is used, it is possible to constrain the type of polymorphic arguments. Such a constraint is ideally expressed as a behaviour specification to which all the polymorphic arguments must conform. This gives rise to a subtype relation that is based on the principle of *substitutability* [Wegner&Zdonik88] [Liskov87]:

> ***Principle of Substitutability***: *An instance of a subtype can always be used in any context in which an instance of a supertype was expected.*

In practice, constraints in bounded polymorphism are not formally expressed as behavioural specifications. They are either based on signatures or purely on protocols (i.e. the names of the public attributes of an object). Constraints are then enforced by a type checking algorithm.

Incremental Modification (Inheritance)
Inheritance lets an inheritor reuse the code of its parent in the form of inherited attributes. Consequently all inheritable entities (classes and objects) have two roles and, accordingly, two interfaces. A class, for example, is used for instantiation[2], i.e. it has a protocol for creating instances; and it is used for deriving new (inheriting) classes, i.e. it has an inheritance protocol.

The underlying mechanisms of inheritance have been amply discussed. In practice, inheritance can serve two different purposes. It can be used as a mere code reuse mechanism, i.e. there is no (conceptual) relation between an inheritor and its parent. Or, it can be used as a classification mechanism, in which inheritance is restricted to those cases where a certain relation between parent and inheritor exist. For example inheritance can be restricted such that only *behaviourally compatible inheritance* [Wegner90] is allowed, i.e. the inheritor must be a behaviourally compatible subtype of its parent. With respect to polymorphism, behaviourally compatible inheritance means that an inheritor may be used in any polymorphic context where the parent can be used. Other restrictions to inheritance exist, of which the restriction to *signature compatible inheritance* [Wegner90] is the most common one. In the latter case inheritance can also be used for type checking purposes. The distinction between inheritance as a mere code reuse mechanism and inheritance as a classification mechanism is an important one (especially in relation to polymorphism). We will come back to this later.

Genericity: Generic Classes, Type Substitution, Object Factories
In object-oriented programming languages a whole range of *genericity mechanisms* has been explored. Examples abound: generic classes (Eiffel [Meyer88]), templates (C++ [Ellis&Stroustrup90]), type substitution [Palsberg&Schwartzbach90], object factories [Gamma&al.93] and so on. In general, genericity allows a generic class to be instantiated with a set of classes and types. As we will see, it is an important way to make a class more reusable. Since the concept of genericity is less well-explored as a code reuse mechanism in object-oriented languages than e.g. inheritance (it is considered less germane to object-orientation), we find it important enough to give a brief overview of some of the mechanisms that introduce genericity in object-oriented languages. Especially since we agree with [Palsberg&Schwartzbach90] that, in a special

2 Only concrete classes are considered here, we will have more to say on abstract classes later on.

way, it can be considered as a complementary inheritance mechanism.

A *generic class* is a class that is parameterised (typically with a set of types). The generic parameters can be used in the definition of the generic class. A generic class is instantiated, to an actual class, by providing a set of actual arguments for the generic parameters. Typically generic classes are associated with bounded polymorphism, i.e. the formal types of the generic parameters put bounds on the possible actual generic parameters. Below is an example[3] of a generic list class, and its instantiation to a list of integers.

```
generic class List (NodeType:Object)
  instance variables
    head:NodeType
    tail:List
  methods
    head returns NodeType
      ^head
    tail returns List
      ^tail
endclass

class ListOfInteger is List(Integer)
```

In a statically typed language a generic class can be reused by adapting it to different types. Parametric polymorphism, which is typically associated with generic classes, is generally considered as a form of polymorphism that is orthogonal to late-binding polymorphism [Palsberg&Schwartzbach90]. Also in weakly and dynamically typed languages, generic classes can play an important role for reusability. A class is client of other classes of which it creates instances. A class can be made more reusable by making it generic with respect to the classes of which it creates instances. In the remainder of this text we will see plenty of examples of this. For example, in the implementation of object-oriented languages, an important kind of expressions are expressions that create, so called, slots (i.e. representations of methods and instance variables). These sorts of expressions can be made more reusable by making them generic to the kind of slots that are created. Another good example (example from [Gamma&al.93]) can be found in the usage of user interface toolkits. A toolkit requires different controls such as scroll bars, buttons, or text editors. An application should not hard code dependencies on the look-and-feel: it should be made generic with respect to which toolkit is used.

Generic classes can not be reused in the same flexible way as classes can be reused through inheritance. They require forethought about what is listed in the generic parameters. Furthermore, generic classes differ from 'ordinary' classes. In general it is not possible to inherit from a generic class. Nor is it possible to gradually instantiate a generic class, i.e. it is not possible to further specialise a generic parameter after the generic class has been instantiated. This limits generic classes as a code reuse mechanism.

Alternatives to generic classes exist that are more suitable for code reuse. In [Palsberg&Schwartzbach90] a mechanism called *type substitution* is introduced as an alternative to generic classes. Type substitution is a subclassing mechanism that complements inheritance. It allows all types used in a class to be gradually specialised, without the need of listing these types as generic parameters. As such it avoids the limitations associated with generic classes. In

[3] The above 'pseudo' language is used for those examples and other class descriptions where the description language is irrelevant. This pseudo language is for the larger part self-explanatory, just note that its message passing syntax is derived from Smalltalk's message passing syntax [Goldberg&Robson89].

[Palsberg&Schwartzbach90] it is shown that type substitution and inheritance form an orthogonal basis for code reuse in the definition of classes. Below is the same generic list example of above but with type substitution[4]. It is shown how a type can be substituted after inheriting from the original list class. Furthermore it is shown how types can be gradually specialised.

```
class List
  instance variables
    head:Object
    tail:List
  methods
    head returns Object
      ^head
    tail returns List
      ^tail
endclass

class DoubleLinkedList extends List
  instance variables
    prev:DoubleLinkedList
  methods
    prev returns DoubleLinkedList
      ^prev
endclass

class DLListOfNumber extends DoubleLinkedList[Object<-Number]
class DLListOfInteger extends DLListOfNumber[Number<-Integer]
```

Type substitution was explored in a statically typed language. Obviously only "type consistent" substitutions are allowed. As is the case with generic classes, a mechanism similar to type substitution can also be used in dynamically and untyped languages. In that case this mechanism is used to make classes more reusable with respect to the classes of which instances are created.

[Madsen,Magnusson&Møller-Pedersen90] investigate the use of “virtual classes” (we will use the term *virtual class attributes*, in our opinion this term reflects better the meaning of the concept) as a means to express genericity. Virtual class attributes are a typical language feature of BETA [Kristensen&al.87]. A virtual class attribute is an attribute much in the style of an instance variable, but that contains a class. This attribute can be modified in later subclasses. Below one can find the list example with virtual class attributes.

```
class List
  virtual class attributes
    NodeType:Object
  instance variables
    head:NodeType
    tail:List
  methods
    head returns NodeType
      ^head
    tail returns List
      ^tail
endclass
```

[4] Notice that in the example recursive class definitions are used and that this recursion must be 'extended' in inheritors. We will not deal with this issue in this or later examples. See [Palsberg&Schwartzbach90] for a discussion.

```
class DoubleLinkedList extends List
  instance variables
    prev:DoubleLinkedList
  methods
    prev returns DoubleLinkedList
      ^prev
endclass

class DLListOfInteger extends DoubleLinkedList
  virtual class attributes
    NodeType:Integer
endclass
```

Virtual class attributes share many of the advantages of type substitution over generic classes. They require forethought in the list of virtually declared class attributes however. A class is only generic with respect to the classes that are listed as virtual class attributes. This need not be the case however. In slot-based languages where slots are accessed through message passing (e.g. Self [Ungar&Smith87]), it is generally so that all attributes can be overridden, including instance variables. In that case virtual classes and type substitution are equivalent code reuse mechanisms.

In analogy with abstract methods, and for symmetry reasons that will become apparent in the next section we will use *abstract class attributes* to express genericity. We will see that abstract class attributes are comparable to virtual class attributes for an untyped language. We opted for a different name since virtual class attributes are so much associated with their realisation in BETA, and because it makes the symmetry with abstract methods apparent in the terminology.

Finally, *object factories* [Gamma&al.93] are worth noting as a technique to introduce genericity in classes. Unlike the above three techniques object factories can be used in almost any object-oriented programming language. It consists of a set of conventions that one imposes on one's self. It grew out of a practical concern that hard coding the names of classes of which one creates instances drastically reduces the reusability of one's code. Rather than directly referring to a class name to create a new instance, one refers to a, so called, object factory to create a new instance. It is possible to have different object factories in a single program. Each factory groups related classes. It is possible to override the creation methods that are defined on the factory objects in order to "install" new classes. Object factories are especially useful when the classes of which instances must be created change dynamically. This is not well-covered by the above techniques.

Encapsulation

The implementation of an object's behaviour is not visible to the clients of that object. Due to this all objects that implement the same behaviour can be reused transparent of their implementation.

As will amply be discussed in the sections on multiple inheritance, encapsulation of inheritance, also, is important for reuse potential. In all cases where the inheritance structure of an object is exposed, and where users of that object depend on this, reuse is seriously hampered.

In case of statically typed programming languages this is an important issue. Typing should only be based on the behaviour of objects, not on their implementation. In the case of inheritance this means that subtyping and inheritance are to be considered as different mechanisms, a generally acknowledged fact in the object-oriented community [Shan&al.93]. All languages that confuse subtyping and inheritance have less code reuse potential.

3.3.2 Reusability in Object-Oriented Frameworks

Not every object-oriented program is a framework, regardless of the fact that object-oriented programs are “de facto” reusable. What distinguishes a framework from a merely reusable application is that the major design issues of the application are made explicit. The kind of reuse that is made possible in a program without further additions can best be typified by code reuse, i.e. no relation exists between the program that is reused and the program that reuses. What we intended was *design reuse*, i.e. the program that reuses must be able to know and respect all the major design issues of the program that is reused.

In a framework the major design issues are made explicit by means of *abstract classes*. Abstract classes form the skeleton of an object-oriented framework. Therefore they will be discussed next.

Furthermore the framework must be reused in such a way that the major design choices are respected. That is to say, even when we know what the major design issues are, it still is possible to (code) reuse the framework in such a way that these design issues are violated. This is discussed in the section on operations on abstract classes.

3.3.3 Abstract Classes

In general terms, an abstract class is a class that is only partially implemented. Before making use of the abstract class it must be made concrete by “filling in” the missing details in the implementation. Conventionally only classes with abstract methods are called abstract classes. We will extend the notion of an abstract class to classes that have an abstract acquaintance.

Abstract Methods
The first kind of abstract class that is considered is a class that implements one set of methods, called the *template methods* [Johnson&Russo91], in terms of another set of unimplemented methods, called the *abstract methods* (*or virtual methods*) . Abstract methods are in most cases public methods, but can be private also. Instances of classes with abstract methods can not be used, since their implementation is incomplete. The different kinds of methods can be defined as below.

> ***An abstract method*** *is a method that has no implementation, and is formally declared as such.*
>
> ***A template method*** *is a method that has an implementation but that calls either directly or indirectly an abstract method. Thus, a method that calls another template method is itself a template method, since it will indirectly call an abstract method.*
>
> ***A concrete method*** *is a method that has an implementation and that does not rely on abstract or template methods.*

An abstract method can be made concrete in a subclass by overriding it with a concrete method. The template methods are reused through inheritance. The question under what conditions template methods can be overridden will be discussed later. Abstract methods are a design issue. Programming language support for abstract methods is not available in all object-oriented programming languages. Although concretisation is generally supported (through inheritance and overriding), no support is generally available for indicating that a particular method is a template method or an abstract method. Let alone, that

restrictions on the rules for overriding template and abstract methods are enforced.

An abstract class that uses abstract methods provides design information by reifying the algorithmic decomposition of template methods.

Abstract Acquaintances
The second kind of abstract classes are classes that use other unknown classes in their implementation. In particular, classes that create instances and that have not yet decided which concrete class these instances must be instantiated from. These are called classes with an *abstract acquaintance* (where the abstract acquaintance is the referenced class). Again, instances of classes with an abstract acquaintance can not be used, since their implementation is incomplete.

> ***An abstract class attribute*** *is an attribute that can be used in the implementation to refer to a class, but contains no reference to a class and must be overridden in a subclass to do so.*
>
> ***A template method*** *is a template method in the sense of the previous definition or is a method that refers either directly or indirectly to an abstract class attribute to create instances from.*

A class with abstract acquaintances can be made concrete, by substituting all abstract acquaintances that are used in the implementation, by concrete classes. Here again, few object-oriented programming languages provide support for this kind of concretisation. Language support varies from the above discussed generic classes [Meyer88] or type substitution [Palsberg&Schwartzbach90] to ad hoc solutions such as factory objects [Gamma&al.93]. Here we will study abstract class attributes as a means to express abstract acquaintances.

Abstract class attributes are dual to abstract methods. The implementation of a class is expressed in terms of a set of class attributes that are declared abstract. In the same way that there is no implementation associated with an abstract method, there is no concrete class associated with an abstract class attribute. An abstract class attribute can be made concrete in a subclass by overriding it with a concrete class.

An abstract class that uses abstract acquaintances, provides design information by reifying the aggregation structure of instances of this abstract class.

3.3.4 Operations on Abstract Classes

The external interface of an abstract class is, obviously, its list of publicly visible methods. The internal interface is directed towards inheritors of the abstract class [Deutsch87]. It specifies which, and the constraints under which method and class attributes can be overridden. In this section we will look at this in greater detail. We will discuss how an abstract class can be reused such that the design of the abstract class is "respected".

The idea is to see what the effect is of, and what constraints should be in effect when overriding abstract class attributes with concrete class attributes and abstract, template and concrete methods by abstract, template and concrete methods. First we will restate our extended definition of an abstract class.

***An abstract class** is a class that contains at least one abstract method or an abstract class attribute. It can not be instantiated since its implementation is not complete. A class that is not abstract is a concrete class.*

In the following example, methods x, and y are abstract, methods a, b, and c are template methods and method f is a concrete method. P is an abstract class attribute.

```
class Example
  abstract class attributes
    P
  methods
    abstract x:anArgument
    abstract y
    template a
      self x:3
    template b
      self a
    template c
      ^P x:4 y:8
    concrete f
      ^3
endclass
```

Concretisation of the Abstract Class

An abstract class can be made more concrete by overriding its abstract methods or its abstract class attributes. An abstract method can be overridden with either a template method or a concrete method. All other methods in the abstract class are inherited.

A concretisation of an abstract class can be either a concrete class, or, again, an abstract class. In this latter case the concretisation is partial. A concretisation can be partial due to the fact that not all abstract (methods and class) attributes are overridden, or that an abstract method is overridden with a template method. In the case where an abstract method is overridden with a template method, this template method can invoke already existing abstract methods (class attributes) or newly introduced abstract methods (class attributes). This latter will give rise to layered abstract classes, and in a larger context to layered frameworks, as in the following example. The abstract method “y” is overridden with a template method. This template method invokes a newly added abstract method “v”.

```
class SubExample extends Example
  methods
    abstract v
    template y
      self v
endclass
```

In absence of type information, not all concretisations give desirable results. It is possible to observe “message not understood” errors as could be the case in the following obvious example where the abstract method x is made concrete. Although it is possible to observe from the template method 'a', that in at least one case the method 'x' is given an integer argument, in the concretisation of 'x' this argument is sent a 'push' message (a message that is probably not defined for integers).

```
class ErrorExample extends Example
  methods
    concrete x:anArgument
      anArgument push:4
endclass
```

In order to avoid e.g. message not understood errors, it needs to be specified more formally what possible concretisations are allowed for a certain abstract attribute. Commonly, for this purpose, the arguments of the abstract methods are typed and a type constraint for the abstract class attributes can be given. In the ideal situation, the constraints to what the concretisations of an abstract attribute should conform should be formally specified (see e.g. [Helm&al90] [Holland92]). For an abstract class attribute this constraint can be expressed as a behaviour specification, or constraints on the behaviour of the instances of the possible concretisations of the abstract class attribute.

In any case, the following terminology can be introduced (referred to as the *substitutability rule* for concretisation later on in the text). A concretisation is *substitutable* for an abstract attribute if it produces no (runtime) errors in the case no formal constraints are specified for the abstract attribute or it conforms to the constraints that are specified for the abstract attribute.

Refinement of the Abstract Class

An abstract class can be refined in different ways. First of all, if the abstract class contains concrete methods, then the abstract class can be refined by overriding existing concrete methods with new concrete methods. Secondly, and more importantly, an abstract class that contains template methods, can be refined by overriding a template method with a new template method or a concrete method. In both cases all other existing template, abstract, and concrete methods are inherited from the abstract class. A refinement of an abstract class is, again, an abstract class (the amount of abstract method does not decrease by refinement). It is especially important to notice the difference between a refinement of an abstract class and a partial concretisation of an abstract class. Although both result in an abstract class they do so in a different way.

Not all refinements respect the design of the abstract class. An example of such a refinement is found below. A template method encodes, for its inheritors, design information about the decomposition of an algorithm. The refinement of class `Example` below does not respect the decomposition of the “a” template method.

```
class BadRefinement extends Example
  methods
    concrete a
      self c
endclass
```

Whether a refinement respects the design of an abstract class can be stated more formally as follows:

> ***Refinement Constraint for Abstract Classes****: A refinement of an abstract class respects the design of that class, if each concretisation of the refinement of the abstract class is substitutable for the same concretisation of the abstract class.*

In other words, when making an abstract class concrete, it is transparent whether one makes the abstract class concrete or whether a refinement that respects the design of this abstract class is made concrete.

Extension of the Abstract Class
An abstract class can be extended by adding new abstract attributes or, template, or concrete methods in its inheritors. It is possible to distinguish two sorts of extensions: extensions that depend on abstract attributes or template methods of the abstract class, and extensions that are independent of the abstract attributes and template methods of the abstract class. We will concentrate on the former kind of extensions.

```
class ExtendedExample extends Example
  methods
    template q
      self x:"a"
endclass
```

In this case we must see to it that an extension that refers to an abstract attribute does not expect more from this attribute than the original abstract class did.

> ***Interoperability Constraint for Extension of Abstract Classes:*** *Insofar that a concretisation is substitutable for a particular abstract attribute in the abstract class, this concretisation must be substitutable for this abstract attribute in the extension of the abstract class.*

Stated otherwise, an extension of an abstract class must not presume, in its reference to abstract attributes, any constraints that are not explicit in the abstract class.

Abstraction of the Abstract Class
An abstract class can be made more abstract by e.g. overriding a concrete or template method with an abstract method. This kind of operations has the intention of generalising the framework rather than making it more concrete. This sort of transformation of a framework lies more in the realm of refactoring [Opdyke92] or iterating over the design of the framework with the intention to broaden its applicability. This is beyond the scope of this text.

3.3.5 Role of Abstract Classes in Frameworks

In the previous section we discussed the operations and constraints on these operations that allow us to reuse the design of an abstract class. Without this sort of constraints abstract classes can only be reused in the sense of code reuse.

An abstract class is the design for a single object. A framework is the design of a set of objects that collaborate to carry out a set of responsibilities. Such a set of objects is called an *ensemble* [Johnson&Foote88]. We now turn to the question of how all of the above can be extended to full-fledged frameworks, and what the role of abstract classes is therein.

Forming Ensembles
In the previous section we saw that not all concretisations are allowed for a certain abstract attribute. Two possible ways to specify constraints were pointed out, i.e. typing abstract methods, and class attributes, or, in the particular case of an abstract class attribute, giving a true behaviour specification. In this section we consider a third alternative: that of using a concrete or abstract class (concrete or template method) as a constraint for an abstract class attribute (abstract method).

Based on the *substitutability rule* of before, a (concrete) class specifies an entire set of behaviours: it specifies all behaviours of objects that are substitutable for

its instances. So, a concrete class can be used to constrain the possible concretisations of an abstract class attribute. Only those concretisations are allowed of which the instances are substitutable for the instances of the constraining concrete class.

Similarly an abstract class can be used as a concretisation constraint for an abstract class attribute. Only those concretisations are allowed that are refinements, concretisations, or extensions of the constraining abstract class. Consider our abstract class example from before. The abstract class attribute P can be constrained to being a derivation of some abstract point class, forming a contract between the users of the abstract class attributes (only the external point interface may be used), and the specialisers of the abstract class attribute (only specialisations of the abstract point class may be used to make P concrete). This is how in practice frameworks are created and documented.

```
class Example
  abstract class attributes
    P:AbstractPoint
  methods
    abstract x:anArgument
    abstract y
    template a
      self x:3
    template b
      self a
    template c
      ^P x:4 y:8
    concrete f
      ^3
endclass
```

Two notes must be made here. The first note is that one could be tempted to conclude that if all abstract class (method) attributes are constrained by abstract or concrete classes (template or concrete methods), necessarily ending at the leaf nodes with concrete classes, then we end up with a plain concrete class. This is not so. The difference is that now the contracts between the major parts of the implementation of the concrete class are made explicit (for a good discussion on this in the specific case of methods see [Lamping93]). And moreover they may not be violated ! This is what we wanted. Both a plain concrete class and the abstract class that is made concrete through its concretisation constraints define an entire design space of (substitutional) behaviours. The difference between the two is that in the latter, the design of the class is made explicit.

The second note is about how ensembles are created. Consider an abstract class with a set of abstract class attributes (constrained by abstract classes), and a set of template methods that create instances of and use these abstract class attributes. Such a configuration specifies an ensemble. The template methods specify how the objects in the ensemble work together.

3.3.6 Frameworks, Conclusion

Although much of the facilities that come with object-oriented programming help in expressing reusable systems, not every reusable object-oriented program forms a framework. In order to study frameworks in the context of building systems with an open design two aspect of frameworks were emphasised. The first is the emphasis on the distinction between the external interface of the framework, and the internal interface of the framework. These interfaces are inherited from the abstract classes that constitute the framework. Secondly emphasis was put on the constraints that surround the derivation of systems or

new frameworks from an existing framework. These constraints guard against using the internal interface of the framework without respect for the major design issues that were made explicit. For concretisation of the framework in particular, we saw that the constraints on abstract class attributes can go beyond mere interface or protocol specifications. In a practical setting we can rely on concrete, or again abstract classes, and the substitutability principle to define more fine-grained constraints on alternative concretisations of abstract class attributes. An abstract class that makes use of a collection of abstract class attributes in its template methods, specifies an ensemble.

3.4 A Simple Object-based Programming Language

The programming language that will be presented here is a very elementary object-based programming language that still carries the essential object-oriented features in it. It is directly motivated by the analysis of object-oriented features in the previous section.

The thus obtained language is so simple that its semantics can be given in the form of a calculus. In this section we will explain this calculus by giving the rewrite rules and examples. In the following sections we will interpret this calculus as a programming language, by giving an implementation in the form of an evaluator for it. Essentially the calculus and the programming language differ in the order of evaluation of subexpressions. In the former evaluation order is arbitrary, in the latter evaluation order is fixed. To emphasise the difference between the two the calculus will be referred to as *OPUS* (Object-based Programming calculUS), the programming language will be referred to as *Simple*.

The calculus that will be presented here is still under development [Steyaert92] [Mens,Mens&Steyaert94]. Still, for our purpose here, it has reached a sufficient level of maturity. Only a part of OPUS will be explained, for issues such as recursion, inheritance, classes, etc. the reader is referred to the above references. A rough understanding of OPUS suffices so that the basic concepts of implementational structures of object-based programming languages can be explained.

3.4.1 A Calculus for Object-based Programming

OPUS is designed as an elementary calculus to express object-orientation. It models, in a direct way, the crucial features of object-based programming, i.e. objects, encapsulation and, message passing. OPUS has strongly encapsulated objects, unary methods, instance variables and unary message passing. The objects in the calculus can not change state. Inheritance is not included. No provisions are made for recursion (e.g. “self” sends), although it can be shown how recursive objects can be constructed (but we will not do so here, see [Steyaert92] [Mens,Mens&Steyaert94]).

First of all, OPUS explicitly uses names for message passing. As already discussed in the literature, the use of names greatly simplifies the modelling of object-oriented systems. Examples of this are: the lambda calculus augmented with records (cf. [Cardelli88], [Cardelli&Mitchell89]), Milner' s π-calculus for describing concurrent computations (cf. [Milner91]) and the lambda-calculus

augmented with names, combinations and alternations, as presented in [Dami93a], which is shown to be more expressive than the lambda-calculus with records.

Secondly it has an explicit encapsulation operator. An important aspect of the calculus is that both the public part and the encapsulated part of an object are objects in their own right; unbound variables in public methods are seen as a form of private method invocation. The public part of an object can contain instance variables (public instance variables); the encapsulated part can contain methods (private methods).

We first proceed with a brief introduction to the calculus. The purpose is to get an idea of the meaning of expressions in this calculus.

Concrete Grammar
An OPUS-expression is *well-formed* if it is an element of the language generated by the following context free grammar.

OPUS concrete grammar:

```
Non-Terminal labels    = { Expression Application Abstraction
                           BaseObject CompoundObject Association
                           MethodAssociation VariableAssociation Literal }
Terminal labels        = { Name }
Start label            = Expression

Expression            -> Application | Abstraction | Name | Literal
Application           -> "(" Expression ")" Name
Abstraction           -> BaseObject | CompoundObject
CompoundObject        -> "<" Expression "," Expression ">"
BaseObject            -> "[" [ Association {";" Association}] "]"
Association           -> MethodAssociation | VariableAssociation
VariableAssociation   -> Name "." Expression
MethodAssociation     -> Name "#" Expression
Literal               -> "0" | "1" | … | "9"
```

Objects and Message Passing in the Calculus
OPUS' s base objects and message passing are comparable to records and record selection. They behave as follows:

object	message expressions	result
[]	([])x	normal form
[x.1; y.2; z.3]	([x.1; y.2; z.3])y	-> 2
[x.a; y.b]	([x.a; y.b])y	-> b

Base objects of this kind are constructed entirely of 'instance variable associations'. Free variables are bound via lexical scoping (as is hinted at in the last example). The rule for message passing to such base objects is:

Rule 1a: Instance variable selection in a base object (lexical scoping)

$$([\ldots ; x_i.e_i ; \ldots]) x_i \rightarrow e_i, \text{ if } \forall j<i: x_j \neq x_i$$

In order to model an object we introduce two extra concepts: 'method associations' and 'compound objects'. A compound object is the composition of a public part (mostly a base object with method associations) and an encapsulated part (mostly a base object with instance variable associations); we use the following notation: *<Public,Private>*.

The selection of instance variable associations in compound objects is a straightforward extension of Rule 1a.

Rule 1b: Instance variable selection in a compound object (lexical scoping)
`(< [ … ; x`$_i$`.e`$_i$` ; … ] , d > ) x`$_i$ $\rightarrow$ `e`$_i$`,  if ∀ j<i:x`$_j$`≠x`$_i$

Method associations (x#e) differ from instance variable associations (x.e) in their scoping: the free variables of a method association in the public part of a compound object, are bound via the encapsulated part of that compound object. An example:

```
(<[x#a; y#b], [a.3]>)x
-> {[a.3]}a                 evaluating the body of method x (i.e. "a")
                            in the encapsulated part (i.e. [a.3])
= ([a.3])a
-> 3
```

We will use the notation {c}d to denote the evaluation of an expression d in a context c. The rules for evaluating an expression in a context will be explained below. Given this notation we can now express the selection of methods.

Rule 2a: Method selection in a base object
`( [ … ; x`$_i$`#e`$_i$` ; … ] ) x`$_i$ $\rightarrow$ `{[]}e`$_i$`,  if ∀ j<i:x`$_j$`≠x`$_i$

Rule 2b: Method selection in a compound object
`( < [ … ; x`$_i$`#e`$_i$` ; … ], d > ) x`$_i$ $\rightarrow$ `{d}e`$_i$`,  if ∀ j<i:x`$_j$`≠x`$_i$

In these rules we use a notation {c}d that is not formally a part of the calculus, i.e. it is not a well-formed expression. It must be considered as some kind of meta-definition for the *evaluation of an expression d in a context c*. Formally, the definition of {c}d can be given inductively as follows:

If d is a literal	**then** {c}d = d
If d is a name	**then** {c}d = (c)d
If d is an application (e) x	**then** {c}d = ({c}e) x
If d is a compound object <e , f>	**then** {c}d = <{c}e, {c}f>
If d is a base object [...x$_i$.e$_i$;...y$_i$#f$_i$;...]	**then** {c}d = [...x$_i$.{c}e$_i$;...y$_i$#f$_i$;...]

Remark that evaluating a base object in a context c yields a new base object where *only the instance variables are evaluated in the context*, while the methods remain unaltered. In principle evaluating an expression in some context distributes over all subexpression, except for method associations (method associations are evaluated in the encapsulated part of the object to which they belong, according to rule 2). Evaluating an unbound variable (a name) in a context is, simply, sending that name to the context. Note that the context, in which an expression is evaluated, can be an arbitrary object.

Two examples will illustrate these rules.

```
(<[x#[k.a; j.b]], [a.3; b.4]>)x
  -> {[a.3; b.4]}[k.a; j.b]                                  (2b)
  =  [k.([a.3; b.4])a; j.([a.3; b.4])b]
  -> [k.([a.3; b.4])a; j.4]                                  (1a)
  -> [k.3; j.4]                                              (1a)

(<[x#<[m#a], [a.b]>], [a.3; b.4]>)x
  -> {[a.3; b.4]}<[m#a], [a.b]>                              (2b)
  =  <[m#a], [a.([a.3; b.4])b]>
  -> <[m#a], [a.4]>                                          (1a)
```

These examples show how objects can be returned as a result of "method invocation", and the effect on the binding of free variables. We can also pass objects (as "arguments") to methods; as an example of this we show the construction of boolean values. The method 'foo' expects as input a condition:

```
[if#true]                                                true object
[if#false]                                              false object

[foo#(<condition,[true.1;false.2]>)if]               object with one
                                                         method: foo

(<[foo#(<condition,[true.1;false.2]>)if], [condition.[if#true]]>)foo
-> {[condition.[if#true]} (<condition, [true.1;false.2]>)if        (2b)
=  (<[if#true],[true.1;false.2]>)if
-> {[true.1;false.2]}true                                          (2b)
=  1
```

For modelling objects the most important advantage of the calculus over lambda-calculus with records is that both parts of a compound object can be, again, compound objects. This is essential on the one hand to model private methods, and on the other hand to have some form of curried binding of instance variables. The key insight is that an unbound variable in a selected method is seen as a message with an implicit receiver: the encapsulated part of the compound object of which the method was a part. The interaction between the encapsulated part and the public part of a compound object, is that a selected method from the public part sends its unbound variables to the encapsulated part and conversely that the encapsulated part is an object that contains methods for the unbound variables in a selected method from the public part.

Although in the previous rules and examples it was shown that the private part of a compound object could be a compound object again, the public part was always a base object. So, we need to consider the case were the public part also is a compound object. This will be called curried binding of private attributes. The rule for curried binding is simply as follows:

Rule 3: currying
<<e, f>, g> → <e, <f, g>>

This rule can be shown to be consistent with the previous rules. It is essential for modelling objects to which the private attributes are bound in different stages. To give an example the foo method from the boolean example above can also be constructed as follows:

```
<[foo#(<condition,[true.1;false.2]>)if], [condition#condition]>
```

Sending the message foo, after binding an actual condition to the receiver object, as before, will give the same result. Not only does this make more clear what the

expected instance variables (or arguments) are for the foo method, but also, now, the object to which the foo method belongs can have its own private attributes (see below). Argument passing could be modelled this way: arguments (the condition) are bound to an object in supplement to the already bound instance variables (the variable a).

```
<[foo#(<condition,[true.a;false.2]>)if], [a.3; condition#condition]>
```

3.5 Definition of the Framework

The above calculus defines an — albeit operational — semantics for our Simple programming language. Before proceeding with an implementation of Simple, we will first give a initial approximation of a framework for object-based programming languages. This initial approximation will be used to express an evaluator for Simple. More importantly, we will illustrate its shortcomings and, accordingly, improve the framework.

3.5.1 Representation of Programs and Compositionality

The representation of programs will be based on *abstract grammars*. For the syntactic aspects of programming languages there is a well-developed theory of formal languages that supports both the description of the syntax of programming languages and the representation of programs in programming languages. The theory of context free grammars [Chomsky56] provides the basis for most, if not all, aspects on defining the concrete syntax of programming languages. These grammars are used in most programming language implementations to “parse” the textual representation of a program into an internal program representation. This program representation is then used by the other components of the programming environment.

For the (internal) representation of programs there is a well developed theory of *abstract syntax trees* (AST’s) [De Hondt93][Madsen&Nørgaard88] that originated in the Mentor project [Donzeau-Gouge&al.80]. Abstract syntax trees are also used in the definition of the semantics of programming languages [Schmidt86]. Whereas the concrete syntax concentrates on “parsability” issues such as precedence rules, and the exact textual representation of the terminal symbols in the grammar, the abstract syntax concentrates on the hierarchical structure of the grammar, i.e. on how compound expressions are composed of subexpressions.

Different forms of abstract grammars exist imposing more or less structure on the form of the grammar definition. We take the simplest form. Below is a definition of an abstract grammar for Simple. It is an abstract grammar that corresponds to the already given concrete grammar for OPUS (OPUS and Simple share the same concrete grammar). It takes the form of a set NT of non terminal nodes, a set T of terminal (or leaf) nodes, a collection of expansion sets, a set of production rules, and a root set.

The nonterminal nodes account for *composite expressions*, i.e. expressions that are compositions of subexpressions. For each composite expression a production rule exists, that gives the number and type of subexpressions. A composite expression

can have either a fixed number or a variable number of subexpressions. Each subexpression can be again an expression of a limited kind, i.e. limited to an expansion set[5] of expressions. In the description of the production rules below, the sets ExpressionSet, PatternSet, etc. are used for this purpose. Terminal nodes account for expressions that have no further subexpressions. The root set is the expansion set of root expressions. Below, the set of possible start nodes (or root nodes) is ExpressionSet.

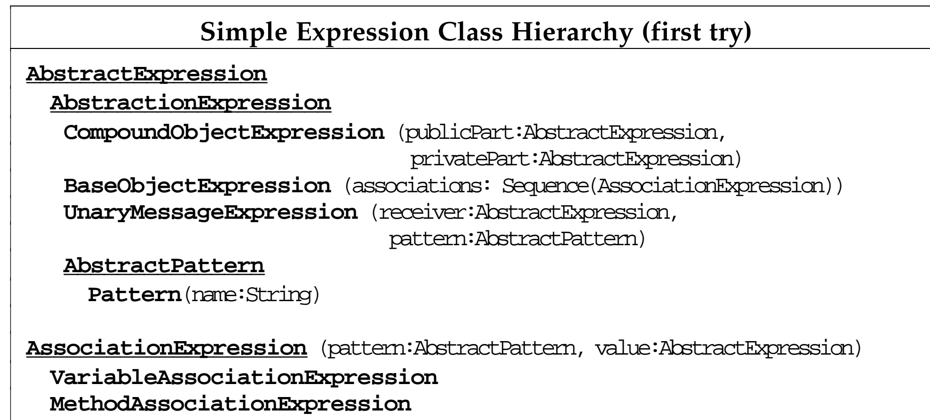
Simple Abstract Grammar

```
NonTerminal Labels      = { Application BaseObject CompoundObject
                            MethodAssociation VariableAssociation }
Terminal Labels         = { Name Literal }
Start Set               = Expression

ExpressionSet           = { Application } + AbstractionSet + PatternSet
PatternSet              = { Name }
AbstractionSet          = { BaseObject } + { CompoundObject }
AssociationSet          = { MethodAssociation } + { VariableAssociation }

Application               -> ExpressionSet x PatternSet
CompoundObject            -> ExpressionSet x ExpressionSet
BaseObject                -> AssociationSet*
VariableAssociation       -> PatternSet x ExpressionSet
MethodAssociation         -> PatternSet x ExpressionSet
```

Abstract grammars are easily transformed into a class hierarchy [Hedin89] [Madsen&Nørgaard88]. Expansion sets are translated into abstract classes. Each abstract class has as subclasses the classes that arise due to the translation of the elements of its expansion set. Each composite expression is translated into a concrete class having the subexpressions that comprise the composite expression as instance variable. The (expansion set) abstract classes are used for the typing of the components of a composite expression. Primitive expressions are translated into concrete classes that are not further aggregated.

The following class hierarchy is such a straightforward translation of a part of the abstract grammar for Simple. This class hierarchy contains the abstract classes AbstractExpression, AbstractionExpression, AssociationExpression and AbstractPattern that arise from the respective expansion sets ExpressionSet, AbstractionSet, AssociationSet and PatternSet (the abstract classes are underlined). The concrete classes UnaryMessageExpression, CompoundObjectExpression, BaseObjectExpression, MethodAssociationExpression and VariableAssociationExpression are the translation of the respective nonterminal nodes Application, CompoundObject, BaseObject, MethodAssociation and VariableAssociation. Finally, the terminal node Name is translated to the concrete class Pattern. Literals are not considered in the remainder of the text.

5 The name 'expansion set' is derived from the fact that an expansion set contains the types of the nodes that can be used in the 'expansion' of a subexpression.



Simple Expression Class Hierarchy (first try)

```
AbstractExpression
  AbstractionExpression
   CompoundObjectExpression (publicPart:AbstractExpression,
                             privatePart:AbstractExpression)
   BaseObjectExpression (associations: Sequence(AssociationExpression))
   UnaryMessageExpression (receiver:AbstractExpression,
                           pattern:AbstractPattern)
   AbstractPattern
     Pattern(name:String)

AssociationExpression (pattern:AbstractPattern, value:AbstractExpression)
  VariableAssociationExpression
  MethodAssociationExpression
```

This simple translation scheme only takes the aggregation structure of expressions into account. A better, but less trivial, translation must take the expected behaviour of each of the elements of the abstract grammar into account. For example, when evaluation behaviour is defined for the expression classes, it can be observed that most, but not all, expressions need an evaluation method. This can be illustrated in the above class hierarchy. Syntactically there is no problem in having a common syntactical representation for patterns used directly as expressions (i.e. as identifiers, an expression that when evaluated looks itself up in the context) and the patterns that are part of a message expression. So, a trivial translation of an abstract grammar in which both kinds of patterns have the same representation would map both onto the same class. Furthermore this class would be a subclass of the abstract expression class, since patterns are elements of the expression expansion set.

Patterns, used as expressions, can be evaluated; their evaluation is looking them up in the context. On the other hand, the pattern part of a message expression is merely used as a unique identifier, and may not be evaluated.

The class hierarchy that results from the trivial translation does not give a hierarchy that is suited for adding evaluation behaviour. In the case of a pattern that is used directly as an expression, an evaluation method is needed. In the case of a pattern that is part of a message expression, one needs to be able to test whether two patterns are the same. An uncoupling of the classes for both kinds of patterns is in order. Both play a different role in the evaluator and accordingly will lead to different objects with different protocols.

Accordingly a new class hierarchy of expressions can be constructed. Remark the uncoupling of both forms of patterns.

Simple Expression Class Hierarchy (second try)

```
AbstractExpression
  AbstractionExpression
   CompoundObjectExpression (publicPart:AbstractExpression,
                             privatePart:AbstractExpression)
   BaseObjectExpression (associations:Sequence(AssociationExpression))
  UnaryMessageExpression (receiver:AbstractExpression,
                          pattern:AbstractPattern)
  PatternExpression (pattern:AbstractPattern)

AssociationExpression (pattern:AbstractPattern, value:AbstractExpression)
  VariableAssociationExpression
  MethodAssociationExpression

AbstractPattern
  Pattern(name:String)
```

Remark also that association expressions are not part of the expression class hierarchy. The association expansion set is not a subset of the expression set. At first sight this could easily be amended. The syntax could be extended to allow expressions such as `x.3`, which would be a syntactically simpler construct equal to `[x.3]`. This would unnecessarily complicate the explanation of the implementation of Simple.

In general, an uncoupling of classes that have the same aggregation structure, but play different roles in the evaluation process suffices in order to create a "good" class hierarchy. Further examples will be given in the remainder of the text.

Evaluation is expressed as a method that is defined on expression objects. Evaluation is done in a context (which for Simple is an object again). The result of evaluating an expression is an object. Compositionality is achieved by encapsulating the compositional structure of each expression, such that a composite expression can only rely on the evaluation methods of its subexpressions to express its own evaluation.

AbstractExpression

eval:Context -> AbstractMetaObject

Figure 3.7

The abstract class "AbstractExpression" of which all expression classes will be derived contains one abstract method.

Abstract Class for Expression Objects

```
class AbstractExpression
  methods
   abstract eval:context result AbstractMetaObject
endclass
```

From the viewpoint of the evaluator adding a new expression "simply" involves extending the evaluator in a compositional way. From the viewpoint of, for example, the parser, adding a new expression class is a more complicated matter. For the time being however this problem will be skirted and we will presume that expression objects can be freely added. We will come back on this issue.

3.5.2 Representation of Objects and Full Abstraction

From our discussion about objects in the first part of the text it should be apparent that there exists a plethora of different kinds of objects, e.g. objects that have a class, idiosyncratic objects, objects that have an encapsulated state, objects that don't have an encapsulated state, objects that are defined as a specialisation of another object, primitive objects, objects that can be used as templates, … and so on. What is common to all these types of objects is that they can receive messages. A message consists of a pattern part and an arguments part. For the time being we will leave it open what form the arguments part can take (for Simple this is not very important since only unary messages are sent). We will first take a look at object representations that are too operational.

Objects could be implemented as, for example, sets of slots (named attributes) that can be searched. A hypothetical implementation is given below. Such an implementation is not abstract enough since message passing is not encoded as an atomic operation. Message passing needs to be encoded by the evaluator as looking up a slot and evaluating the body of this slot in some context.

```
class SlotObject
  instance variables
    slots:Set(Slot)
  methods
    concrete lookup:pattern result Slot
      slot := slots findSlot:pattern
      if slot found
        then [^slot]
        else [… raise an error …]
endclass
```

Although, it is presumable that slots will play an important role in the implementation of objects, it is a violation of the encapsulation principle if slots need to be exported for message passing. This representation of objects is not fully abstract. The encapsulation principles that govern at the level of the programming language are not respected in the representation of objects. In particular we will say that such an implementation has a non-encapsulated representation of objects.

So, the implementation of objects must be such that it encodes an atomic message passing operator, and the internal structure (e.g. inheritance structure) of objects remains hidden in the implementation of each object. The abstract class `AbstractMetaObject`[6] (see figure 3.8) presents the protocol that all concrete implementations of objects must have to ensure this.

Objects are represented as instances of concrete subclasses of the `AbstractMetaObject` class. The protocol of this class will include at least a subprotocol to send messages; i.e. each object representation includes at least a send method. The send method should implement message handling for the represented object. The send method accepts a pattern argument and an optional client argument. Accordingly, an actual call of the send method will include the message pattern, and optionally any additional information that must be transferred from the sender object (e.g. arguments of the message, or encapsulated parts in the case of Simple). The actual role of *client objects* will be discussed in a following section.

---

[6] Apart from speaking about expression objects (objects that implement expressions), context objects (objects that implement contexts), etc., the term meta-objects will be used for objects that implement objects, hence the name of this class.

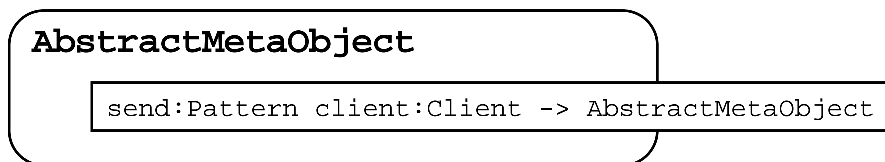
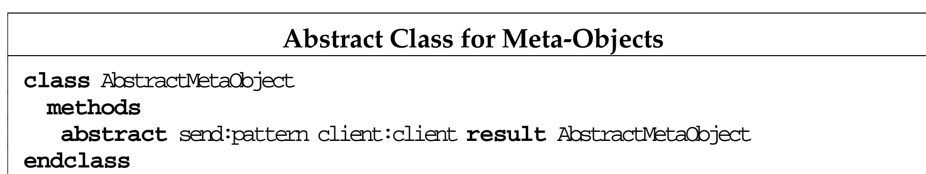


Figure 3.8

Internally, objects can be implemented as, for example, sets of slots, that can be searched. This implementation remains hidden for the user of an object. If this is respected, then objects are represented fully abstract. Stated otherwise, even at the implementation level it is not possible to directly access the private attributes of an object: encapsulation of objects is preserved at the implementation level. This will be called a fully encapsulated implementation of objects.

The class “AbstractMetaObject” of which all meta-object classes are to be derived has one abstract method.

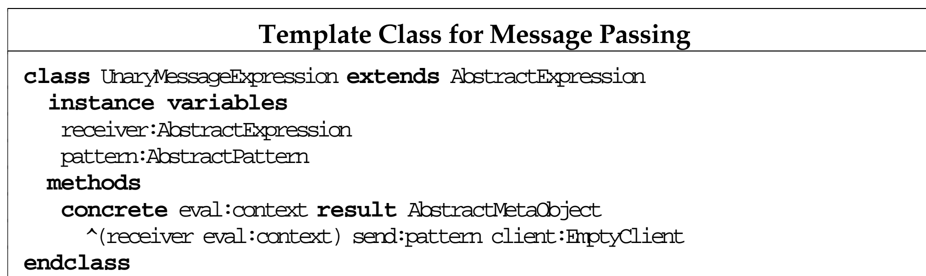


Abstract Class for Meta-Objects

```
class AbstractMetaObject
  methods
    abstract send:pattern client:client result AbstractMetaObject
endclass
```

3.5.3 Message Passing

Given the two abstract classes above, it is possible to express how message passing proceeds. This is encoded in the class “UnaryMessageExpression” as found below (we presume the existence of an empty client object "EmptyClient").



Template Class for Message Passing

```
class UnaryMessageExpression extends AbstractExpression
  instance variables
    receiver:AbstractExpression
    pattern:AbstractPattern
  methods
    concrete eval:context result AbstractMetaObject
      ^(receiver eval:context) send:pattern client:EmptyClient
endclass
```

This class ties together evaluation of expressions and sending messages to objects. Especially because message passing plays such an important role, and because this concrete class ties together two important abstract classes, the expression class for message passing will play an important role in the definition of the framework and it is the intention to be able to inherit it in all extensions to the framework.

Furthermore the above message passing class must be complemented with an abstract class for patterns. Patterns are used as unique identifiers in the implementation of message passing. They essentially implement an equality test.

Abstract Class for Patterns

```
class AbstractPattern
  methods
   abstract = pattern result Boolean
endclass
```

3.6 Concretisation to a Simple Object-based Language

We can now show how the above abstract classes can be made concrete in order to express an evaluator for Simple programs.

3.6.1 Abstraction Expressions and Object Structures

Calculus-objects are created by the evaluation of either the `CompoundObjectExpression` or the `BaseObjectExpression`. Both expression classes are concretisations of the abstract expression class. Their evaluation is simply composed of the evaluation of their components and the creation of an according meta-object (i.e. instances of `CompoundObject` and `BaseObject`, the code of which will be listed in the next paragraph).

To illustrate all the following class descriptions, they will be preceded by a short explanation. In this explanation underlined terms are syntactic objects that can be evaluated. For example "<u>`<a, b>`</u>" is a compound object expression. Patterns — instances of the `AbstractPattern` class — will be represented as `#x`. All other terms are evaluated objects that can, for example, be sent messages. For example "`<a,b>`" is a meta-object of type compound object. Note that these descriptions are only illustrative.

```
<publicPart,privatePart> eval:context
  -> <publicPart eval:context, privatePart eval:context>
```

```
class CompoundObjectExpression extends AbstractionExpression
  instance variables
   publicPart:AbstractExpression, privatePart:AbstractExpression
  methods
   concrete eval:context result AbstractMetaObject
     ^CompoundObject
          publicPart:(publicPart eval:context)
          privatePart:(privatePart eval:context)
endclass
```

```
[...; xi.ei; ... ; yi#fi; ...] eval:context
  -> [...; xi.ei eval:context; ... ; yi#fi eval:context; ...]
```

```
class BaseObjectExpression extends AbstractionExpression
  instance variables
   associations:Sequence(AssociationExpression)
  methods
   concrete eval:context result AbstractMetaObject
       local variables slots:Sequence(Slot)
     for each association in associations
        slots add:(association eval:context)
     ^BaseObject slots:slots
endclass
```

More interestingly are the so created meta-objects listed below. Both meta-object types conform to the above given abstract meta-objects, and implement the `send:client:` method. The role of client objects becomes clear in the following code. Client objects are used in the implementation of a compound object to accumulate its encapsulated part with the already existing client object, i.e. the currying rule of the calculus has an almost literal counterpart in the implementation of message passing to compound objects. In the implementation of a base object this client object is then used as a context in which to interpret the body of a selected method. So, the client object is used to carry information from the sender object to the receiver.

```
[...; xi.ei; ...] send:#xi client:client
  -> xi.ei valueIn:client
```

```
class BaseObject extends AbstractMetaObject
  instance variables
   slots:Sequence(Slot)
  methods
   concrete send:pattern client:client result AbstractMetaObject
     ^(slots findSlot:pattern ifAbsent:[^ERROR]) valueIn:client
endclass
```

```
<publicPart, privatePart> send:#x client:client
  -> publicPart send:#x client:<privatePart, client>
```

```
class CompoundObject extends AbstractMetaObject
  instance variables
   publicPart:AbstractMetaObject
   privatePart:AbstractMetaObject
  methods
   concrete send:pattern client:client result AbstractMetaObject
     ^publicPart
        send:pattern
        client: (CompoundObject publicPart:privatePart
                                privatePart:client)
endclass
```

3.6.2 Association Expressions and Slots

Methods and instance variables are added to an object by evaluating association expressions. An association expression is not a true expression. It is not a subtype of the `AbstractExpression`. Notice that the evaluation method is overloaded on `AssociationExpression`. Its evaluation returns a slot rather than an `AbstractMetaObject`.

Abstract Class for Association Expressions

```
class AssociationExpression
  instance variables
   pattern:AbstractPattern
   value:AbstractExpression
  methods
   abstract eval:context result Slot
endclass
```

```
pattern.value eval:context
  -> pattern.(value eval:context)
```

```
class VariableAssociationExpression extends AssociationExpression
  methods
   concrete eval:context result Slot
     ^VariableSlot key:pattern value:(value eval:context)
endclass
```

```
pattern#value eval:context
  -> pattern#value
```

```
class MethodAssociationExpression extends AssociationExpression
  methods
   concrete eval:context result Slot
     ^MethodSlot key:pattern value:value
endclass
```

The class `Slot` encodes instance variable and method slots in an object. They associate the pattern that identifies a slot, to its value in a given context. The class `Slot` is an abstract class that is newly introduced.

Abstract Class for Slots

```
class Slot
  instance variables
   key:AbstractPattern
  methods
   abstract valueIn:context result AbstractMetaObject
   concrete key result AbstractPattern
     ^key
endclass
```

The class Slot has two concrete subclasses for representing instance variables and methods. Their implementation is straightforward.

```
(key.value) valueIn:context
  -> value
```

```
class VariableSlot extends Slot
  instance variables
   value:AbstractMetaObject
  methods
   concrete valueIn:context result AbstractMetaObject
     ^value
endclass
```

```
(key#value) valueIn:context
  -> value eval:context
```

```
class MethodSlot extends Slot
  instance variables
   value:AbstractExpression
  methods
   concrete valueIn:context result AbstractMetaObject
     ^value eval:context
endclass
```

3.6.3 Message Passing

Messages are sent to objects by evaluating expressions of the type UnaryMessageExpression, or PatternExpression. The class UnaryMessageExpression has already been presented. Identifier lookup is interpreted as sending an according message to the context object.

```
pattern eval:context
  -> context send:#pattern client:EmptyClient
```

```
class PatternExpression extends AbstractExpression
  instance variables
   pattern:AbstractPattern
  methods
   concrete eval:context result AbstractMetaObject
     ^context send:pattern client:EmptyClient
endclass
```

Finally, the concrete pattern class can be given, as the last class in our implementation of the calculus. Essentially patterns can be compared for identity. In our encoding of patterns we use overloading on the pattern argument, i.e. the actual class of the pattern argument determines, together with the receiver, what exact equality test method is chosen. In the general case the equality test is applied to the commuted arguments. In the case where two patterns of the same concrete pattern class are compared, the names of the patterns determine equality.

```
class Pattern extends AbstractPattern
  instance variables
    name:String
  methods
    concrete = (pattern:AbstractPattern) result Boolean
      ^pattern = self
    concrete = (pattern:Pattern) result Boolean
      ^name = pattern name
    concrete name result: String
      ^name
endclass
```

3.6.4 Implementation of Simple, Summary

To implement Simple, concretisations of the abstract classes AbstractExpression, AbstractMetaObject and AbstractPattern were given. These classes represent the abstract concepts of expressions, meta-objects, and patterns. The expression class for message passing played an important role in tying all these abstract classes together. These three abstract classes and the expression class for message passing will form the kernel of our framework. As was shown in the previous section, for a first approximation these abstract classes are sufficiently detailed for the derivation of an implementation of a simple object-based language. In the next section we will show that they can be improved.

Furthermore, specifically for the implementation of Simple we introduced two new class hierarchies: that of AssociationExpression, and Slot. Although they are not a part of the basic framework, we will see that in practice the concept of slots plays an important role in the implementation of object-oriented programming languages. For this reason slots will be made part of the framework. The concept of slots can also be found in [Mulet&Cointe93].

Below are listed the key abstract classes in the implementation of Simple.

Abstract Class for Meta-Objects

```
class AbstractMetaObject
  methods
    abstract send:pattern client:client result AbstractMetaObject
endclass
```

Abstract Class for Expression Objects

```
class AbstractExpression
  methods
    abstract eval:context result AbstractMetaObject
endclass
```

Abstract Class for Patterns

```
class AbstractPattern
  methods
    abstract = pattern result Boolean
endclass
```

Abstract Class for Slots

```
class Slot
  instance variables
   key:AbstractPattern
  methods
   abstract valueIn:context result AbstractMetaObject
   concrete key result AbstractPattern
     ^key
endclass
```

3.7 Improving the Framework

The previous implementation of our Simple object-based programming language is limited with respect to extensibility. Although some of its major design decisions are made explicit (i.e. expressions, expression evaluation, objects, and message passing), we will show in what respects it is a “closed” implementation.

3.7.1 Reifier Methods

The first and most obvious question that is left open in the above implementation is the question of how new expression classes can be added to our Simple object-based programming language, and what the effect of this is on the rest of the environment. As we already said, from the viewpoint of a parser, adding a new expression class is a more complicated matter than it is from the viewpoint of an evaluator. With each new expression class a syntax must be associated from which this new expression class will be generated. Since it is the task of the parser to parse program text into a suitable internal representation, this involves extending the parser such that it recognises this new syntactic construct. This brings us into the realm of languages with an *extensible syntax*. In its most general form in a language with an extensible syntax, arbitrary syntactic constructs can be added to the language. This could for example be supported by a table driven parser.

We will take an approach that can better be termed as the *generic syntax* approach. Rather than allowing arbitrary new syntactic constructs, a few generic syntactic constructs, that will be termed reifier classes, or reifier methods, will be provided that can be instantiated. This approach is very similar to the concept of special forms in Lisp-like languages. It is inspired by reifier functions [Smith82] as found in procedural reflective languages; hence the name. It is also related to structured grammars as found in [De Hondt93].

From the viewpoint of adding new expression classes, compound expressions are the most important. We can identify two sorts of compound expressions: those that have a fixed number of subexpressions of possibly heterogeneous type (*compound expressions*), and those that have a variable number of subexpressions of homogeneous type (*aggregate expressions*). For both a generic syntax can be given. For compound expressions, the syntax is made generic in its use of the keywords that identify the syntactic construct. For aggregate expressions the syntax is made generic in its use of delimiters. In the latter the number of delimiters that can be used is fixed. Other delimiters could be devised, of course. What is important is that keywords and delimiters are separately recognisable lexical symbols.

```
GenericCompoundExpression
  -> [ BoldKeyword Expression { BoldKeyword Expression } ]

GenericAggregateExpression
  ->  LeftAggregateSymbol
       [ Expression { ";" Expression } ]
      RightAggregateSymbol

BoldKeyword            -> BoldIdentifier ":"
BoldIdentifier         -> BoldCharacter { BoldCharacterOrDigit }
BoldCharacter          -> "a" | "b" | …
LeftAggregateSymbol    -> "[" | "{"
RightAggregateSymbol   -> "]" | "}"
```

In a similar way a generic syntax can be given for non-compound expressions.

```
GenericPrimitiveExpression -> BoldIdentifier
```

A *generic expression* is instantiated by giving a concrete set of keywords, or delimiters. With respect to the evaluator, each such instance must be associated with an evaluation method, or stated otherwise it must be associated with an expression class that contains an evaluation method. The translation of an instance of a generic expression to its corresponding expression class can be done either by the parser (statically) or by the evaluator itself (dynamically). In view of our later ambitions of constructing a reflective language we will examine the latter case.

A possible dynamic implementation would translate generic expressions to an internal representation that corresponds to the following abstract grammar for generic expressions. The evaluation method for the components of this abstract grammar is responsible for dispatching to the right evaluation method according to the component's associated keywords, or delimiters.

Generic Abstract Syntax for Compound Expressions

```
ExpressionSet          = …  + { GenericCompoundExpression
                                GenericAggregateExpression
                                GenericPrimitiveExpression } + …

GenericPrimitiveExpression     -> IdentifierSet
GenericCompoundExpression      -> (IdentifierSet x ExpressionSet)+
GenericAggregateExpression     -> DelimiterSet x ExpressionSet*
```

Expression Class Corresponding to the Above Abstract Grammar

```
class GenericCompoundExpression extends AbstractExpression
  instance variables
    keywords:Sequence(Identifier)
    subexps:Sequence(Expressions)
  methods
    abstract eval:StandardContext result AbstractMetaObject
      … dispatch to the appropriate evaluation method according
      … to keywords
endclass
```

In principle the entire syntax of Simple can be recast in terms of instances of generic expressions. The syntax of expressions for the construction of base objects, for example, can be seen as an instance of a generic aggregate expression with the "[" and "]" symbols as delimiters. This is, with the current generic expressions, not possible however. In an ordinary abstract grammar, expansion sets introduce hierarchical structuring capabilities, i.e. they are used to put constraints on what

kinds of subexpressions can be used in a compound expression. This capability is lacking in the above generic expressions. All subexpressions of a generic compound expression can be arbitrary expressions. We will see how the introduction of evaluation categories amends this situation.

Finally, we note that a more object-oriented view on generic expressions is possible. In this view a compound generic expression is encoded as a special message, the receiver and arguments being expressions. Such an encoding will be called a *reifier message*. Correspondingly, an expression class then has a set of associated *reifier methods* that implement the evaluation for the corresponding reifier messages. In a similar vein generic expressions can be seen as instantiation messages of corresponding expression classes. If this is the case then we talk about *reifier classes*. Both approaches will be elaborated upon in the implementation of Agora.

3.7.2 Extra Indirection Needed in Context and Client Objects

As we saw before, evaluation of expressions is done in a context. In the preceding implementation this context is being built up during method lookup, i.e. method bodies are evaluated in a context that contains information that is local to the object in which the method is being looked up. This is a direct consequence of the encapsulation principle. On the other hand, sometimes it is necessary to transfer information from the sender object to the receiver object (otherwise objects would become autistic). Client objects are used for this purpose.

In the previous implementation, client and context objects were restricted to the encapsulated parts of compound objects. As such, they were directly implemented as instances of either `BaseObject` or `CompoundObject` (i.e. subclasses of `AbstractMetaObject`). In general, however, this is not possible since contexts and clients must contain other information as well.

Consider adding a self expression to the above calculus with the standard meaning of evaluating to the current receiver. During message passing, the change of receiver must be recorded. Furthermore this information must be accessible when this newly added self expression is evaluated. Rather than extending the evaluator with an extra “self” argument, it seems better to encode this “receiver” information as part of the client and context objects. So, client and context objects must be encoded as aggregates of all the different components that make up the context and the client. Contexts serve as an aggregate for all the information needed by the evaluator; clients serve as an aggregate for all the information passed from sender to receiver.

Below we can find the definitions for the context and client objects that can be used in the above implementation of the calculus. The encapsulated part object that formerly served as client and context is now an instance variable (called `private`) of the explicit client and context objects. The encapsulated part is a *public instance variable* since it can be read and set freely by all users of client and context objects. The evaluator of Simple must be adapted in order to take these definitions into account. This involves replacing all references to a client or a context as object, with respectively “`client private`” and “`context private`”, and, creating a client and context object the moment an encapsulated part is turned into a client or context.

```
class StandardContext
 public instance variables
  private:AbstractMetaObject
endclass
```

```
class StandardClient
 public instance variables
  private:AbstractMetaObject
endclass
```

The obvious advantage is extensibility of contexts and clients without having to adapt the entire evaluator. For example, for the introduction of a self expression, clients and context can be extended with an extra field. Furthermore the evaluator must be adapted such that 1) objects, on reception of a message, fill in the current receiver in the client; 2) the current receiver is copied from the client to the context when a selected method body is evaluated, and 3) the self expression is added, with the straightforward evaluation method of returning the current receiver from the context.

```
class ContextWithSelf extends StandardContext
 public instance variables
  currentReceiver:AbstractMetaObject
endclass

class ClientWithSelf extends StandardClient
 public instance variables
  currentReceiver:AbstractMetaObject
endclass
```

The class `SelfExpression` has the following form:

```
class SelfExpression
 methods
  concrete eval:(context:ContextWithSelf) result AbstractMetaObject
    ^context currentReceiver
endclass
```

Two notes should be made here. One is about the apparent parallel between the context and client class hierarchies. Due to this parallel one could be tempted to eliminate one of both. In the implementation of Agora, where objects have a more complex internal structure, we will show that clients and contexts do serve different purposes, and need not parallel each other.

The second is about the (lack of) compatibility between the existing object structures and the extension of the evaluator with a self expression. One part of this extension involves overriding the send method defined on objects such that the current receiver is added to the client object. Only the objects that have this overridden send method can use the self expression in their implementation. In some way this should be reflected in the adapted class hierarchy for expressions. The self expression can not be substituted in all program contexts where an abstract expression is expected[7]. It can be used only in those program contexts where an object that fills in the current receiver, is defined. A mechanism to control this is needed. This will be defined in the following section with the introduction of evaluation categories.

---

[7] The issue of typing has been avoided up until now, but another way to look at the above problem is that the self expression can not be made a subtype-subclass from `AbstractExpression`. Whereas the evaluation method for `AbstractExpression` has a context argument from `StandardContext` type, and, whereas the evaluation method for `SelfExpression` has a context argument with type `ContextWithSelf`, and whereas `ContextWithSelf` is a subtype of `StandardContext`, it can be concluded that due to the contravariance rule on method arguments, `SelfExpression` is not a subtype of `AbstractExpression`.

3.7.3 Evaluation Categories and Category Patterns

In the calculus and its implementation only side-effect free expressions are considered. When extending the calculus with side effects, one must also consider statements. The essential difference between an expression and a statement is that the former returns a result and the latter does not. Correspondingly, they have different evaluation methods.

Consider extending the calculus with side-effects. This involves adding statements such as a compound and an assignment statement, but also the imperative variant of message expressions.

```
class AbstractStatement
  methods
    abstract eval:context
endclass

class CompoundStatement extends AbstractStatement
  instance variables
    statements:Sequence(AbstractStatement)
  methods
    concrete eval:context
      for each statement in statements
        statement eval:context
endclass

class AssignmentStatement extends AbstractStatement
  instance variables
    pattern:AbstractPattern
    value:AbstractExpression
  methods
    concrete eval:context
      context assign:pattern value:(value eval:context)
endclass

class UnaryMessageStatement extends AbstractStatement
  instance variables
    receiver:AbstractExpression
    pattern:AbstractPattern
  methods
    concrete eval:context
      (receiver eval:context) send:pattern
endclass

class PatternStatement extends AbstractStatement
  instance variables
    pattern:AbstractPattern
  methods
    concrete eval:context
      context send:pattern
endclass
```

Secondly, the object structures must be extended such that imperative messages can be sent. Furthermore a mechanism must be provided to assign a new value to an instance variable attribute of an object. For the time being this latter is resolved in an ad hoc manner; in the framework for Agora a more definitive solution will be considered. Since imperative objects must also be able to accept functional messages, they inherit the standard message passing behaviour from the already defined standard objects. Only the abstract class for imperative objects is listed, all others are straightforwardly implemented.

```
class AbstractImperativeObject extends AbstractMetaObject
  methods
    abstract send:pattern client:client
    abstract assign:pattern value:AbstractMetaObject
endclass
```

Also the slot hierarchy must be adapted. Only the abstract class for imperative slot is listed, in the implementation of the concrete slots the value of an instance variable slot may be reassigned, the value of a method slot not.

```
class ImperativeSlot extends Slot
  methods
    abstract valueIn:context
    abstract value:newValue
endclass
```

The class hierarchy of statements is not related to the class hierarchy of expressions. The drawback of having different class hierarchies is that the according evaluation methods are not related to each other. Moreover, as is the case now, only the refinements of the abstract class of the (side-effect free) expressions are part of the framework. In the same vein, unary message expressions, as documented in the framework, are limited to messages that return a result. In practice other such *expression kinds* can be expected, that are essentially different in the types of the arguments, and result type of the evaluator. In fact, in the implementation of the calculus, the hierarchy of `AssociationExpression` is such an example. Similarly, the self expression of the previous section can be regarded upon as an expression kind that can only be evaluated in a context that keeps track of the current receiver.

To capture the existence of different expression kinds, we will introduce the notion of evaluation categories. Rather than overloading the evaluation method on unrelated expression hierarchies whereby each evaluation method has a possibly different signature, the evaluation method will be overloaded on the context argument also. Much in the style of multi-methods, the selection of the evaluation method does not only take the receiving expression object into account, but also the class of the context. The logic behind this is that, in practice, each expression kind is evaluated in its own particular context class. For example self expressions are evaluated in a context in which the receiver is recorded, statements are evaluated in a context that allows sequencing of expressions, ordinary expressions are evaluated in a standard context, etc. This gives rise to different evaluation categories. When these evaluation categories are encoded by means of overloading all different expression kinds can be part of the same expression hierarchy. Furthermore, one and the same expression can be evaluated in different evaluation categories. An example of this latter is the evaluation of message expressions. Formerly, two, and in the general case an unlimited number of, classes were needed for message expressions. After the introduction of pattern categories we will see how this can be encoded in one and the same class.

Evaluation categories can also be used to reintroduce the hierarchical organisation of syntactic structures in a system employing a generic grammar. Consider a generic compound expression. In principle the subexpressions of this compound expression are not constrained. In some cases they ought to be, however. For example in an extension of Simple with statements, the subexpressions of a statement sequence, must be again statements. To ensure this it suffices that the evaluation method associated with a statement sequence evaluates its subexpression in the appropriate ‘statement’ evaluation category.

The abstract class for expressions is extended to take overloading on the context into account. The context argument is annotated with a class (i.e. StandardContext). In the annotation of the context argument the + superscript indicates the fact that the evaluation method can be overloaded, in addition to being overridden, in later subclasses. In a concrete subclass the argument of a method that is overloaded on this argument is represented as a couple "(formal argument name: overloaded class name)". An example can be found in the adapted implementation for self expressions.

Rather than introducing a separate evaluation method for the evaluation of imperative expressions (i.e. one that does not return a result), we will expect all imperative evaluations to return a dummy result. In theory a more general solution could be adopted whereby the result of an evaluation is returned in a specially defined result aggregate (much in the style of context and client aggregates). For simplicity reasons we adopt the more ad hoc solution of returning dummy results.

Abstract Expressions with Overloading on the Context Argument

```
class AbstractExpression
  methods
    abstract eval:StandardContext+ result AbstractMetaObject
endclass
```

The self expression of the previous section now takes the following form:

```
class SelfExpression extends AbstractExpression
 methods
  concrete eval:(context:StandardContext) result AbstractMetaObject
    ERROR("the self expression must be evaluated in an appropriate
context")
  concrete eval:(context:ContextWithSelf) result AbstractMetaObject
    ^context currentReceiver
endclass
```

Similarly to evaluation categories, message pattern categories are introduced, although the introduction of different message kinds, such as imperative and side-effect free messages, occurs less often. The class AbstractMetaObject is extended such that the possibility is left open to override the send method on the type of the pattern argument.

Abstract Meta-Object with Overloading on the Pattern Argument

```
class AbstractMetaObject
  methods
    abstract send:AbstractPattern+ client:StandardClient
        result AbstractMetaObject
endclass
```

Since pattern equality is dependent on the classes these patterns belong to, pattern categories will mainly be used for overloading one and the same pattern in different categories. For example, a functional and an imperative method can be given the same pattern name, they will be differentiated by the categories of their respective patterns. So, in general, pattern categories will not be used to overload the send method for objects, but rather to differentiate, in looking up a pattern, patterns from different categories.

Finally, the class UnaryMessageExpression can be adapted so that it is compatible with all of the above. This class is a typical example of an expression

class that can be evaluated in different evaluation categories. The one thing that must be given special attention is that the pattern that is used in sending the message to the evaluated receiver object inherits the category from the evaluation context in which the entire message expression is evaluated. For this purpose the asCategory method is introduced on patterns. This is an ad hoc solution, in the section on classifiers and traces we will show a better solution for this problem.

Message Passing with Evaluation and Pattern Categories

```
class UnaryMessageExpression extends AbstractExpression
  instance variables
    receiver:AbstractExpression
    pattern:AbstractPattern
  methods
    concrete eval:(context:StandardContext+) result AbstractMetaObject
      ^(receiver eval:context)
          send:(pattern asCategory:context)
          client:EmptyClient
endclass
```

Adapted Abstract Pattern Class

```
class AbstractPattern
  methods
    abstract = pattern result Boolean
    abstract asCategory:StandardContext+ result AbstractPattern
endclass
```

```
--- example of adapted concrete pattern
class Pattern
  instance variables
    name:String
  methods
    …
    concrete asCategory:(context:FunctionalContext)
        result AbstractPattern
      ^FunctionalPattern name:name
    concrete asCategory:(context:ImperativeContext)
        result AbstractPattern
      ^ImperativePattern name:name
    …
endclass
```

3.7.4 Making the Layered Structure Explicit

Up until now the layered structure in the implementation of Simple is not made explicit. Some of the concrete classes of this implementation can be made more reusable by the introduction of abstract class attributes. An example is given below.

Final, More Reusable Version of Compound Object Expressions

```
class AbstractCompoundObjectExpression extends AbstractionExpression
  abstract class attributes
   ACompoundObject
  instance variables
   publicPart:AbstractExpression, privatePart:AbstractExpression
  methods
   concrete eval:context result: AbstractMetaObject
     ^ACompoundObject
        publicPart:(publicPart eval:context)
        privatePart:(privatePart eval:context)
endclass
```

Concretisation to 'Standard' Compound Object Expressions

```
class CompoundObjectExpression extends AbstractCompoundObjectExpression
  concrete class attributes
   ACompoundObject:CompoundObject
endclass
```

All expression classes in the hierarchy must be adapted accordingly, i.e. the classes `BaseObjectExpression`, `VariableAssociationExpression`, and `MethodAssociationExpression` must each be constructed as an abstract class with an abstract class attribute for the creation of instances of respectively `BaseObject`, `VariableSlot`, and `MethodSlot`.

An implementation in which all (meta-) objects are created by means of abstract class attributes is more open-ended than one in which the classes of the meta-objects are 'hard-coded'. Typical extensions of the evaluator that make use of this are debuggers. Rather than making the expression class hierarchy concrete with the standard implementations of meta-objects, the expression class hierarchy is made concrete with meta-objects with debugging facilities.

Other example usages of this layered structure can be found in optimisation of object representations, i.e. all meta-object classes can be overridden with meta-object classes that represent objects more efficiently.

3.8 Conclusion

To summarise we list the key classes involved in the improved framework used to implement and extend Simple.

The class `AbstractExpression` was extended such that different sorts of evaluation can be dealt with. Furthermore, context objects are introduced to bundle all the arguments of the evaluator. Context object are also instrumental for overloading the evaluation function to obtain different *evaluation categories*. A

special notation was introduced for overloaded arguments.

Abstract Expressions with Overloading on the Context Argument

```
class AbstractExpression
  methods
   abstract eval:StandardContext+ result AbstractMetaObject
endclass
```

Standard Context Object for Simple, Grouping All Evaluation Arguments

```
class StandardContext
 public instance variables
  private:AbstractMetaObject
endclass
```

The notion of generic expressions was introduced so that the framework can be extended with new kinds of expressions. Three sorts of generic expression were discussed.

It was shown that the expression class hierarchy must be extended with abstract classes with abstract class attributes for capturing the layered structure of the implementation of Simple. The adapted class hierarchy is shown below. Although the so obtained abstract classes are specific to the implementation of Simple (not all object-based languages need to have the notion of e.g. compound objects), a good rule of thumb can be distilled from this experience. This rule of thumb says that in an implementation all meta-objects must be created by means of abstract class attributes. This rule will be followed in the next sections.

Simple Expression Class Hierarchy

```
AbstractExpression
  AbstractionExpression
   AbstractCompoundObjectExpression (ACompoundObject)
     CompoundObjectExpression (publicPart:AbstractExpression,
                               privatePart:AbstractExpression)
   AbstractBaseObjectExpression (ABaseObject)
     BaseObjectExpression (associations:Sequence(AssociationExpression))
  UnaryMessageExpression (receiver:AbstractExpression,
                          pattern:AbstractPattern)
  PatternExpression (pattern:AbstractPattern)
AbstractAssociationExpression (AAssociation)
  AssociationExpression (pattern:AbstractPattern, value:AbstractExpression)
   VariableAssociationExpression
   MethodAssociationExpression
AbstractPattern
  Pattern(name:String)
```

The class `AbstractMetaObject` was extended such that the send method can be overloaded on the pattern type. Furthermore, client objects are introduced to bundle all the arguments of the send method.

Abstract Meta-Object with Overloading on the Pattern Argument

```
class AbstractMetaObject
  methods
   abstract send:AbstractPattern+ client:StandardClient
       result AbstractMetaObject
endclass
```

Standard Client Object for Simple, Grouping all Send Arguments

```
class StandardClient
 public instance variables
  private:AbstractMetaObject
endclass
```

The class UnaryMessageExpression was adapted to take evaluation and pattern categories into account. UnaryMessageExpression is an expression type that can be evaluated in any possible evaluation category. The pattern class is extended so that evaluation categories can be ‘inherited’.

Message Passing with Evaluation and Pattern Categories

```
class UnaryMessageExpression extends AbstractExpression
  abstract class attributes
   EmptyClient
  instance variables
   receiver:AbstractExpression
   pattern:AbstractPattern
  methods
   concrete eval:(context:StandardContext+) result AbstractMetaObject
     ^(receiver eval:context)
        send:(pattern asCategory:context)
        client:EmptyClient
endclass
```

Adapted Abstract Pattern Class

```
class AbstractPattern
  methods
   abstract = pattern result: Boolean
   abstract asCategory:StandardContext result: AbstractPattern
endclass
```

Chapter 4

Specialising the Framework with Inheritance

4.1 Introduction

In the previous chapter we introduced and discussed a framework for an object-based programming language. In this chapter we will specialise this framework to include inheritance. It will be shown that inheritance can be added without modifying the basic structure of the framework.

First we will discuss the issues involved in designing a language that employs some form of inheritance. Closely related to this is the topic of scoping and visibility rules in object-oriented languages. An overview of the design issues will be given. Based on this a full-fledged object-oriented programming language is presented (Agora). It is shown that although Agora differs from Simple in some fundamental ways, the framework presented in the previous section can be used as a skeleton to implement Agora. This implementation will be used to define a second layer of abstract classes in the framework that handles inheritance.

4.2 Inheritance, Design Issues

In this section we will discuss the design issues that arise when designing an object-oriented programming language that employs some form of inheritance. We use the term inheritance for both object-based inheritance and class-based inheritance. We already saw that both forms are closely related, they are both a form of incremental modification. The problems and design issues that are discussed are independent of whether one considers either class-based or object-based inheritance. To emphasise this, we will use the terms inheritor or heir (for subclass/inheriting object), and ancestor or parent (for superclass/ancestor object). The terms class and object will be used both, even though in most places 'class or object' would be more appropriate. First we will go into some general problems with inheritance, then we will consider the problems that relate strictly to *multiple inheritance*.

4.2.1 Inheritance and Encapsulation Problems

In a similar way that it is important for an object that it can encapsulate attributes, it is equally important that an inheritor can encapsulate the fact that it depends on an ancestor for the implementation of certain attributes. When an inheritor is not free to change its inheritance structure we say that the inheritance is exposed. It is important that inheritance is not exposed to future inheritors or clients that use instances of a class/object. Examples of exposed inheritance will be given in the section on multiple inheritance, but can also be found in typed object-oriented languages where the notions of type and class are identified with each other.

Another interaction between encapsulation and inheritance stems from the fact that in most object-oriented languages an inheritor can access its ancestor in two ways. One, by direct access to the private attributes of the ancestor (direct access to the implementation details). Two, by access to the public attributes of the ancestor (parent operations). The trade-off between direct access to the implementation details of an ancestor and using parent operations is discussed in [Snyder87].

If an inheritor depends directly on implementation details of its ancestor, then modifications to the implementation of the ancestor can have consequences for the inheritor's implementation. An inheritor that uses only parent operations is likely to be less sensitive to ancestor changes (it is more abstract). Inheritors that make use of the implementation details of an ancestor are said to inherit from their ancestor in a non-encapsulated way; inheritors that make use of parent operations only are said to inherit from their ancestor in an encapsulated way. That is to say, whereas strict encapsulation implies that the private attributes of an object are not directly accessible by other objects; similarly *strictly encapsulated inheritance* implies that the private attributes of a subobject[1] within an object are not directly accessible by other subobjects within that object.

One solution to this problem is to have all ancestor references done through parent operations. This implies that for each class/object two kinds of interfaces must be provided: a public interface destined for instantiating clients that use instances of that class/object and, a so called private interface for future inheritors that are called the *inheriting clients* of the class/object.

[1] Each object is composed out of subobjects according to the inheritance hierarchy.

4.2.2 The Need for Flexible and Controllable Inheritance

The need to control and abstract over the way inheritance hierarchies are constructed has been expressed in several ways and in several places. On the one hand it seems an obvious extension of the "incremental changes" philosophy of the object-oriented paradigm to be able to incrementally change entire inheritance hierarchies. On the other hand there is a need to control the complexity arising from the use of multiple inheritance [Hendler86] [Hamer92].

A notable example of the first is that given by Lieberman in [Cook87]. The question is how an entire hierarchy of black and white graphical objects can be incrementally changed so that the initially monochrome graphical objects can be turned into coloured objects. In present day systems, one either has to destructively change the root class of this hierarchy by adding a colour attribute, or one has to manually extend each class in the hierarchy with a colour subclass.

The second need stems from the observation that unconstrained multiple inheritance hierarchies often end up as tangled hierarchies. Multiple inheritance is less expressive than it appears, essentially in its lack to put constraints on multiple inheritance from different classes [Hamer92]. For example, one would like to put a mutual exclusion constraint on triangle and rectangle classes, registering the fact that a graphical object can not be both a triangle and a rectangle. Consequently, a graphical object class can not multiply inherit from both the triangle and rectangle class.

Multiple inheritance is used to combine existing classes in order to construct new classes. *Tangled inheritance hierarchies* occur in multiple inheritance hierarchies where the classes expose a high degree of possible combinations. This is often the case where classes are decomposed in an unusually fine granularity or where classes are decomposed into different views or perspectives. An example of a tangled multiple inheritance hierarchy is given below. The example shows points that can be implemented as either polar or cartesian points. Point movements can be bounded; both the cartesian and the polar point classes have their own way to implement these boundaries.

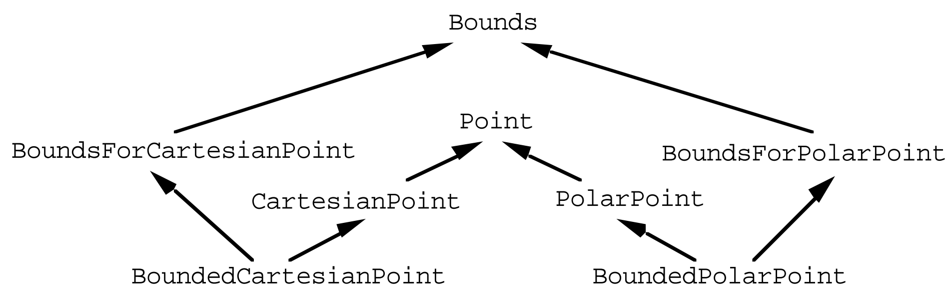


Figure 4.1

Some observations concerning tangled hierarchies can be made on the basis of the above hierarchy. The first observation is that there is a proliferation of classes. In the worst case, where there are two orthogonal sets of classes that can be combined, the possible subclasses are in the cartesian product of these two sets. In the example this is not the case since it does not make sense to combine e.g. BoundsForCartesianPoint with PolarPoint. The two sets of classes in the example are not orthogonal. In most languages with multiple inheritance all combinations of orthogonal sets of classes must be explicitly constructed. Most of the so constructed classes are so called *empty classes*, i.e. they contain no declarations. Empty classes serve only as a basis for instantiation. This phenomenon is referred to as the *proliferating subclass problem*.

Another observation that can be made is the fact that the above hierarchy does not represent all the information that is available about this hierarchy. Essentially this hierarchy only shows what combinations have been made but says nothing about what other possible combinations can still be made, or, equally important, what combinations can not be made. The fact that it is senseless to combine e.g. BoundsForCartesianPoint with PolarPoint is not represented in the hierarchy.

Even more important than the observation that constraints on the possible combinations are not represented, is the observation that the hierarchy does not represent the fact that the classes BoundsForCartesianPoint and BoundsForPolarPoint play a similar role in the respective resulting combination classes BoundedCartesianPoint and BoundedPolarPoint. The correspondence between the two can be summarised in the following table:

	CartesianPoint	PolarPoint
Add Boundaries	BoundsForCartesianPoint	BoundsForPolarPoint
Result	BoundedCartesianPoint	BoundedPolarPoint

Figure 4.2

Multiple inheritance mechanisms are poor at expressing the role an ancestor plays in the inheriting class; let alone to express the fact that two classes, used each as ancestor for a different inheritor, play a similar role (such as in the above example) for their respective inheritors.

4.2.3 Multiple Inheritance

It is important to distinguish single inheritance from multiple inheritance. Single inheritance is characterised by the fact that each inheritor has exactly (or at most) one parent. With multiple inheritance each inheritor can have multiple parents. Proponents of single inheritance say that multiple inheritance is not as yet well-understood and that in most cases single inheritance is satisfactory to express their problems. Proponents of multiple inheritance find that a sufficient number of 'real world' problems can not be expressed with a tree-structured classification mechanism.

The design issues in inheritance mostly concern multiple inheritance, since single inheritance is yet well-understood. In the design of an inheritance mechanism the chief concern is the interaction between inheritance and encapsulation, and also how flexible an inheritance hierarchy can be constructed.

In languages that use multiple inheritance each class/object can have multiple parents. This gives rise to *inheritance graphs* rather than *inheritance trees* (as is the case with single inheritance). This graph is a directed acyclic graph. Usually the graph has one single root (i.e. the root class/object). We will discuss different multiple inheritance mechanisms and the different problems involved. These different mechanisms are distinguished by how they treat the inheritance graph and how name conflicts are resolved. We will briefly overview the two major problems involved.

The first problem one has to face when an inheritor inherits from two or more parents is the problem of *name collisions*. Parents can have attributes with the same names. In [Knudsen88] three different sorts of name collision are identified: intended name collision, casual name collision and illegal name collision. These three sorts of name collision correspond roughly to respectively: 1) a name

collision where the attributes of the names that collide are intrinsically the same (e.g. in the case where the names are inherited, in their turn, of a common superclass) or intrinsically separable (e.g. in the case where the attributes are of a different nature such as a method and an instance variable, or the attributes have a different domain such as two methods that apply to arguments with disjunct types); 2) collision of names that are not related at all, but the names are separated by e.g. a qualifier; 3) collision of names that are not related and the names are not separated.

Conflicts are resolved in different ways (according to the specific language at hand). Either a mechanism is provided to rename the conflicting operation in one of the subclasses, or a mechanism is provided for qualified message passing, or it is an error to inherit two operations with the same name. With qualified message passing a message is qualified with an ancestor's name to direct the method lookup.

The second problem one has to face when dealing directly with the multiple inheritance graph is the *diamond problem* (or *common ancestor duplication problem*) depicted in figure 4.3. An inheritor B multiple inherits from classes/objects S1 and S2, which in their turn inherit from a common ancestor A. Now the question is whether the inheritor B will contain one or two subobjects A, and how the name collisions for the attributes defined on A will be be handled. That is, do we deal with the graph as is, or do we transform the graph into a linear chain, or do we transform the inheritance graph in e.g. a tree where both S1 and S2 have their own 'copy' of the ancestor A ?

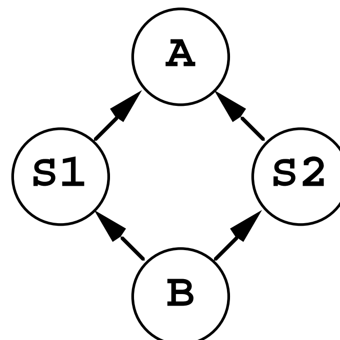


Figure 4.3

We will consider all problems that are related to the diamond problem. We will see how these problems are resolved, or not resolved, in the different multiple inheritance strategies.

Graph Multiple Inheritance

A first approach to multiple inheritance are strategies that deal with the inheritance graph directly without transforming it.

In *graph multiple inheritance* operations are inherited along the inheritance graph unless they are redefined in an inheritor. Conflicts arise when two operations with the same name are inherited along different paths in the graph. Conflicts are resolved either with qualified message passing or with renaming of conflicting attributes.

In general a distinction is made between 1) conflicts arising from two attributes that have the same name and are essentially different and 2) conflicts arising from two attributes that have the same name and are essentially the same (i.e. an attribute that is inherited along two different paths from one and the same non-direct ancestor, such as can be seen in the diamond problem). In general, whereas for the former case special provisions must be made (i.e. it is an error or qualified message passing must be used), the latter case is not really seen as a name conflict. This option is motivated by convenience but is shown to violate the encapsulation of inheritance.

It can be shown easily that not considering these "same attributes" as name conflicts exposes the use of inheritance. Consider the diamond figure again. An attribute ' x' defined on A is, according to the above rule, not conflicting in B. However if S2 is changed so that it is implementing attribute ' x' itself, this operation becomes a conflict in B. On the other hand, if names inherited from a shared parent along different paths are considered as name conflicts then in languages where all classes/objects inherit from a given root class/object (with attributes defined on the root, of course) any instance of multiple inheritance would result in conflicts for the root attributes.

The foremost reason why it is argued that we should not deal directly with the inheritance graph is that it exposes the inheritance structure [Snyder87]. The inheritor B should not be aware of how S1 and S2 are realised and whether they inherit from A or not. Let's take a look at the situation where S1 implements by itself (i.e. by not inheriting from A) all attributes that were previously defined in A and compare it with the situation where S1 inherits from A for the purpose of implementing these attributes. In the former case side-effects on the 'A-attributes' via S1 are not visible to S2, in the latter case they are. Just like in the previous paragraph this means that if one deals with the inheritance graph directly the implementor of S1 has not the freedom to reimplement S1 without inheriting from A or vice versa. The inheritance structure in S1 is exposed to its inheriting clients.

An additional problem related to graph oriented multiple inheritance is that of the *undesired duplicate parent operation invocation*. This will be illustrated with an example (example taken from [Snyder87]). It is the, by now almost classical, example of a point class with its two subclasses 'BoundedPoint' and 'HistoryPoint'. Points can be moved. History points record all point movements. Bounded points can only be moved within certain boundaries.

```
class Point
  instance variables: x y
  methods
    moveX:dx moveY:dy
      x := x + dx
      y := y + dy
    location
      ^(x,y)
endclass

class HistoryPoint
 inherits Point
 extended with
  instance variables: history
  methods
    moveX:dx moveY:dy
      history record: ("moved to: ", self location)
      super moveX:dx moveY:dy
endclass
```

```
class BoundedPoint
 inherits Point
 extended with
  instance variables: bounds
  methods
    moveX:dx moveY:dy
      if (self location + (Point x:dx y:dy)) within: bounds
        then super moveX:dx moveY:dy
endclass
```

In an attempt to make points that are both bounded and keep a history of moves, we define the BoundedHistoryPoint class that multiply inherits from BoundedPoint and HistoryPoint classes. In the definition of the move operation the move operation of both parents must be invoked. Of course this simple solution has the wrong effect that each move message sent to a bounded history point results in two move messages to the point subobject. The x and y instance variables of the receiving object are incremented twice. The classes BoundedPoint and HistoryPoint cannot be 'sufficiently' combined with graph multiple inheritance to form bounded history points.

```
class BoundedHistoryPoint
 inherits BoundedPoint HistoryPoint
 extended with
  methods
    moveX:dx moveY:dy
      super BoundedPoint.moveX:dx moveY:dy
      super HistoryPoint.moveX:dx moveY:dy
endclass
```

Linear Multiple Inheritance (implicit and explicit)

Implicit *linear multiple inheritance* strategies first flatten the inheritance graph for each class in a linear chain and then treat the collection of chains as a single inheritance hierarchy. Of course the linearisation strategy must obey some constraints. First of all it must preserve the ordering of classes/objects along each path in the inheritance graph. That is, an inheritor of some class/object may not become an ancestor of that class/object or vice versa during the linearisation process. Other restrictions may apply also. In CLOS, for example, programmers have some degree of control over the linearisation process by the order in which the superclasses of some class are listed; this order is also respected in CLOS's linearisation strategy.

Name conflicts are resolved automatically in the linearisation process. In case of a name conflict one of the conflicting attributes is selected even though there is no single best choice. Consider again the diamond picture from above, and consider a possible linearisation (A—S2—S1—B) of this inheritance graph (figure 4.4). Suppose both S1 and S2 define a conflicting attribute 'x'. In the linearised graph B will inherit the x attribute of S1, and the x attribute of S2 has been 'masked away' by S1, even though in an equally correct linearisation the reverse would be true.

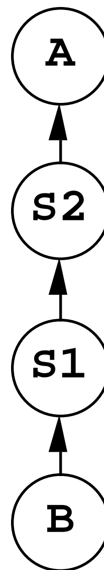


Figure 4.4

Related to this problem is the fact that an inheritor can not reliably communicate with its direct ancestors. Due to linearisation sometimes unrelated classes/objects are inserted between an inheritor and one of its direct ancestors. This is the case for S1 in the linearised inheritance graph above. Even though in the graph representation S1 has A as direct ancestor, in the linearised graph S2 has become the direct ancestor of A (A becomes an indirect ancestor of S1). Especially when name conflicts are involved this can give surprising results.

Another way to look at the above problem is that with linearised inheritance it is not possible to 'combine' two conflicting methods. In case of a conflict there is one single attribute that is visible to the inheritor. The other attributes are masked. To resolve this problem most languages with linearised inheritance provide a declarative mechanism that is called method combination (in CLOS, for example, an entire range of method combination mechanisms are provided [Moon89]). Method combination provides solutions to the above problems for a set of standard situations, still it does not take away the need for a class to be able to set up reliable communication with its direct ancestors.

Apart from the advantage that no name conflicts arise, linearised multiple inheritance performs well with respect to the problem of duplicate parent operations invocation. This is best illustrated by looking at the point, history point and bounded point example from the previous section. Whereas with graph multiple inheritance the classes BoundedPoint and HistoryPoint could not be combined to get the desired effect, this is perfectly possible with linearised inheritance.

In an encoding with linearised multiple inheritance the BoundedHistoryPoint class's move method must only invoke (and in fact *can* only invoke, since no name conflicts can arise) the move method of its ancestor once. This encoding is shown below.

```
class BoundedHistoryPoint
 inherits BoundedPoint HistoryPoint
 extended with
  methods
    moveX:dx moveY:dy
      super moveX:dx moveY:dy
endclass
```

This solution is heavily based on the knowledge that the inheritance graph will be linearised. In a possible linearisation, shown below, the effect will be as such that the move method in BoundedHistoryPoint invokes the one in HistoryPoint which in turn invokes the one in BoundedPoint which in his turn invokes the final move method in the Point ancestor. In fact what we first considered as a problem, i.e. the insertion of unrelated superclasses, now turns out to be part of a solution. We will see below that this sort of techniques can be generalised to what is called *mixin-based inheritance*, and that both the HistoryPoint and the BoundedPoint classes are best encoded as *mixins* (see also [Snyder87]).

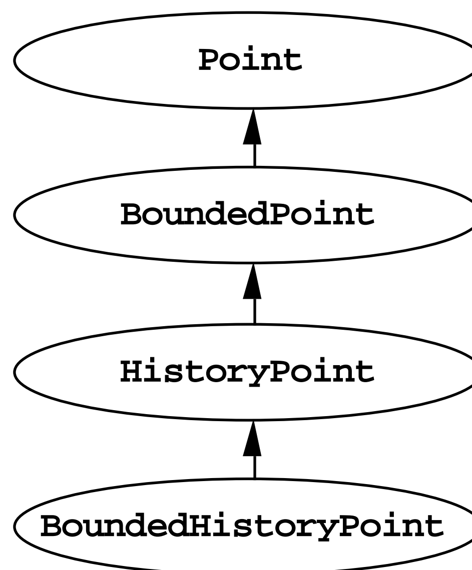


Figure 4.5

Although implicit linearisation solves some of the problems that occur in graph multiple inheritance, these problems are solved in a radical way. No name conflicts can occur due to changes in the inheritance hierarchy of some class (changes that do not alter the semantics of that class), but on the other hand conflicts are resolved even if there is no clear 'best choice'.

An important, and possitive, side effect of the unintended interleaving of an unrelated ancestor class is the concept of *mixin-classes*. Again, let's have a look at the diamond example, and its linearisation (A—S2—S1—B) above. It is possible for S1 to invoke parent operations that are not declared in its direct ancestor A, but due to linearisation are found in the newly assigned ancestor S2 (e.g. consider an operation x that is defined on S2 and not on A that is invoked via a parent operation from S1).

Going one step further, it is trivial to see that it is possible to have classes that have no apparent ancestor but that do invoke parent operations in a meaningful way. This sort of classes have been named mixin-classes since they rely on the linearisation to be 'mixed in' at the appropriate place (i.e. as inheritor from a class that provides the necessary operations) in the linearised inheritance hierarchy. The effect is that it is possible to create mixin-classes that can be

applied (mixed-in) to a set of different superclasses. (in mixin terminology also called base classes) The prototypical example of a mixin-class is the one that adds colour attributes to all sorts of base classes. We will see many more examples in the remainder of this text.

Mixins have been identified as very useful and flexible building blocks to construct inheritance hierarchies. Another approach to multiple inheritance uses mixins as the sole mechanism to create inheritance hierarchies, and is called *mixin-based inheritance* [Bracha&Cook90] [Bracha92] [Hense92] [Steyaert&al.93] [Codenie,Steyaert,Lucas92].

Mixin-based inheritance and its relation to multiple inheritance will be further explored in subsequent sections.

Tree Multiple Inheritance

If it is said that linear multiple inheritance solves name collision problems in a radical way by not having any name collisions at all, then *tree multiple inheritance* solves name collision problems in a radical way by always having name collisions. Tree multiple inheritance is directly motivated by a need to solve problems related with the diamond problem. We saw that graph multiple inheritance exposes the inheritance structure due to two reasons. One, by not duplicating parents that are inherited via different paths in the inheritance graph; two, by not signalling name conflicts when one and the same operation is inherited via different paths in the inheritance graph. With respect to these two problems tree multiple inheritance takes exactly the two opposing design decisions as graph multiple inheritance.

Just as with implicitly linearised multiple inheritance the multiple inheritance graph is transformed, but in a less radical way. Rather than transforming the inheritance graph for some class in a linear chain, the inheritance graph is transformed into a tree where each parent that is inherited via different paths in the graph has been duplicated. For the diamond above this results in the following tree.

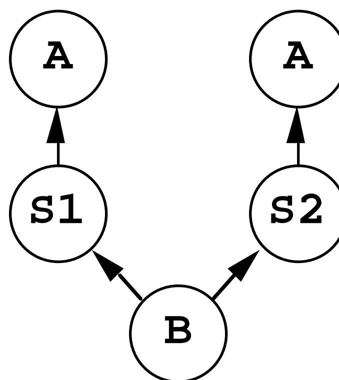


Figure 4.6

After this transformation all name collisions are treated on an equal footing. No name collisions can arise from attributes that are inherited via different paths, because all ancestors in joining paths have been duplicated.

Although tree multiple inheritance solves the inheritance encapsulation problems of both graph and implicitly linearised multiple inheritance it does so at a certain cost. But, let us first look at an example (example due to [Knudsen88]) where tree multiple inheritance works fine.

In the example we want to model a small part of the employees database from some university. There are two sorts of employments, lecturer and administrative staff. Each employee has a seniority. The seniority is used for example to calculate wages. In a very sensible way, the management of the seniority attribute is factored out in some abstract superclass called '`University Employee`'. For the example no other attributes are considered.

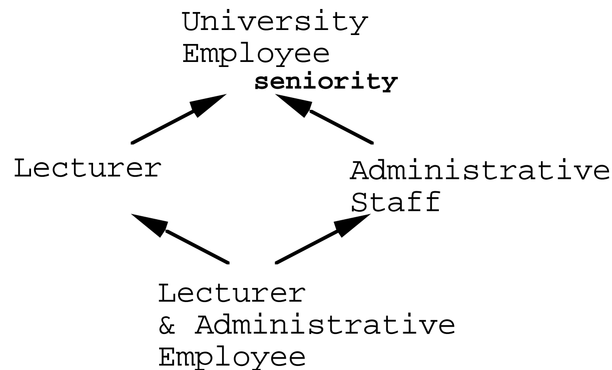


Figure 4.7

While designing the hierarchy we come to the conclusion that there are employees that take up both a position as lecturer and as administrative person. This is easily modelled by a class that multiply inherits from the lecturer class and the administrative staff class. What about seniority then ? The employee in question has two seniorities, one for each sort of employment. Of course this example only works fine with tree multiple inheritance. The transformation to a tree has the effect that for the "Lecturer&Administrative Employee" class, the Employee class is duplicated and as such also the seniority attribute.

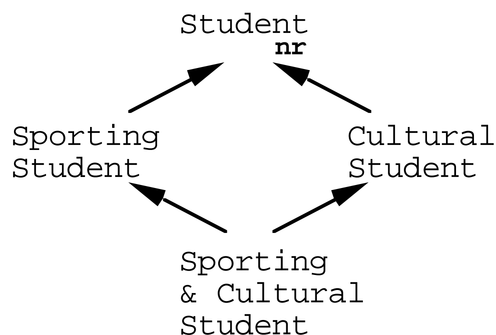


Figure 4.8

It is obvious that examples can be found where things don't work out as well as above. Consider the same university. This time we want to model a small part of the student database. Very similar to our employment database, there are two sorts of students: sporting students and students that take an interest in culture. Each student has a student number. The student number is used for administrative purposes. In a very sensible way, the management of the student number attribute is factored out in some abstract superclass student. For the example no other attributes are considered.

While designing the hierarchy we come to the conclusion that there are students that take up both an interest in sports as well as in culture. This is easily modelled by a class that multiple inherits from the sport student class and the cultural student class. What about student number then ? In contrast with the employee example all students have only one student number, regardless of

whether they are interested in sports or culture or both.

The student example can not be implemented easily in tree multiple inheritance. Not only because a cultural and sporting student will have two copies of his student number (that need to be synchronised), but also since the operations that are needed to manage this student number are now inherited from each duplicate of the student class and consequently raise name conflicts that need to be resolved. Solutions to this problem exist and are given in [Snyder87]. The idea is to avoid shared ancestors, leading to an extensive use of mixins, however.

The above problem imposes a serious limitation on the use of tree multiple inheritance since in most cases a very similar situation occurs if one wants to use a single root class in the inheritance graph that contains a set of general purpose attributes. These attributes are normally inherited by all objects in the system. With tree multiple inheritance, however, every class in the system that multiple inherits from two or more ancestors will need to resolve the conflicts that arise from the duplication of this root class. This imposes an absurd overhead on the use of multiple inheritance.

In conclusion, tree multiple inheritance avoids the exposure of inheritance that is inherent to graph multiple inheritance and implicitly linearised multiple inheritance, but does it at the cost of usability. Solutions to the problems of tree multiple inheritance exist, but must be found in an extensive use of mixins.

Point of View Notion of Multiple Inheritance
The *point of view notion of multiple inheritance* [Carré&Geib90] has grown out of a concern about the problems involved with *qualified message passing* as a means to resolve name conflicts. It handles somewhat orthogonal problems to the above discussed forms of multiple inheritance. Still we find it important enough to discuss it here.

Let’s start with reviewing what possible problems can arise due to the use of qualified message passing. In the discussion qualified message passing means that every message (either an ‘ordinary’ message or an invocation of a parent operation) can be qualified with the name of a class. Any class that is in the inheritance chain of the receiver of a message can be used as qualifier. Due to this qualification method lookup starts from the specified class rather than directly from the class of the receiver of the message. It is important to note this since there are useful restrictions to the above scheme. One useful restriction is to allow only qualification for invocation of parent operations. This can further be restricted to allow only qualification with the names of the direct ancestors of the class that invokes the parent operation.

Unrestricted qualified message passing exposes the inheritance structure of each class. Not only does it make visible all the names of the ancestors of a class, it also allows a user of a class to select a non-most-specific definition of some attribute defined on that class (*refinement inhibition problem*). Furthermore, qualified message passing encodes too much information about the class hierarchy, which may change, as constant information [Bobrow&al.86]. Due to this it can disable further refinement of a certain attribute. Consider a class A, with two methods x and y. Although at first glance it does not seem so, there is a fundamental difference between sending a message y, from within the method x, to self, and sending a message y, from within the method x, to self qualified with class name A. Both have the same behaviour for instances of class A, but for inheritors of class A it is in the latter case impossible to refine (overwrite) the method y (*genericity inhibition problem*).

It would seem reasonable to impose the above mentioned restrictions on the use of qualified message passing. These restrictions can be interpreted such that from the viewpoint of some class the only information about the inheritance hierarchy that it can rely on is the list of its direct ancestors. It should also be noted that using a renaming technique for inherited attributes to solve name conflicts is as safe as this restricted form of qualified message passing. Renaming must be done explicitly by the programmer. This is an obvious disadvantage. On the other hand we will see later on in the text that renaming has an advantage over qualified message passing.

The effect of the above restrictions (and also for renaming as a technique for solving name conflicts) is that name conflicts must always be resolved in an explicitly defined inheritor. This is not always desirable. Consider the following example (example due to [Carré&Geib90]).

```
class SportsMan
 inherits Person
 extended with
    instance variables
       sportsmanNumber
    methods
      cardNumber:x
        if self validateCard:x
          then sportsmanNumber := x
      validateCard:x
        … check if x is a valid sportsman card number
      cardNumber
        ^sportsmanNumber
endclass

class Student
 inherits Person
 extended with
    instance variables
      studentNumber
    methods
      cardNumber:x
        if self validateCard:x
          then studentNumber := x
      validateCard:x
        … check if x is a valid student card number
      cardNumber
        ^studentNumber
endclass

class SportyStudent
 inherits SportsMan Student
endclass
```

Both the class SportsMan and Student are given classes. A possible combination of these two classes results in name conflicts for the methods that manipulate the card number of either the person as a student or the person as a sportsman. Still a student that is also a sportsman will have two different card numbers and as such a class that is a combination of the classes SportsMan and Student must respond to two sets of messages to manipulate the two different card numbers. Here, in this case some sort of qualified message passing, to differentiate between the messages sent to some person as a student and messages sent to that same person seen as a sportsman, seems appropriate. It must be possible to send messages to a person object seen from different viewpoints. Hence the name of this sort of multiple inheritance.

We will make a sharp distinction between the multiple viewpoint sort of inheritance, where the interfaces of the combined classes are kept separate, and all the other discussed sorts of inheritance where the name conflicts in the interfaces of the combined classes are explicitly resolved in the inheritor.

Although both sorts of inheritance have to address some of the same problems — e.g. the common ancestor duplication problem —, they both have to address problems that are specifically related to either sort of inheritance. The problem of duplicate invocation of parent operations is, for example, not relevant for the point of view sort of inheritance, because in the point of view approach conflicting methods are not combined in the inheritor.

One problem that is specific for the point of view sort of inheritance has to do with self reference in ancestor classes. This problem can be made apparent in the previous example. Take John a student that is also a sportsman. Let's presume that we can refer to e.g. the card number attribute of John as a student as 'John Student.cardnumber' and to this same attribute of John as a sportsman as 'John SportsMan.cardnumber' respectively. Irrespective of the fact that this sort of qualified message passing is problematic with respect to encapsulation (refinement and genericity inhibition problems), a simplistic approach to this sort of qualified message passing will fail with respect to self references in the so-invoked methods.

What will happen to the following valid message: John SportsMan.cardnumber:423 ? The method lookup will correctly find the method named cardnumber: in the SporsMan class, and will invoke this method. This leads to the evaluation of '**self** validateCard:423', which is an unqualified message expression. The desired effect, of course, is that this expression is interpreted as '**self** Sportsman.validateCard:423'. Any other interpretation would lead to either an error or unpredictable behaviour. All 'naive' approaches to qualified message passing will fail to correctly interpret this sort of programs.

We will not give solutions to the above problem, for the time being it suffices to point out the problem. Solutions exist, in the form of a modified sort of qualified message passing [Carré&Geib90]). Later on in the text we will see that 'points of view' are strongly related to incremental modifications of objects in the prototype based approach to object-oriented programming.

One reason to introduce the viewpoint notion of multiple inheritance is that it enables us to give a crisp example of a particular sort of multiple inheritance that has largely been neglected, i.e. that of multiple inheriting of one and the same parent class. Consider again our sporting student example. Considering the fact that the SportsMan class and the Student class have very similar code it seems obvious to make the following abstraction:

```
class Member inherits Person
 extended with
  instance variables
    memberNumber
  methods
      cardNumber:x
        if self validateCard:x
          then memberNumber := x
      validateCard:x
        … check if x is a valid member card number
      cardNumber
        ^memberNumber
endclass
```

Of course this means that a possible sporting student class must inherit the Member class twice. The first time to express a student view and a second time to express a sporting view on the same person. It is obvious to see that, in the case that we do not want to define two 'empty' classes 'SportsMan' and 'Student' just to provide appropriate qualifiers, messages qualified with a class name are no longer sufficient in this example. Another means to make qualifiers must be provided.

Multiple Inheritance, Conclusions?
We confirm with Knudsen that:

> *"…by choosing strict and simple inheritance rules, one is excluding some particular usage of multiple inheritance …"*
>
> ***(Knudsen88)***

We add to this conclusion that there is a *trade-off between full encapsulation of inheritance and the expressiveness of the inheritance strategy* (in how effectively existing classes can be combined), and consequently, that it is sometimes necessary to expose the inheritance structure in a controlled way.

The trade-off between expressiveness and exposure of inheritance is apparent in two of the above examples. Firstly, in the example where one wants to avoid duplication of a shared parent (the sporting and cultural student example); secondly in the example where one wants to avoid duplicate invocation of a parent operation (the bounded history point example). In both cases it is necessary to expose some of the inheritance structure. In the former case both the '`SportyStudent`' class and the '`CulturalStudent`' class must expose the fact that they inherit from the 'Student' class and that they both don't mind that this parent will become a shared parent, so that the `Student` class can be shared in the '`Sporting&Cultural-Student`' class. In the latter case either the 'BoundedPoint' class or the '`HistoryPoint`' class must expose the fact that it inherits from the '`Point`' class and that it doesn't mind that another parent (hopefully with a similar behaviour) gets inserted between itself and its original Point parent, so that one of both can be assigned the other as parent (for the purpose of linearisation).

A possible solution could be devised where the programming language provides different inheritance operators: one that exposes inheritance and one that does not expose inheritance (much like in C++). Furthermore, a multiple inheritance operator that linearises the specified parents must be provided, one that keeps the inheritance graph as is and one that duplicates specified parents. This is more or less the direction taken in [Knudsen88], although there, the set of inheritance operators has been restricted to those that control the duplication and sharing of shared ancestors (unification inheritance and intersection inheritance).

In this text we propose a different solution based on mixins. We will extend the mixin-based approach with a mechanism to resolve name conflicts and we will show that, given this extension, mixins are sufficient to express all the above multiple inheritance hierarchies in an effective and simple way. Mixin-based inheritance was not only chosen because of its capacity to effectively construct multiple inheritance hierarchies but also for its capabilities to control and abstract over how these hierarchies are constructed, given the fact that mixins can be seen as attributes. The scope rules that emerge from the use of nested mixins also play an important role. We will show that mixins are exactly the right building blocks to construct (multiple) inheritance hierarchies.

4.2.4 Mixin-based inheritance

Mixin-Classes

In multiple inheritance languages that linearise the inheritance graph, it is possible to have classes that have no apparent ancestor but that do invoke parent operations in a meaningful way. This sort of classes has to rely on linearisation to be 'mixed in' at the appropriate place in the linearised inheritance hierarchy (i.e. as inheritor from a class that provides the necessary operations). These classes have therefore been named mixin-classes. The effect is that it is possible to create mixin-classes that can be applied to (mixed in) a set of different superclasses (in mixin terminology also called base classes).

A mixin-class in CLOS is a class that has no fixed superclass and as such can be applied to different superclasses. In CLOS terminology, this means that a mixin-class can invoke a Call-Next-Method, even though it has no apparent superclass. Mixin-classes in CLOS depend directly on multiple inheritance, and more specifically linearisation, for them to work.

The prototypical example is that of a colour mixin-class, that adds a colour attribute and the associated access methods, and can be applied to classes as different as vehicles and polygons. A typical example involving the invocation of parent operations (Call-Next-Method) is the "bounds" mixin that puts boundaries on the co-ordinates of a geometric figure. The actual base class can be taken from a set of possible classes. This could be, amongst others, a class Point, a class Line or a class Circle.

Mixin-Based Inheritance

Contrary to mixin-classes, in *mixin-based inheritance*, a mixin is not a class (a mixin cannot be instantiated for example), and multiple inheritance is a consequence of, rather than the supporting mechanism for, the use of mixins. In contrast to CLOS, in which mixins are nothing but a special use of multiple inheritance, mixins are promoted as the only abstraction mechanism for building the inheritance hierarchy [Bracha&Cook90] [Bracha92] [Hense92] [Codenie,Steyaert,Lucas92] [Steyaert&al.93].

To introduce mixins, we must return to our model of inheritance of the previous chapter. Inheritance was modelled as an incremental modification mechanism where a parent P (the superclass) is transformed with a modifier M to form a result $R = P \Delta M = P + M(P)$.

The above model is the essence of the model of inheritance in [Bracha&Cook90] where it is used as a basis for the introduction of mixin-based inheritance. In [Bracha&Cook90] it is also shown that mixin-based inheritance subsumes the inheritance mechanisms provided in Smalltalk, Beta and CLOS.

Whereas in conventional single or multiple inheritance the modifier M has no existence of its own (generally it is more or less part of the result R), the essence of mixin-based inheritance is exactly to view the modifier M as an abstraction that exists apart from parent and result. Modifiers are called "mixins". The composition operation Δ is called "mixin application". The class to which a mixin is applied is called the base class. In "pure" mixin-based inheritance, classes can only be extended through application of mixins.

The Δ operator sees to it that the parent P is passed as explicit parameter to the modifier M. In practice a mixin does not have its base class as explicit parameter, but rather, a mixin has access to the base class through a pseudo variable, in the same way that a subclass has access to a superclass through a pseudo variable

(e.g. the “super” variable in Smalltalk). In a statically typed language, though, this means that a mixin must specify the names and associated types of the attributes a possible base class must provide. This is why mixins are sometimes called “abstract subclasses”.

```
class-based inheritance
class R1
  inherits P1
  extended with NamedAttribute1 ... NamedAttributen
endclass

class R2
  inherits P2
  extended with NamedAttribute1 ... NamedAttributen
endclass

mixin-based inheritance
M is mixin
  defining NamedAttribute1 ... NamedAttributen
  applicable to base-class with2
    SuperAttributeSignature1 ... SuperAttributeSignaturen
  endmixin
class R1 inherits P1 extended with M endclass
class R2 inherits P2 extended with M endclass
```

4.2.5 Mixin-Method Based Inheritance

Mixin-based inheritance in the above form is an inheritance mechanism that is directly based on the model of inheritance as an incremental modification mechanism. It makes wrappers and wrapper application explicit [Bracha&Cook90] [Hense92]. In this section we generalise mixin-based inheritance in three ways.

Our mixins are based on a more general form of wrappers, where wrappers can have multiple parents. The notion of wrappers with multiple parents has already been pointed out in [Cook89]. The notion of multiple parents will be used to solve name-collision problems for multiple inheritance hierarchies where the interfaces are merged.

Furthermore, we extend the use of mixins to object-based inheritance. This sort of object-based inheritance is similar to implicit anticipated delegation [Stein,Lieberman&Ungar89], the resulting objects are comparable to split objects of [Dony,Malenfant&Cointe92]. We will show how this solves the problem of name-collisions in multiple inheritance hierarchies where the interfaces are not merged.

And finally we address the question of how mixins can be seen as named attributes of objects in the same way that objects and methods are seen as named attributes of classes. The general idea is to let an object itself have control over how it is extended. This results in a powerful abstraction mechanism to control the construction of inheritance hierarchies in two ways. Firstly, by being able to constrain the inheritance hierarchy; secondly, by being able to extend a class in a way that is specific for that class. Nested mixins are a direct consequence of having mixins as attributes. The scope rules for nested mixins are discussed, and shown to preserve the encapsulation of objects.

[2] This specification will be omitted in further examples for reasons of brevity.

Mixins with Multiple Parents

Consider the following classical example for multiple inheritance: we have a class Car and a class Toy and we want to combine their features to make toy cars. We want to merge both interfaces, so that only one version of the print-message is applicable to ToyCar (for the example no other attributes will be considered). In the definition of the print-method on the ToyCar-class we want to invoke the print-methods of both parents. That way we can combine e.g. the brand name of the car and the size of the toy to form the specification of the toy car. To do this, we need a mechanism to combine the conflicting methods.

```
Car-Mixin is mixin
  defining
    instance variables brand
    methods
      print
        brand print
 endmixin

Toy-Mixin is mixin
   defining
    instance variables size
    methods
      print
        size print
 endmixin

ToyCar-Mixin is mixin
  defining
     methods
      print
        --- invoke print operation of my 'car' part
        --- invoke print operation of my 'toy' part
  endmixin
```

Simple qualified message passing does not work for mixin-based inheritance since a mixin does not have a single base class that could serve as a qualifier. In the above example the ToyCar-Mixin mixin has no knowledge of the base classes to which it will be applied. In fact, it does not even know whether the car or the toy mixins will be part of the base class to which it will be applied.

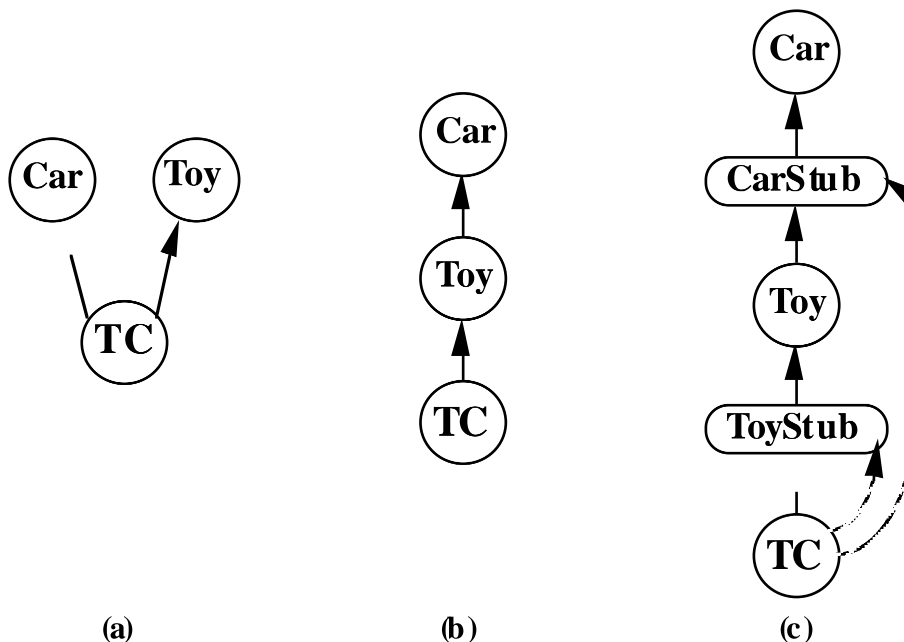


Figure 4.9

The problem here is that we have to deal with a linearised inheritance chain (figure 4.9b), but we still want to be able to refer to non-direct super classes (i.e. we want to simulate figure 4.9a). To do this we have to bring some 'hierarchy' into the chain. We already mentioned in the introduction that the solution should be found in mixins that have access to non-direct parents. We therefore introduced the notion of *stubs*. Just as mixins, the stubs have to be inserted at the right place in the inheritance chain (figure 4.9c). In this manner subclasses can use non-direct superclasses as parameters and 'mimic' a graph structure in the linear chain (dashed arrows in figure 4.9c). Stubs then serve as pointers to the place in the inheritance chain where method lookup should start when invoking parent operations.

```
ToyCar-Mixin is mixin
  needs Car-Stub Toy-Stub
  defining
    methods
     print
       print super:Car-Stub
       print super:Toy-Stub
  endmixin

class ToyCar inherits Root
  extended with Car-Mixin defining Car-Stub for ToyCar-Mixin
  extended with Toy-Mixin defining Toy-Stub for ToyCar-Mixin
  extended with ToyCar-Mixin
endclass
```

Using these stubs, a `ToyCar`-mixin can be created, that solves the name conflicts appearing when combining `Toy` and `Car`. To avoid problems with self references in inherited methods, *all* name conflicts have to be explicitly resolved here. It is not sufficient to simply resolve name conflicts occurring through combination of `Car` and `Toy`. Consider different implementations of the `print`-methods in the `Car`- and `Toy`-mixins, that do a self send of e.g. a message `getName`. This method could equally well be implemented in one of the ancestors of `Car` or `Toy`. It is therefore necessary to resolve *all* name conflicts in `ToyCar`.

The use of stubs must be restricted so that they can only be used to invoke parent operations of non-direct parents. In the case of the toy car, only the `ToyCar`-mixin should be able to use `Car-Stub` and `Toy-Stub`. On the other hand the definition of stubs cannot put constraints on the order in which mixins are applied. A concrete realisation of stubs should respect these constraints.

Separated Interfaces

Let us return to the example used in the introduction of the point of view notion on inheritance. We clearly want to keep the interfaces of the `Student` and `Sportsman` classes strictly separate. We want to be able to treat a `SportyStudent` as a student or as a sportsman, depending on the situation. We already mentioned that points of view are strongly related to incremental modifications of objects in the prototype-based approach to object-oriented programming.

In the previous discussion we left implicit the fact that mixins can be applied to objects. However, mixins can be used to dynamically extend objects in a prototype-based approach to object-oriented programming. New objects can be created by taking an existing object and extending it with a set of variables and methods. Similar to mixins in a class-based language we can identify a base object and a set of extensions. Here as well, extensions can be considered as separate abstractions. The terminology mixins and mixin application from the class-based case can be retained. Application of mixins to objects is an important part of the solution to our multiple inheritance problems. Consider the following example.

```
Person is class ...

Sportsman-Mixin is mixin
   -- same definitions as before
 endmixin

Student-Mixin is mixin
   -- same definitions as before
 endmixin

john is instance of Person;
...
johnAsASportsman is john extended with Sportsman-Mixin;
johnAsAStudent is john extended with Student-Mixin;
```

In the code displayed above, we first create an instance `john` of class `Person`. We can then create two new objects, `johnAsASportsman` and `johnAsAStudent`, each representing a different view on `john`. Being two dynamic extensions of `john`, they share its attributes (i.e. the attributes of `Person`).

Furthermore, as we now send messages to these new objects, the self reference problem is also resolved. When `setCardNr` is sent to either `johnAsASportman` or `johnAsAStudent`, `self getCardNr` is accordingly sent to this same initial receiver object. All `Person`-messages sent to `johnAsASportman` or `johnAsAStudent` are implicitly delegated to `john`.

Mixins as Attributes: Mixin-Methods

Applying the orthogonality principle to the facts that we have mixins and that an object consists of a collection of named attributes, one must address the question of how a mixin can be seen as a named attribute of an object. The adopted solution is that an object lists as mixin attributes all mixins that are applicable to it. The mixins that are listed as attributes in a certain object can only be used to create inheritors of that object and its future inheritors. Furthermore, an object can only be extended by selecting one of its mixin attributes. In much the same way that selecting a method attribute from a certain object has the effect of executing the selected method-body in the context of that object, selecting a mixin attribute of a certain object has the effect of extending that object with the attributes defined in the selected mixin. So, rather than having an explicit operation to apply an arbitrary mixin to an arbitrary object, an object is asked to extend itself. This form of inheritance has been named *mixin-method based inheritance* in the previous chapter.

Inheritance of mixins plays an important role in this approach. If it were not for the possibility to inherit mixins, the above restriction on the *applicability of mixins* would amount into a rather static inheritance hierarchy and duplication of mixin code (each mixin would be applicable to only one object).

A mixin can be made applicable to more or less objects according to its position in the inheritance tree. The higher it is defined, the more objects that can be extended with this mixin. In a programming language (such as Agora) where mixin-based inheritance is the only inheritance-mechanism available, this means that all generally applicable mixins (such as a mixin that adds colour attributes) must be defined in some given root object.

```
inheritance of a mixin-attribute
--- Root object attributes ---
 ColourMixin is mixin
   defining colour
 endmixin

 CarMixin is mixin
   defining enginetype
 endmixin

 car is object obtained by CarMixin extension of Root
 --- car inherits ColourMixin from the Root object
 colouredCar is object obtained by ColourMixin extension of Car
```

4.2.6 Mixin-based inheritance, A Solution to Multiple Inheritance Problems ?

Applicability of Mixins
An object lists as mixin-attributes those mixins that are applicable to it. What defines applicability of a mixin to an object ? There is no decisive answer to this question. The possible answers accord to the possible varieties of incremental modification mechanisms (e.g. behavioural compatible, signature compatible, name compatible modification, and modification with cancellation) used for inheritance [Wegner&Zdonik88]. In a regime where nothing but behavioural compatible modifications are allowed, only the mixins that define a behaviour compatible modification of a certain object are applicable to that object.

To put it another way, restricting the applicability of mixins puts a constraint on the possible inheritance hierarchies that can be constructed. This could answer our desire to constrain multiple inheritance hierarchies.

One such constraint is a mutual exclusion constraint on subclasses. The following example is taken from [Hamer92]. Consider a Person class with a Female and a Male subclass. A mutual exclusion constraint on the Female and the Male subclasses expresses the fact that it should not be possible to multiple inherit from both Female and Male at the same time. In terms of mixin-based inheritance, we have a Person class, with two mixin-attributes: `Female-Mixin`, and `Male-Mixin`. Once the Female mixin is applied to the person class, the Male mixin should not be applicable to the resulting class, and vice versa. This mutual exclusion constraint is realised simply by cancelling the `Male-Mixin` in the `Female-Mixin`, and by cancelling the `Female-Mixin` in the `Male-Mixin`. This solution relies on the ability to cancel inherited attributes. Other more formal solutions can be developed.

```
mutual exclusion constraint on classes
--- MarriedPerson class attributes ---
 Female-Mixin is mixin
  defining husband
  cancelling Male-Mixin
 endmixin

 Male-Mixin is mixin
  defining wife
  cancelling Female-Mixin
 endmixin
```

A Global View on the Inheritance Graph with Mixins
Mixin-based inheritance causes *explicitly linearised inheritance*. The order in which mixins are applied is important for the external visibility of public attribute names. Attributes in the mixin override the attributes of the base class having the same name. In absence of any name clash resolution mechanism, attribute name lookup is determined by application order.

Apart from this explicit linearisation, duplication of sets of attributes of shared parent classes (mostly used for duplication of instance variables) can be controlled explicitly by the programmer as well: not by the order of application, but by the number of applications of one and the same mixin.

Since mixin-based inheritance gives rise to linearised inheritance, it is obvious that the undesired duplicate invocation of parent operations can be resolved with mixins. A possible implementation of the bounded history point can be found below.

```
Point-Mixin is mixin
  defining
    instance variables x y
    methods
    moveX:dx moveY:dy
      x := x + dx
      y := y + dy
 endmixin

History-Mixin is mixin
   defining
    instance variables history
    methods
    moveX:dx moveY:dy
      history record: ("moved to: ", self location)
      super moveX:dx moveY:dy
 endmixin

Bounds-Mixin is mixin
   defining
    instance variables bounds
    methods
    moveX:dx moveY:dy
      if self (location getX + dX) within: bounds
           & (location getY + dY) within: bounds
      then super moveX:dx moveY:dy
 endmixin

class BoundedHistoryPoint inherits Root
  extended with Point-Mixin
  extended with Bounds-Mixin
  extended with History-Mixin
endclass
```

As the programmer has total control over the linearisation, there are no unforeseen insertions of unrelated classes between a class and its parent. This leads to the preservation of encapsulation of the inheritance hierarchy and makes parent invocations safe, as one always has control over the direct parent. It should be noted that the above solution is heavily based on global information of the inheritance graph. The bounded history point can only be constructed as a linear chain of the point, history and bounds mixin because we have information about the way each of them invokes parent operations. Since a mixin is an abstract subclass, the parent operations it invokes are part of its interface.

Let us now take a look at the common ancestor duplication problem. Reconsider

the examples of the sporty student and the univesity employee that were used to illustrate the common ancestor duplication problem. It is obvious that these are examples of the view on inheritance in which the interfaces of the combined classes are kept strictly separate. They can easily be described in a mixin-based approach. One remaining problem is that we want duplication of the attributes inherited from the common ancestor in the first example, while we want only one shared copy in the latter. This can easily be resolved here by applying an `Employee-Mixin` twice in the first example, and a `Person-Mixin` only once in the second. This results in the following definitions.

```
john is instance of Person;

johnAsASportsman is john extended with Sportsman-Mixin ;
johnAsAStudent is john extended with Student-Mixin ;

johnAsALect is john extended with Employee-Mixin Lect-Mixin
johnAsAnAdmin is john extended with Employee-Mixin Admin-Mixin
```

Mixin-Methods, Conclusions and Open Questions
In response to our analysis of multiple inheritance we proposed an inheritance mechanism based on mixins. We extended the mixin-based approach with mechanisms to resolve name conflicts. We showed that, given these extension, mixins are sufficient to express an entire range of multiple inheritance hierarchies in an effective and simple way. Mixins are so expressive because they allow unanticipated combinations of behaviour to be made. If uncontrolled, one faces an explosion of possible combinations of mixins. A mechanism to control this combinatorial explosion is needed. Mixin-methods are proposed as a uniform framework to control and make abstraction of the way multiple inheritance hierarchies are constructed. Central to this are the notions of applicability of mixins and dynamic application of mixins. Due to the treatment of mixins as attributes, mixins can be inherited and overridden. This introduces an extra level of abstraction in the way classes are extended that is not available (to the authors' best knowledge) in present day object-oriented languages.

The approach suggested was based on two different views on the inheritance hierarchy. In one view the interfaces of the combined classes were merged, in the other they were kept separate. One question now suggests itself: should it be possible to combine these two views ? In other words, should we be able to merge and separate interfaces within one single branch of the hierarchy ? These problems are related to the problems with split objects [Dony,Malenfant&Cointe92] and to the modelling of inheritance with explicit bindings [Hauck93]. They are left open for future research.

Another question that was left open is a more formal approach to restricting the applicability of mixins. In the next section we will show how mixin applicability can be restricted when a mixin depends on the implementation details of the base class it is applied to. A natural form of nesting of mixins will result from this. Other mechanisms for restricting the applicability of mixins will be shown in the section on extensions to Agora.

4.3 Visibility and Nesting in Object-Oriented Languages

Visibility rules are an important issue in the design of a programming language. In an object-oriented language this is especially so. Names (Identifiers) are very important in object-oriented languages. Objects are essentially collections of *named* attributes and defining an object is essentially the definition of a collection of names. Furthermore, the notion of encapsulation puts an a priori restriction on visibility.

This section is not about visibility rules for attributes that are in the interface of some object and are accessible through passing a message to that object (e.g. this section is not about different views on one object), but rather about the visibility rules of the attributes that are directly (i.e. without message passing to an explicit receiver) accessible for some object, and that are part of this object's encapsulated part.

The scope of an identifier is defined as the program code in which this identifier is visible. A name space is defined as a collection of all identifiers with a same given scope. Name spaces can be nested.

Most object-oriented languages define the scope of identifiers more or less ad hoc. In those languages (including Smalltalk), scope rules do not emerge from nesting, but rather for each kind of "variable" a different lookup strategy is defined. Smalltalk for example, has a blend of variables (class variables, class instance variables, global variables, pool variables, instance variables, arguments, local variables) each with their own visibility rules.

Block and nested structures have come into disfavour in object-oriented languages (with the notable exceptions of Simula and its descendant BETA). Block structures provide locality. The lack of locality in e.g. Smalltalk, where all classes reside in one flat name space, has its drawbacks to structure large programs[3]. Block structures are a natural way to hierarchically structure name spaces (modules are an alternative). Accordingly scope rules can be imposed. Typically the scope of an identifier declared in some block includes this block and all the blocks enclosed in it, but not the enclosing blocks.

Introducing block structure in an object-oriented system is a very delicate operation [Buhr&Zarnke88]. This is because the "natural" form of scoping that emerges from the nesting of blocks -- identifiers declared in some context are visible in blocks declared in the same context -- can seriously interfere with the notion of encapsulation.

One must take care since in an object-oriented language in which objects are considered to be encapsulated, this encapsulation implies that each object has a separate name space; similarly strictly encapsulated inheritance implies that each subobject[4] within an object has a separate name space. The intention is to regulate the sharing of name spaces of subobjects. While this breaks the encapsulation of subobjects, objects are still considered as totally encapsulated, i.e. access to the encapsulated part of an object is reserved to the implementation of the public part of that object, but one subobject can access the encapsulated part of another subobject within the same object (mediated by the above discussed rules of course).

---

[3] In Smalltalk this is partially remedied with the category concept. Classes are organised into categories. Categories however are only for documentation purposes.

[4] Each object is composed out of sub-objects according to the inheritance hierarchy.

Sometimes there is a need to share name spaces between objects, rather than subobjects. The above mentioned class variables and global variables, as found in Smalltalk, are examples of such name spaces that are shared by a number of (or all) objects. In the same way that the scope rules for nested mixins regulate the sharing of name spaces of subobjects, it is obvious that another set of scope rules can regulate the creation of shared name spaces for objects. This is normally what is done with nested classes, and will be discussed in the section on class nesting.

4.3.1 Is There a Need for Scope Rules for Encapsulated Attributes ?

The visibility rules that are discussed in this section apply to the names of the private attributes of an object. Alternatively, visibility rules can be, and have been, devised for the names of the public attributes. In the normal case any object can send any message to all the objects it has knowledge of, provided that the messages it sends are in the receiver's interface. Sometimes it is desirable to restrict this. In its most general form the invocation of a particular method can be restricted to a limited set of objects. Vice versa, it is possible that all objects that have knowledge of some common object, can not all send the same messages to this common acquaintance. In most cases visibility restrictions on the interface of an object take a particular form. Visibility is, in these cases, based on the fact whether one object is derived from another object, according to inheritance.

It can be argued that in presence of visibility restrictions on the interface of an object, there is no need for encapsulated attributes. In fact this is argued in [Ungar&Smith87] for the programming language Self. We will take a closer look at this.

Self is a slot-based language, i.e. variable access and method invocation take the same form. This is realised by access-methods. Each variable declaration introduces two access-methods: one to retrieve the value and one to store some value in the variable. Hence no assignment statement nor identifier lookup is needed in slot-based languages. All variable access takes place by sending messages.

Uniform access to state and behaviour has certain important advantages. First of all the message passing paradigm is not diluted with assignment and state access or identifier lookup. Secondly it allows a programmer to freely re-implement variable declarations into corresponding method declarations without having to check all the users of a variable. Furthermore, when inheriting from a class, all variable declarations can be overridden with method declarations (override both or one of both access-methods), or vice versa.

Since in slot based languages all variable access is done by message passing, this leads to very verbose programs. An object that needs to access one of its variables must send a message to its 'self', the receiving object. Special provisions are made to overcome this problem. For instance, in the language Self all messages that are sent from an object to this object itself (i.e. to the self) can omit the receiver. In the remainder of this text messages with an implicitly determined receiver (not necessarily the self), will be called 'receiverless messages'.

Self objects are 'flat' objects. All attributes (called slots) are part of an object' s interface, including those slots that correspond to what would be called instance variables elsewhere. Encapsulation is realised by dividing the interface in public slots and private slots. All possible objects can send messages that invoke public slots. Only an object itself can send messages that invoke private slots. As such Self could be called an object-oriented programming language with encapsulation. Encapsulation depends on the identification of the sender of a

message. An object has, as a sender of messages to itself, a greater accessibility to itself than other objects. However, we will argue that the encapsulation achieved this way is essentially module based.

Of course, visibility rules for interfaces of objects interfere with inheritance. Are visibility properties, associated to some method, inherited or not ? This question is the more relevant, in view of the encapsulation of inheritance, if these same rules are used to enforce encapsulation. We will explore some variations for public and private slots, and we will show that only module based encapsulation can be supported.

A private slot can not only be made visible for the container object itself but also for its inheritors. This corresponds to non-encapsulated inheritance: an inheritor can access the private slots of its parent. It should be noted that an object can also access the private slots of non-direct ancestors. This need not always be desirable.

Conversely, a private slot can be made visible to the ancestors of the container object. This is not only a prerequisite for exploiting the full potential (i.e. the ability to override variable slots with methods in inheritors) of slot-based languages, but results in a cumbersome semantics otherwise. As depicted in the following figure, an ancestor can only access its own private attributes by sending messages to the inheritor that it is part of. Consequently the ancestor also has access to the private slots of its inheritor.

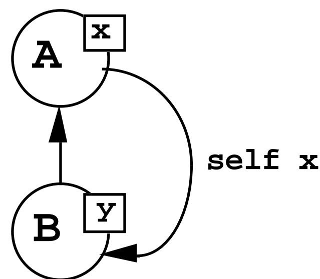


Figure 4.10

Private slots that are visible to ancestors can give rise to *module based encapsulation* [Ungar&Smith87]. In prototype based languages parent-objects can be used to store shared attributes. Typically, the collection of methods that implement a data type are stored in an object shared by all members of that data type. Parent objects that are used for this purpose are called 'traits' objects in Self. Traits objects play the role of classes in prototype based languages. The result is that all objects that belong to a certain 'data type' (e.g. all point objects) are derived from one and the same traits object. The traits object has as ancestor access to private slots of all the elements of the data type it implements. Every method in the implementation of the data type has access to the private slots of the elements of the data type.

Even in cases where private slots are made visible only to an object itself (and not to its descendants or ancestors), it can be shown that this leads to a degenerate form of module based, rather than object-based encapsulation. This can be observed in a method that takes an actual argument that is the same as the receiver of the method. This method has, in contrast with object-based encapsulation, access to all the private slots of both the receiver and the argument object.

As we saw earlier module based encapsulation is useful in cases where for example two elements of one and the same data type need to be compared. Other

useful examples include the initialisation/creation of elements of a data type. However, as we argued earlier module based encapsulation is best provided by a separate language construction, i.e. modules.

To conclude: although visibility restrictions on the interface of an object are very useful and can be used as a way to achieve encapsulation, they serve the purpose of encapsulation far from perfect. The essence of the above is that visibility restrictions on interfaces allows the separation of private from public attributes, but nothing is provided to structure the name space of the private attributes. We will see in the following sections that there is a need to structure the private name spaces of objects that are more sophisticated. Two important concepts were discussed in this section: that of slot based languages, i.e. languages where message passing is not diluted with state access primitives, and that of receiverless messages, i.e. messages that have an implicitly determined receiver.

4.3.2 Nested Classes, Classes as Attributes

Having classes as attributes, or having *nested classes* can serve two different purposes:

• locally visible classes: classes that are only visible in some local context

• locally defined classes: classes that are only meaningful in some context

In most object-oriented languages, classes reside in a name space shared by all objects. Indeed, almost all classes should be visible for all objects. Still, the ability to have classes that can only be used in a limited context is useful. In most cases this amounts to having a class as attribute-value of a private attribute of some object or class. On the other hand for class-based inheritance the ability to have classes as attributes normally implies class nesting, and as such it is not possible to have locally visible classes without nesting.

Apart from restricting the scope from some class, a class can be nested in another class in order to express the fact that a class only has meaning in relation to an instance from the enclosing class. This kind of nesting usually occurs when the nested class is a public attribute of instances of the enclosing class. Consider the following example (example from [Madsen&Møller-Pedersen89]):

```
class Grammar
  class Symbol
    methods
      isTerminal
      isNonTerminal
  end Symbol ;

  methods
    … grammar methods come here
end Grammar ;

G1 : Grammar;              S1,S2 : G1.Symbol ;
G2 : Grammar;              T1,T2 : G2.Symbol ;
```

A class Grammar is defined and a class Symbol is defined local to this Grammar class. Notice that there is no sub- or superclass relation between Symbol and Grammar. The example is such that the class Symbol is a public attribute of instances of class Grammar. The class Symbol is used to represent the lexical symbols that occur in a grammar. With each grammar a different set of symbols is used. This is naturally expressed by the fact that the class Symbol is an attribute of instances of — and is nested in — the class Grammar. As can be seen in the

example, instances of the Symbol class can only be created by selecting the Symbol attribute of an instance of grammar. In fact we can talk about different 'versions' of class Symbol. In the example the classes `G1.Symbol` and `G2.Symbol` are different, and in some sense incompatible, classes.

As said before, for class-based inheritance the ability to have classes as attributes normally implies class nesting. When nesting classes, the scope rules that are naturally connected with nested block structures result in shared name spaces for objects. This is obviously apparent in the grammar example. Presume the existence of a set of instance variables in the class grammar. Instances of some version of the Symbol class have direct access to the instance variables of the grammar instance on which they depend. Although the example is taken from BETA, a language in which objects are not encapsulated, we will illustrate the effect of nested classes on encapsulation.

Although nested classes can be very useful (as is shown in both [Madsen87] and [Buhr&Zarnke88]), they can be used by a programmer to break the encapsulation of objects. The next example is an example of class nesting. Both b1 and b2 can access the same variable i. Modification of this variable in, let' s say b1, has an effect on the variable seen by a and b2. So, instances of class B can directly access the instance variables of an instance of class A even if there is no sub or superclass relation between A and B.

```
class A extends SuperOfA
 i : Integer ;
 class B extends SuperOfB
  -- i is visible here !
 end B ;
end A ;
a : A ;
b1 : a.B ;
b2 : a.B
```

Non-encapsulated inheritance and nested classes do not mix very well. This is apparent in languages such as BETA [Madsen87]. Identifier lookup is ambiguous, because in every class, two different contexts can be consulted: the surrounding block context or the context of the superclass. In the above class nesting example this ambiguity would be apparent if an instance variable with the name "i" were defined in the super class of B. This problem is resolved by giving priority to one of both name spaces in case of a name clash, e.g. by first looking in the super class chain and then in the surrounding scope (here again the superclass chain must be searched and so on …). Ironically, nested classes, that can be used to break the encapsulation of objects, can only be used unambiguously in a language with strictly encapsulated inheritance.

4.3.3 Nested Mixins

In most object-oriented languages a subclass can access its superclass in two ways. One, by direct access to the private attributes of the superclass (direct access to the implementation details). Two, by access to the public attributes of the superclass (parent operations). A mixin is applicable to a class if this class provides the necessary private and public attributes for the implementation of the mixin. This puts an extra restriction on the applicability of a mixin.

The trade-off between direct access to the implementation details of a superclass and using parent operations is discussed in [Snyder87]. If a mixin depends directly on implementation details of the class it is applied to, then modifications to the implementation of the base class can have consequences for the mixin's

implementation. A mixin that uses parent operations only is likely to be applicable to a broader set of classes (it is more abstract). Mixins that make use of the implementation details of a superclass are said to inherit from their superclass in a non-encapsulated way; mixins that make use of parent operations only are said to inherit from their superclass in an encapsulated way.

One solution to this problem is to have all superclass references done through parent operations. This implies that for each class, two kinds of interfaces must be provided: a public interface destined for classes (= instantiating clients) that use instances of that class and, a so called private interface for future subclasses (= inheriting clients).

The solution we adopt is to differentiate between mixins that don't and mixins that do rely on implementation details of the base class they are applied to, recognising the fact that in some cases direct access to a base class's implementation details is needed. To put it otherwise: a mixin is applicable to a class, if this class provides the necessary private attributes for the implementation of the mixin, but not all mixins that are applicable to a class need access to the private attributes of that class. Essentially mixins are differentiated by how much of the implementation details of the base class are visible to them.

```
nested mixin-attributes
--- BaseClass attributes ---
 MC is mixin
   defining
     properToC       --- e.g. an instance variable

     PMC is mixin
       defining
         --- properToC is visible here
     endmixin --- PMC ---
 endmixin  --- MC ---

 NotPMC is mixin
   defining
     --- properToC is NOT visible here
 endmixin --- NotPMC ---

C is class obtained by MC extension of BaseClass
PC is class obtained by PMC extension of C
NotPC is class obtained by NotPMC extension of C

--- both PC and NotPC are subclasses of C, but the visibility of C
attributes is different for both subclasses ---
```

The degree to which a mixin has access to the implementation details of a class is solely based on whether a mixin is a proper or an inherited attribute of a certain class. Consider a class C that was constructed by application of a mixin MC to some base class. There are two sorts of mixins that can be used to create subclasses of C: mixins that are proper attributes of C (in the example: PMC defined in the mixin MC) and mixins that are inherited (in the example: NotPMC). A mixin that is a proper attribute of the class C, has, by definition, access to the proper private attributes of that class C, and to the same private attributes that the mixin MC has access to. A mixin that is inherited has no access to the proper private attributes of the class it is applied to. Note that this leads in a natural way to, and is consistent with, nested mixins. For a mixin to be a proper attribute of the class C, it must be defined in (and consequently nested in) the mixin MC, and according to lexical scope rules has access to the names of the attributes defined in the mixin MC.

So, the amount of detail in which a subclass depends on the implementation aspects of its superclass is determined by the relative nesting of the mixins used to create the sub- and superclass. Not only are the instance variables defined in a mixin visible for the method declarations in that mixin, but also those of the surrounding mixins. A mixin can be made more or less abstract according to its position in the inheritance tree.

Complete abstraction in mixins can be obtained by not nesting them in other mixins, resulting in a totally encapsulated form of inheritance, as is proposed in [Snyder87]. It is called abstract because the resulting subclass must use message passing to access inherited private attributes.

If abstraction is not required, exposure of inherited private attributes in mixins can be obtained by making the nesting and inheritance hierarchy the same, i.e. by nesting the mixin provided to create the subclass in the mixin provided to create the superclass. Consequently it is very easy to construct Smalltalk-like inheritance using this approach.

Of course, combinations between full and no nesting at all are possible. The higher in the hierarchy a mixin is defined, the more objects that can be extended with this mixin, the more abstract the mixin has to be. Thanks to the fact that the applicability of a mixin is restricted to the classes where it is defined on, it is always guaranteed that the resulting object is consistent in the sense that components referred to through nesting always exist.

It is obvious that the possibility to nest mixins provides the user with a very powerful tool for building inheritance hierarchies. Instead of promoting one single strategy for handling encapsulation between inheriting clients (instance variables are either always or never visible in subclasses) one allows the user to build his own application specific encapsulation mechanism.

4.4 Design of Agora

4.4.1 Introduction

Agora is solidly rooted in the object-oriented paradigm. Agora is a *prototype*-based language [Ungar&Smith87] featuring a generalised *mixin* [Bracha&Cook90][Steyaert&al.93] approach to inheritance. The extension of prototypical objects through the application of mixins is embedded in the lexical scoping of identifiers in Agora [Buhr&Zarnke88, Madsen89]. Consistent *reification* [Smith84] is the approach used for capturing features such as name binding, deferred evaluation, self reference etc.

Any of these features, taken by themselves, do not constitute innovations. Agora is innovative in the way that these features are bound together in one consistent language framework. Mixins are specified as methods and mixins are applied in the same way that ordinary messages are sent. Reification is equally structured as message passing: reifiers are nothing but methods defined within the bodies of abstract grammar prototypes. Whereas in most programming languages, e.g. inheritance and name-binding mechanisms are expressed in structures that differ fundamentally from ordinary programming structures, Agora requires but one

programming paradigm for all components of the system.

4.4.2 Agora Syntax

Agora syntax resembles Smalltalk syntax in its *message expressions*. The different kinds of message expressions are: unary, operator and keyword messages. Message expressions can be imperative (statements) or functional (expressions). For clarity, in the text keywords and operators are printed in italics.

```
aString size                      unary message
aString1 + aString2               operator message
aString at:index put:aChar        keyword message
```

A second category of message-expressions is the category of *receiverless messages*. Receiverless messages have the same syntax as the pattern part of message expressions. Their principal usage is to invoke messages on an implicit receiver, for example to invoke private methods; they will also be used as part of other syntactic structures where message patterns need to be manipulated (i.e. method declarations).

```
size                 receiverless unary message (identifier)
+ aString2           receiverless operator message
at:index put:aChar   receiverless keyword message
```

A third category of message expressions is the category of *reify messages*[5]. Reify messages have the same syntax as message expressions, and respectively receiverless message expressions except for their bold-styled keywords/operators. Reify expressions (i.e. reify messages, receiverless reify messages, and reify aggregates as can be found in the next paragraph) collect all "special" language constructs in one uniform syntax (comparable to Lisp special forms). They correspond to syntactical constructs such as variable declarations, pseudo variables, control structures and many other constructs used in a more conventional programming language. Reify expressions help in keeping Agora syntax as small as possible. Special attention must be paid to the precedence rules. Reify expressions have, as a group, lower precedence than regular message expressions. In each category unary messages have highest precedence, keyword messages have lowest precedence.

```
self             receiverless unary reifier
a <> 3           operator message reifier
a define: 3      keyword message reifier
```

Message expressions can be grouped to form blocks. Blocks are an example of the third kind of reify expressions, i.e. reify expression whereby the delimiters are the variable part of the syntax (it is not necessary to have bold styled delimiters since delimiters are not used for any other purpose). Although other expression aggregates are imaginable, in this text only blocks will be considered.

```
[c1 define: Complex clone ;
 c2 define: Complex clone ;
 c1 real:3 imag:4 ;
 c2 <- c1]
```

5 In a reflective variant of Agora it is possible to add reify methods, hence the name. Reify methods are executed 'at the level of the interpreter' in which all interpreter parameters (context and such) are 'reified'.

The following shows the concrete grammar of Agora in BNF form. Terminals are included in quote (“”) symbols. Production rules have the form: … -> …, where the left-hand side is always a non terminal. In the right-hand side of a production vertical bars (|) are used to indicate alternatives, square brackets ([]) to indicate optional parts, and curly brackets ({}) to indicate zero or more repetitions.

Agora Concrete Grammar

```
Expression                -> ReifierMessage | ReifierPattern | Pattern

ReifierPattern            -> ReifierUnaryPattern | ReifierOperatorPattern
                             | ReifierKeywordPattern
Pattern                   -> UnaryPattern | OperatorPattern | KeywordPattern

ReifierMessage            -> ReifierOperation { ReifierKeywordPattern }
ReifierKeywordPattern     -> BoldKeyword ReifierOperation
ReifierOperation          -> ReifierUnary [ ReifierOperatorPattern ]
ReifierOperatorPattern    -> BoldOperator ReifierOperation
ReifierUnary              -> Message { ReifierUnaryPattern }
ReifierUnaryPattern       -> BoldIdentifier

Message                   -> Operation { KeywordPattern }
KeywordPattern            -> Keyword Operation
Operation                 -> Unary [ OperatorPattern ]
OperatorPattern           -> Operator Operation
Unary                     -> Factor { UnaryPattern }
UnaryPattern              -> Identifier

Factor                    -> Literal | Aggregate | "(" Expression ")"
Aggregate                 -> LeftAggregateSymbol
                                   [ Expression { ";" Expression } ]
                             RightAggregateSymbol

Identifier                -> Character { CharacterOrDigit }
BoldIdentifier            -> BoldCharacter { BoldCharacterOrDigit }
Operator                  -> OperatorSymbol [ OperatorSymbol ]
BoldOperator              -> BoldOperatorSymbol [ BoldOperatorSymbol ]
Keyword                   -> Identifier ":"
BoldKeyword               -> BoldIdentifier ":"

Literal                   -> StringLiteral | RealLiteral |
                                   IntegerLiteral | CharacterLiteral

Character                 -> "a" | "b" | … | "z"
Digit                     -> "0" | "1" | … | "9"
CharacterOrDigit          -> Character | Digit
BoldCharacter             -> "a" | "b" | … | "z"
BoldDigit                 -> "0" | "1" | … | "9"
BoldCharacterOrDigit      -> BoldCharacter | BoldDigit
OperatorSymbol            -> ">" | "<" | "|" |"\" | "*" | "+" | …
BoldOperatorSymbol        -> ">" | "<" | "|" |"\" | "*" | "+" | …
LeftAggregateSymbol       -> "[" | "{"
RightAggregateSymbol      -> "]" | "}"
```

4.4.3 Standard Agora Reifiers

Agora' s syntax consists of two layers. The above given syntax only specifies the generic or variable layer. Reifiers form the variable part of Agora' s syntax. Much of the design of Agora is found in the exact list of reifiers that can be used by the programmer. In the section on reflection we will show how the set of reifiers can be extended. For the time being we will need a standard set of reifiers. We will not try to be complete in this list. The idea is to define a vanilla variant of Agora, that can be used in a subsequent section to be extended with more elaborate constructions.

Variable Slots
Variables, be it instance variables or local variables, are declared with a variant of the **`define`** reifier. Its three standard variants are listed below.

```
x define                variable declaration reifier
x define: 3             same, but with initial value
d <> Dictionary         same, but with clone of initial value
```

Agora is a slot based language. The value of its variables must be accessed and modified through message passing. The receiver, however, can, in case of an access to a private variable, be left implicit. So, a private variable 'x' is accessed via the receiverless unary pattern 'x' and its value is set to e.g. 3 with the receiverless keyword message 'x:3'. An equivalent assignment can be used also:

```
x                       variable access
x:3                     variable assignment
x <- 3                  assignment reifier; equivalent to the above
```

A note should be made here. Since receiverless messages are used to access variables, they must, in contrast with other messages, obey the lexical scoping when being looked up (more on this in the section on nested mixin methods).

Control Structures
Due to Agora's nature control structures can be introduced in two ways. The first is as user defined control structures based on a notion comparable to first class 'blocks' (e.g. Smalltalk) or closures (e.g. Scheme). A second way is by the definition of reifiers that implement a fixed set of control structures. The former will be discussed as an extension to Agora in a later section. For the time being, examples of the standard control structure reifiers are listed below.

```
a = b ifTrue:[a <- 3 ]
      ifFalse:[a <- 4 ]                 if reifier

a < b whileTrue:[a <- a + 1 ]           while reifier
```

More importantly Agora has a self pseudo variable reifier and a reifier to invoke parent operations. Both reifiers are receiverless.

```
self                    self pseudo variable reifier

super:(at:3 put:5)      super message invocation reifier; invokes the
                        parent's 'at:put:' method
```

The self pseudo variable reifier, when evaluated, returns the current receiver. The super reifier delegates its message argument (which must be receiverless) to the parent object. The super reifier has a bit of an unusual form. Rather than having a super pseudo variable, invocation of parent operations looks more like a control structure. The difference between a super pseudo variable and a parent invocation control structure is subtle, however.

Note also the difference between receiverless messages and messages to the self pseudo variable. Both are directed to different parts of the receiver. The former is used to invoke private operations (and in this variant of Agora only identifier lookup is supported), the latter to invoke public operations. So '**self** x' and 'x' generally have a different result.

Mixins & Methods
The following is an example *mixin method*. This method adds a colour attribute and its access methods to the object it is sent to. In all the examples that follow, mixin definitions standing free in the text (top-level mixins), are presumed to be defined on the root object called Object. So, in the example below, the root object Object is extended with colour attributes by invoking its *addColour* mixin (sending the message *addColour* to it). The resulting ColourObject object is an inheritor of Object.

```
[ addColour Mixin:
    [ colour define ;
      colour:newColour Method:[colour <- newColour] ;
      colour Method: colour
    ] ;
 ColourObject define: Object addColour
]
```

Object is extended with an instance variable "colour" and two methods: an imperative method *colour:* and a functional method *colour*. The body of a method can be either a block or, for example for functional methods, a single expression. To the left of the **Method:** reifier keyword is the pattern to invoke the method; it has the form of an ordinary message expression, except that it has no receiver and the arguments to the keywords are replaced by the names of the formal arguments.

Although we often used the term 'class' in the explanations of the above examples, Agora is at its heart an object-based language with object-based inheritance. In fact in 'standard' Agora no notion of classes exists. Agora's mixin method are applicable to objects. An object is extended by invoking one of its mixin methods.

In Agora, mixins and methods are very similar. Methods are to be considered as executing in a temporary, local extension of the receiver object. An explicit notion of closures, or method activation can be avoided due to the object-based nature of Agora. The difference between methods and mixins is that the one extends the receiver object only temporarily and the other extends the receiver object more permanently. Due to this similarity, arguments and local variables of methods can be defined and accessed in a totally similar way as instance variables. This opens the question of method declarations local to method declarations. Agora's design restricts all declarations within method bodies, to variable declarations !

Object Creation
Agora objects are created by taking copies (clones) of existing objects. In its most elementary form this takes the form of a clone reifier. A more elaborate cloning method will be discussed in a later section. Whereas in class-based languages one talks about classes and instances, in object-based languages one speaks about *prototypes* and *clones*. By convention the names of objects that are consistently used as prototypes start with an uppercase letter.

```
p1 <- (P clone)    assign a copy of P to p1

p1 <> P            declare a new variable p1 with initial value
                   a copy of P
```

Another way to create new objects is by application of mixins to objects. Since mixins can be applied to objects (rather than classes), different independent extensions of one and the same object can be made.

In the following example two extensions of a same, shared parent (john) are

made. These extensions implement different views on the same object, in this case to resolve the ' card number' name conflict. This example encodes, in Agora, the ' multiple viewpoint' example from the section on multiple inheritance.

```
MakePerson Mixin:
  [ name define ;
    name:newName Method:[name <- newName] ;
    name Method:name ] ;

MakeSportsman Mixin:
  [ cardnumber define ;
    number:newNr Method:[cardnumber <- newNr ] ;
    number Method:cardnumber ] ;

MakeStudent Mixin:
  [ cardnumber define ;
    number:newNr Method:[cardnumber <- newNr ] ;
    number Method:number ] ;

Person define: Object MakePerson ;

john <> Person ;
johnAsASportsman define ;
johnAsAStudent define ;

johnAsASportsman <- john MakeSportsman ;
johnAsAStudent <- john MakeStudent ;

john name:'john' ;
johnAsASportsman name                    ----> 'john'
johnAsAStudent name                      ----> 'john'

johnAsASportsman number:4 ;
johnAsAStudent number:5 ;
johnAsASportsman number                  ----> 4
johnAsAStudent number                    ----> 5
```

An Example of Mixin Nesting in Agora

As said before, a mixin is either nested in another mixin, or not nested at all, to control the amount of detail to which an inheritor depends on the implementation of its heir. This is illustrated in the two following examples.

The general idea in the first example is to have turtles which are, in our case, a sort of point that can be moved in a "turtle-like" way (no drawing is involved at the moment). The essence is that a turtle user does not manipulate the location and heading of the turtle directly but uses the home/turn/forward protocol.

```
MakeTurtle Mixin:
  [ location define: Point rho:0 theta:0*pi ;
    heading define: 0*pi ;
    position Method: location ;
    home Method: [ location <- Point rho:0 theta:0*pi; heading <- 0*pi ];
    turn:turn Method: [heading <- heading + turn] ;
    forward:distance Method:
      [ location <- location + (Point rho:distance theta:heading) ] ;

    MakeBounded Mixin:
      [ bound define: Circle m:location r:infinite ;
        home Method:
            [ super:home ; bound <- Circle m:location r:infinite ] ;
        newBound:maxRho Method: [ bound <- Circle m:location r:maxRho ] ;
        forward:distance Method:
          [ newLocation define ;
            newLocation <- location + (Point rho:distance theta:heading) ;
            (newLocation - (bound center)) rho > bound r
                  ifTrue:
                    [super:(forward:
```

```
                              (((LineSeg p1:location p2:newLocation)
                                   intersect:bound)  - location)rho )]
                  ifFalse: [ super:(forward:distance)] ] ] ] ;
Turtle define: Object MakeTurtle ;
BoundedTurtle define: Turtle MakeBounded ;
aBoundedTurtle define: BoundedTurtle clone ;
aTurtle define: Turtle clone ;
aBoundedTurtle forward:1 ;
aTurtle forward:3
```

Once the turtle is defined, the next step is to create an inheritor that puts boundaries on the movements of the turtle. In the example turtles are restricted to move within the bounds of a circle. For this purpose the `forward` method is overridden in the inheritor that implements this boundary checking. This overridden `forward` method uses direct access to the turtle instance variables `location` and `heading` in its implementation.

For the construction of the prototypes `Turtle` and `BoundedTurtle`, two mixins, `MakeTurtle` and `MakeBounded` respectively, are defined. To make sure that the prototype `BoundedTurtle` inherits from prototype `Turtle` in a non-encapsulated way, the `MakeBounded` mixin is nested in the `MakeTurtle` mixin. Notice that, since the `MakeBounded` mixin is defined only for `Turtle`, it can only be used to extend the `Turtle` prototype and its inheritors. Not only is it impossible to extend the root object `Object` with the `MakeBounded` mixin since it is not defined for the root object but also since `Object` does not define the `location/heading` instance variables that are required by the `MakeBounded` mixin.

Each clone of `Turtle` and each clone of `BoundedTurtle` has its own set of `location/heading` instance variables. Furthermore, if in the `MakeBounded` mixin an instance variable were to be declared with a name that collides with a name in the `MakeTurtle` mixin (e.g. an instance variable with the name “`heading`”), then each `BoundedTurtle` would have two instance variables with this name. One instance variable would only be visible from within methods defined in the `MakeTurtle` mixin, the other instance variable would only be visible from within methods defined in the `MakeBounded` mixin. There is a “hole in the scope” of the instance variable defined in the `MakeTurtle` mixin. So, there is no merging going on for instance variables with equal names, neither is it an error to have an instance variable with the same name in an inheritor (as is the case in Smalltalk). Notice that identifier lookup is a static operation: the instance variable that is referred to in an expression can be deduced from looking at the nested structure of the program. No dynamic lookup strategies are applied. Similar observations can be made for non-nested mixins. Encapsulating the names of instance variables in this way is an important aid in enhancing the potential for mixin composition. This is all the more important if mixins are used to create/emulate multiple inheritance hierarchies.

Thus, if a mixin is nested in another mixin, objects created by the innermost mixin are always (not necessarily direct) inheritors of objects defined by the outermost mixin. However, the reverse statement is not always true. Nesting is not a requirement for inheriting.

```
MakeDrawingTurtle Mixin:
  [ penDown define: true ;
    togglePen Method: [penDown <- penDown not] ;
    forward:distance Method:
      [ newPosition define ;
        oldPosition define: self position ;
        super:(forward:distance);
        newPosition <- self position ;
        penDown ifTrue:[… draw line from old position to new position …]];
```

```
MakeDashed Mixin:
  [ dashSize define: 1 ;
    setDashSize:newSize Method: [dashSize <- newSize] ;
    forward:distance Method:
      [ penDown
          ifTrue:
            [ 1 to: (distance div: dashSize)
                do: [ super:(forward: dashSize); self togglePen ] ;
              super:(forward: (distance mod: dashSize)) ;
              penDown <- true ]
          ifFalse: [ super:(forward:distance) ] ]
    ]
  ] ;
DrawingTurtle define: Turtle MakeDrawingTurtle ;
DashedDrawingTurtle define: DrawingTurtle MakeDashed ;
```

The goal in the above example is to extend the `Turtle` object so that it draws, or does not draw (depending on the status of the pen), on the screen where the turtle is heading. The drawing capabilities can be added fairly independently of the implementation of the turtle. Once again the `forward` method is overridden. But all that is needed in the implementation of the overridden `forward` method is the old `forward` method and a method that returns the current location of the turtle. Notice that, even though an accessor method to the location of the turtle must now be made public, the `heading` instance variable is still encapsulated. The `MakeDrawingTurtle` mixin that implements this extension does not have to be nested in the `Turtle` mixin, resulting in a `MakeDrawingTurtle` mixin that can be applied to other sorts of turtle objects that respect the `forward`/`position` protocol.

Earlier on we said that the `MakeBounded` mixin could only be applied to the `Turtle` object and its inheritors. `DrawingTurtle` is such an inheritor. We now have two ways to create bounded drawing turtles. On the one hand, by applying the `MakeDrawingTurtle` to a `BoundedTurtle` (`DrawingBoundedTurtle` **`define:`** `BoundedTurtle` *`MakeDrawingTurtle`*), on the other hand, by applying the `MakeBounded` mixin to a `DrawingTurtle` (`BoundedDrawingTurtle` **`define:`** `DrawingTurtle` *`MakeBounded`*). In this example both results are the same; the `forward` method in the `MakeDrawingTurtle` mixin is such that it only draws a line up to the position where the turtle has moved, even if it moved a shorter distance than was intended.

It is important to note that the order of mixin application has no effect on the exposure of implementation details of the applied mixins to each other. The order in which the mixins *`MakeDrawingTurtle`* and *`MakeBounded`* are applied has no effect on the respective exposure of implementation details of the turtle base object to the inheriting clients `DrawingBoundedTurtle` or `BoundedDrawingTurtle`. The `makeBounded` and the `makeDrawingTurtle` cannot access each other's encapsulated part (independently of which mixin is applied first), and in both cases only the `MakeBounded` mixin has access to the `turtle` object's implementation details.

It is coincidental in the example that we can choose in which order the mixins *`MakeDrawingTurtle`* and *`MakeBounded`* are applied, and that both results exhibit the same *behaviour*. Not all mixins are commutatively applicable. Therefore the order of mixin application must not have an effect on the exposure of implementation details of the applied mixins to each other.

Mixin Methods
The fact that mixin application is realised by mere message passing, and that mixins can be applied dynamically has clear advantages. In this section we will give a simple example of dynamic mixin application, and an example of late binding of mixins.

Mixins can be combined to form chains of mixins that can be applied as a whole. Chains of mixins are useful to abstract over the construction of complex object hierarchies. A simple example is given making use of the Turtle objects shown earlier on. The idea is to construct different sorts of dashed drawing turtles without having to explicitly create a simple drawing variant, and a dashed drawing variant for each sort of turtle. This is, of course, the simplest example of how dynamic mixin application is used to abstract over the construction of an inheritance hierarchy.

```
MakeDashedDrawing Method: self MakeDrawingTurtle MakeDashed ;

DashedDrawingTurtle define: Turtle MakeDashedDrawing ;
DashedDrawingBoundedTurtle define: BoundedTurtle MakeDashedDrawing
```

In the example a chain of mixins is constructed as a method that successively applies two mixins. A declarative operator (as in [Bracha&Cook90]) to construct chains of mixins (or even entire hierarchies) could prove useful.

To illustrate the use of late binding of mixin attributes, consider a program in which two freely interchangeable implementations of point objects exist; one implementation based on polar co-ordinates and one based on cartesian co-ordinates. In some part of the program, points must be *locally* (for this part of the program only) restricted to bounded points, i.e. points that can not move outside given bounds. To do this, every point must have a mixin attribute to add methods and instance variables that implement this restriction. Each of the point implementations can have its own version of this mixin in order to take advantage of the particular point representation. For example, the mixin defined on polar co-ordinate represented points, can store its bounding points in polar co-ordinates in order to avoid excessive representation transformations. An anonymous point object (one of which we don’t know whether it is a polar or a cartesian point; typically a parameter of a generic class) can now be asked to extend itself to a bounded point by selecting the bounds mixin by name. The appropriate version will be taken.

```
MakeCartesianPoint Mixin:
  [ x define: 0 ; y define: 0 ;
    move:aPoint Method: … ;

    MakeBounded Mixin:
      [ bound define: CartesianBasedBounds clone;
        move:aPoint Method: …
      ]
  ] ;

MakePolarPoint Mixin:
  [ rho define: 0 ; theta define: 0*pi ;
    move:aPoint Method: … ;

    MakeBounded Mixin:
      [ bound define: PolarBasedBounds clone;
        move:aPoint Method: …
      ]
  ] ;

--- suppose Point is bound to either a Polar or Cartesian Point
BoundedPoint define: Point makeBounded
```

The highly expressive combination of nested mixin methods and object-based programming (apart from the reflective architecture, which will be discussed later on) is what differentiates this variant of Agora from most object-oriented programming languages.

4.5 The Agora Framework

The experiences, techniques and terminology acquired in building the framework for Simple can now be put to use in building a framework for a full-fledged object-oriented programming language. The standard variant of Agora is used for this purpose.

We will see that the kernel of the framework — i.e. expression, object, pattern and slot classes —, is shared between Simple and Agora. This should come as no surprise. The differences between Agora and Simple are important however. In Simple the encapsulation operator played an important role. In Agora this operator is abandoned[6,7], and a more standard way of argument passing is employed. Simple has no inheritance, inheritance is an important aspect of Agora. Agora is entirely built around generic expressions, called reify expressions. So, Agora is not a specialisation of Simple, rather Simple and Agora are both specialisations of the same framework.

An important aspect of Agora is how much the representation of objects is affected by the inheritance structure. Especially for object-based inheritance, one could be tempted to encode the inheritance structure by means of delegation. An example is given below.

```
class ShouldBeDelegatedToObject
  methods
    abstract delegate:pattern receiver:receiver
      result ShouldBeDelegatedToObject
endclass

class ObjectWithParent extends ShouldBeDelegatedToObject
  instance variables
    slots: Set(Slot)
    parent: ShouldBeDelegatedToObject
  methods
    concrete delegate:pattern receiver:receiver
        result ShouldBeDelegatedToObject
      slot := slots findSlot:pattern
      if slot found
         then [… evaluate the body part of the slot … ]
         else [^parent delegate:pattern receiver:receiver]
endclass
```

6 Incorporating this encapsulation operator in Agora is a non-trivial task due to interference with Agora’s inheritance. We think, however that it is an important lack in Agora since this issue is related to the issue of virtual private attributes (that are also lacking in Agora).

7 Note that this does not mean that encapsulation is abandoned. Agora’s objects still make a distinction between encapsulated and public attributes.

Here objects (i.e. objects of class `ObjectWithParent`) that have a parent object, will, upon reception of a message, look up the corresponding slot in their own set of slots and delegate the message if the right slot is not found. For this to work, all objects must be represented as instances of concrete subclasses of the abstract `ShouldBeDelegatedToObject` class, that specifies the nature of delegation.

Such an implementation can be discarded as being too operational. In our analysis of object-oriented programming languages, we discarded those languages that have an explicit delegation operator. So, when introducing an explicit delegation operation in the implementation of objects, again, at the implementation level we have a finer view to distinguish objects than is possible at the programming language level. Thus, such an implementation is not ‘fully abstract’. We will show that it *is* possible, even in the presence of inheritance, to maintain our abstract representation of objects.

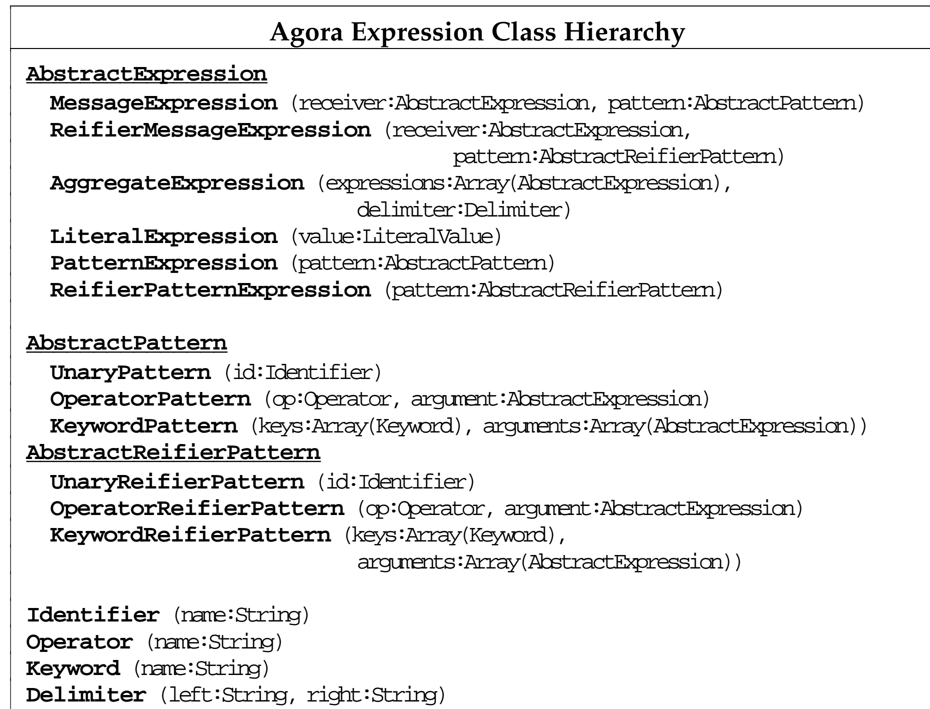
4.5.1 Abstract Grammar, Expression Objects and Reifier Methods

Agora takes the notion of generic expressions to the extreme. Its syntax (see previous section) is built up solely with message passing expressions and generic expressions. Generic expressions in their own right are formulated in the form of message expressions. Message passing forms the kernel of Agora.

Agora Abstract Grammar

```
Non-Terminal          = { Aggregate, Message, ReifierMessage, UnaryPattern,
                          OperatorPattern, KeywordPattern, ReifierUnaryPattern,
                          ReifierOperatorPattern, ReifierKeywordPattern }

Terminal              = { Identifier, Operator, Keyword, Literal, Delimiter }

Root                  = ExpressionSet

--- expansion sets ---
ExpressionSet       = { Literal } + { Aggregate }
                      + { Message } + { ReifierMessage }
                      + PatternSet + ReifierPatternSet
PatternSet          = { UnaryPattern } + { OperatorPattern } + { KeywordPattern }
ReifierPatternSet   = { ReifierUnaryPattern } + { ReifierOperatorPattern }
                      + { ReifierKeywordPattern }
IdentifierSet       = { Identifier }
OperatorSet         = { Operator }
KeywordSet          = { Keyword }
DelimiterSet        = { Delimiter }

--- productions ---
Message         -> ExpressionSet x PatternSet
ReifierMessage  -> ExpressionSet x ReifierPatternSet

ReifierKeywordPattern       -> (KeywordSet x ExpressionSet)+
ReifierOperatorPattern      -> OperatorSet x ExpressionSet
ReifierUnaryPattern         -> IdentifierSet

KeywordPattern       -> (KeywordSet x ExpressionSet)+
OperatorPattern      -> OperatorSet x ExpressionSet
UnaryPattern         -> IdentifierSet

Aggregate   -> DelimiterSet x ExpressionSet*
```

The class hierarchy that implements Agora’s abstract grammar can be found below.



Agora Expression Class Hierarchy

```
AbstractExpression
  MessageExpression (receiver:AbstractExpression, pattern:AbstractPattern)
  ReifierMessageExpression (receiver:AbstractExpression,
                                pattern:AbstractReifierPattern)
  AggregateExpression (expressions:Array(AbstractExpression),
                           delimiter:Delimiter)
  LiteralExpression (value:LiteralValue)
  PatternExpression (pattern:AbstractPattern)
  ReifierPatternExpression (pattern:AbstractReifierPattern)

AbstractPattern
  UnaryPattern (id:Identifier)
  OperatorPattern (op:Operator, argument:AbstractExpression)
  KeywordPattern (keys:Array(Keyword), arguments:Array(AbstractExpression))
AbstractReifierPattern
  UnaryReifierPattern (id:Identifier)
  OperatorReifierPattern (op:Operator, argument:AbstractExpression)
  KeywordReifierPattern (keys:Array(Keyword),
                           arguments:Array(AbstractExpression))

Identifier (name:String)
Operator (name:String)
Keyword (name:String)
Delimiter (left:String, right:String)
```

Three forms of generic expressions exist: reifier messages, reifier patterns and aggregate expressions; instances of the respective expression classes ReifierMessageExpression, R eifierPatternExpression, A ggregateExpression. Their evaluation functions are responsible for dispatching to an appropriate evaluation method. We will not go into the technical details of this mapping. An evaluation method that is invoked due to a reifier message is called a reifier method. We will not go into the details of how reifier methods are represented. We will use a pseudo code instead for introducing new reifier methods (and we will see in the section on reflection that this pseudo code is not to far away from reality).

A *reifier method* is declared as a special sort of method in an expression or pattern class. Since it defines an evaluation method, a reifier method has, in addition to its apparent "subexpression" arguments, a context argument (hidden at the call site of the reifier). This is reflected in the pseudo code by the using clause. An example will illustrate this. The example is the declaration of an assignment reifier method. A reifier method is invoked by an instance of ReifierMessageExpression.

```
class UnaryPattern extends AbstractPattern
  methods
    reifier <- (rightHand:AbstractExpression)
        using (context:StandardContext)
      … evaluate the rightHand expression, and store the value
      … in the context, using the receiver as key
endclass

usage in Agora:
a <- 3
```

In a similar vein it is possible to declare reifier classes. A *reifier class* is created and its evaluation method is invoked, every time an instance of the class ReifierPatternExpression is evaluated. The dispatcher function for reifier patterns creates an instance of the correct reifier class and then sends an evaluation message to this instance. A reifier class is declared as a special sort of

class. It has an associated pattern (i.e. the name of the reifier pattern that creates it). The associated pattern automatically contains the declaration of the instance variables for subexpressions.

```
reifierclass SelfExpression
    pattern self
    extends AbstractExpression
  methods
    concrete eval:(context:StandardContext)
      … return the current receiver from the context
endclass

usage in Agora:
3 + self
```

Generic aggregate expressions (i.e. generic compound expressions with a variable number of subexpressions) are a straightforward variation of reifier classes. The pattern of an aggregate reifier class contains the delimiters for the aggregate, and a declaration of an “instance variable” for the sequence of subexpressions.

```
reifierclass BlockExpression
    pattern [ exps:Sequence(AbstractExpression) ]
    extends AbstractExpression
  methods
    concrete eval:(context:StandardContext)
      … evaluate each of the subexpressions in the context
endclass

usage in Agora:
[ … ; … ; … ]
```

4.5.2 Message Passing

Like in the implementation of the calculus, the implementation of message expressions plays an eminent role. Unlike the calculus, in the implementation of message passing in Agora, parameter passing must be dealt with. Still, the implementation of message expressions can be done in a way that is independent of evaluation categories.

Agora Message Passing

```
class MessageExpression extends AbstractExpression
  instance variables
   receiver:AbstractExpression,
   pattern:AbstractPattern
  methods
   concrete eval:(context:StandarContext+)
       result AbstractMetaObject
     local variables arguments:ArgumentList

     for each argument in pattern do
        arguments add:(argument eval:(context asFunctionalContext))

     ^(receiver eval:(context asFunctionalContext))
         send:(pattern asCategory:context)
         client:(StandardClient arguments:arguments)
endclass
```

In the description of message passing three things should be noted. One is that arguments are stored in the client object. As announced earlier client objects are used to carry information from the sender object to the receiver. Notice that Agora's client objects are totally unrelated to Simple's client objects due to the lack of an explicit encapsulation operator in Agora.

Agora Standard Client

```
class StandardClient
  public instance variables
   arguments:ArgumentList
endclass
```

Secondly, and more importantly, note the need for casting the context in the course of evaluating expressions. In the above case, contexts are cast to ensure that message expressions can be evaluated in all possible evaluation categories. When evaluating a message expression, the context in which the entire message expression is evaluated, and the context in which the receiver and argument expressions are evaluated can not be (exactly) the same since the receiver and arguments always have to be evaluated in a functional evaluation category. On the other hand, the context in which the receiver and arguments of a message expression are evaluated must be derived from the context in which the entire message expression is evaluated. So the context in which message expressions are evaluated is cast to a functional context for the evaluation of receiver and arguments. The protocol of context objects is adapted accordingly.

Agora Standard Context (Extract)

```
class StandardContext
  public instance variables
   …
  methods
   abstract asFunctionalContext result StandardContext
     … return an instance of functional context with the same
     … content
endclass
```

The need for casting contexts is not limited to message expressions. Due to Agora's extremely simple syntax, evaluation categories play an eminent role, and thereby also the need for expressing dependencies between evaluation categories.

Finally, a word is in order about the role of patterns in the implementation of Agora. Unlike patterns in the calculus, Agora patterns that are part of some program representation, include argument expressions. For this reason an uncoupling of message patterns that are part of a program representation, and patterns that are used at run-time is in order. The latter sort of patterns (instances of the class `AbstractCategoryPattern`) are mainly used as unique identifiers. Patterns that are part of a program representation are turned into patterns that can be used for message passing via the `asCategory` message, i.e. this transformation process must take evaluation categories into account.

Patterns Used in Expressions Versus Patterns Used in Messages

```
class AbstractPattern
  methods
   abstract asCategory:StandardContext+
     result AbstractCategoryPattern
endclass
```

```
class AbstractCategoryPattern
  methods
   abstract = AbstractCategoryPattern+ result Boolean
endclass
```

Apart from message expressions with an explicit receiver, Agora also has message expressions with an implicit receiver. These are the so called *receiverless message expressions*. They are represented as instances of the PatternExpression class. The implementation of this class is similar to the implementation of message expressions, except for the fact that the receiver is a predefined part of the context.

Evaluation of Receiverless Messages

```
class PatternExpression extends AbstractExpression
  instance variables
   pattern:AbstractPattern
  methods
   concrete eval:(context:StandaredContext+)
       result AbstractMetaObject
     local variables arguments:ArgumentList

     for each argument in pattern do
        arguments add:(argument eval:(context asFunctionalContext))

     ^(context privatePart)
         send:(pattern asCategory:context)
         client:(StandardClient arguments:arguments)
endclass
```

4.5.3 Mixin Application and Object Structures

Agora objects differ from the previously discussed objects, from the calculus, in different ways. Although the essential protocol of message passing is the same, Agora objects have a more complex internal structure. Furthermore, since Agora is essentially a language with side-effects, issues such as object equality and object cloning must be dealt with. Calculus objects are richer in one way, the encapsulation operator on objects, that plays such an eminent role in the calculus, is not present in Agora. We will see that internally, Agora objects will use an operator reminiscent of Simple' s encapsulation operator. The difference is that the latter is an encapsulation operator on objects, and the former is more comparable to an encapsulation operator on generator functions as discussed in the previous chapter.

A note should be made about the relation of the framework with (nested) mixin method inheritance. Mixin method inheritance is important for Agora, and it is this form of inheritance that will be discussed in the framework. The question then arises whether it is possible to implement other inheritance mechanisms in the framework. This question can not be answered with a convincing 'yes'. Clearly the fact that interaction with objects is limited to message passing (and not delegation for example) is a serious constraint in this context. Since with mixin methods, inheritance is based upon sending messages to objects this is no

problem. For other inheritance mechanisms such a strong encapsulation probably is a problem. Accordingly implementing such a mechanism in this framework will involve extending the framework outside its intended usage, i.e. it will involve extending the framework in a less reusable fashion.

4.5.4 Agora Internal Object Structure

As we already said before, there exists a plethora of different kinds of objects. Still, we have opted for, and briefly discussed the advantages of, an abstract object representation. In casu, objects that can be sent messages. All implementation details of objects remain hidden in the object representation. This must be equally true for the inheritance structure of objects. It should not become apparent in an objects representation whether it inherits (or should inherit) from another object.

Unlike Simple objects, Agora objects have a complex internal structure. On the one hand this structure must be hidden in the object representation, on the other hand, with the eye on extensibility, a complex, and ad hoc, object structure needs to be avoided.

A solution was found in encoding the object structure as a structure of finer grained internal objects that communicate with each other with (variants of) delegation. This solution is based on the fact that it is possible to mimic almost any inheritance structure with a delegation based system [Lieberman86]. As we will show in a moment, the notion of delegation is extended to take mixins and objects with structured private attributes into account.

The general idea is to delegate messages to objects in an explicitly given context. This context not only encodes how message passing must proceed, but also how the body of a slot must be interpreted once it has been found. For example, the context can have a field that stores the original receiver of the object for interpreting future “self expressions” (which is the original meaning of delegation); it can have a “parent” field that will either be used to further delegate a message if no slot is found in the current object that corresponds to the message, or it is used for interpreting “super expressions” in the evaluation of the slot if it is found; or it can have an “encapsulated part” field that is used, amongst other things, as the receiver of all receiverless messages (in casu identifier lookup) that occur in the evaluation of the body of the found slot.

This will give rise to a set of different (internal!) objects that can be flexibly combined through delegation. The subset used in the implementation of Agora is given below. It will be shown how this set of internal objects is used to implement mixin based inheritance. Abstractly all internal objects respond to a delegate message that has an extra delegation context argument. This latter argument encodes all extra delegation information.

Root of the Abstract Internal Object Classes

```
class AbstractInternalObject
  methods
    abstract  delegate:CategoryPattern
              client:StandardClient
              context:DelegationContext+
        result AbstractMetaObject
endclass
```

A delegation context is a context in which extra fields can be filled in. The two extra fields used in the delegation structure for the encoding of mixin methods in

Agora are a parent field and a private field.

While delegating a message the parent field contains the parent object of the object that receives the delegation request. In fact when an object that is not composed of subobjects receives a delegated message that it does not want to reply to, it can use the parent field to further delegate the message.

The private field is used in the encoding of objects that have encapsulated attributes. Again, an object that is not composed of subobjects and that accepts a delegated message can use the private part of the delegation context to evaluate its method bodies in.

Contexts Used in the Realisation of Mixin methods

```
class DelegationContext extends StandardContext
  methods
   …
   concrete parent:AbstractInternalObject result DelegationContext
     --- returns a copy of the context with a new parent field
   concrete private:AbstractInternalObject result DelegationContext
     --- returns a copy of the context with a new private field
   concrete noParent result DelegationContext
     --- returns a copy of the context with a empty parent field
   concrete noPrivate result DelegationContext
     --- returns a copy of the context with a empty private field
   …
endclass
```

```
class ParentContext extends DelegationContext
  public instance variables
   parent: AbstractInternalObject
endclass
```

```
class EncapsulatingContext extends DelegationContext
  public instance variables
   private: AbstractInternalObject
endclass
```

Agora's inheritance structures are constructed by means of instances of the following three internal object classes. The first class encodes objects that have a parent object. On reception of a delegated message, this message is forwarded to the object that contains the locally defined attributes (i.e. `thisPart`). It is forwarded such that the parent object (the `parentPart`) is recorded in the delegation context. The local part of the object will forward the message to the parent if it does not respond, itself, to the message.

Concrete Internal Object Classes: 1) Objects with a Parent

```
class ObjectWithParent extends AbstractInternalObject
  instance variables
   thisPart: AbstractInternalObject
   parentPart: AbstractInternalObject
  methods
   concrete delegate:pattern client:client context:(context:DelegationContext+)
       result AbstractMetaObject
     ^thisPart delegate:pattern
             client:client
             context:(context parent:parentPart)
endclass
```

The second class encodes objects that have encapsulated attributes. Similar to above a message is forwarded to the object that contains the public attributes

(the `publicPart`) in a delegation context that records the private part of the object. The selected attribute in the public part will be evaluated in a context that is built up with the private attributes found in the delegation context.

This class resembles the `CompoundObject` class from the implementation of Simple. The difference is that here attributes are not encapsulated into an object after the object was created by an explicit encapsulation operator, but rather they are declared encapsulated when the object is created. This difference has already been discussed in the section on encapsulation operators for objects in the previous chapter.

Concrete Internal Object Classes: 2) Objects with Encapsulated Attributes

```
class EncapsulatedObject extends AbstractInternalObject
  instance variables
   publicPart: AbstractInternalObject
   privatePart: AbstractInternalObject
  methods
   concrete delegate:pattern client:client context:(context:DelegationContext+)
      result AbstractMetaObject
     ^publicPart
        delegate:pattern
        client:client
        context:(context private:privatePart)
endclass
```

The last class encodes objects that have no further subobjects, but directly store slots. These objects are important in this discussion since they will use the information in the delegation context to respond to delegated messages. First of all they will use the parent field of the delegation context to further delegate a message when this message is not locally handled. Secondly, they combine all other information found in the client (i.e. the information coming from the sender), and the information from the delegation context. This combined information is used to evaluate selected slots in. The parent field of the delegation context, for example, will be used to interpret parent operation invocations.

Concrete Internal Object Classes: 3) Objects without Subobjects

```
class SimpleObject extends AbstractInternalObject
  instance variables
   thisPart:Sequence(Slot)
  methods
   concrete delegate:pattern client:client context:(context:ParentContext+)
      result AbstractMetaObject
     slot := thisPart findSlot:pattern
     if slot
       then ^slot valueIn:(context with:client)
       else ^(context parent) delegate:pattern
                              client:client
                              context:(context noParent)
endclass
```

The above internal objects can be combined in different ways. The corresponding delegation contexts must be combined likewise. Delegation contexts can be combined with an appropriate multiple inheritance mechanism. The exact details are not important. What is important is that the delegation context in an actual implementation will be a combination of the two above listed and all other delegation contexts defined further on in the text.

Agora's inheritance structure can be used to illustrate how the above internal objects can be combined in a useful way. In the construction of nested mixin methods, objects with parents are not only used to link all public parts of an object, but also to link all encapsulated parts of an object. This can best be illustrated with an example. Figure 4.11 shows the internal representation of the objects of the following Agora program. This figure is best interpreted from right to left. The wrapper objects, totally on the right will be explained in the next section.

```
[ MakeX Mixin:
    [ xiv define ;
      xm Method: [ xiv <- 3 ] ;

      MakeY Mixin:
      [ yiv define ;
        ym Method: [ xiv <- 4 ; yiv <- 8 ] ] ;

      MakeZ Mixin:
      [ ziv define ;
        zm Method: [ xiv <- 5 ; ziv <- 9 ] ] ] ;

  Root define: self ;
  X define: Root MakeX ;
  XY define: X MakeY ;
  XYZ define: Y MakeZ ]
```

Each mixin-application adds a set of public attributes and a set of private attributes to the receiving object. Correspondingly each mixin application results in the creation of a new 'layer' in the hierarchy of internal objects. All layers are linked by instances of `ObjectWithParent` (the linked chain of objects on the right in the figure). Within each layer the private attributes are associated to the public attributes with an encapsulated object. Finally, all encapsulated objects form a hierarchy built up, again, with objects of class `ObjectWithParent`. This hierarchy encodes the nesting structure of the mixins in the above program.

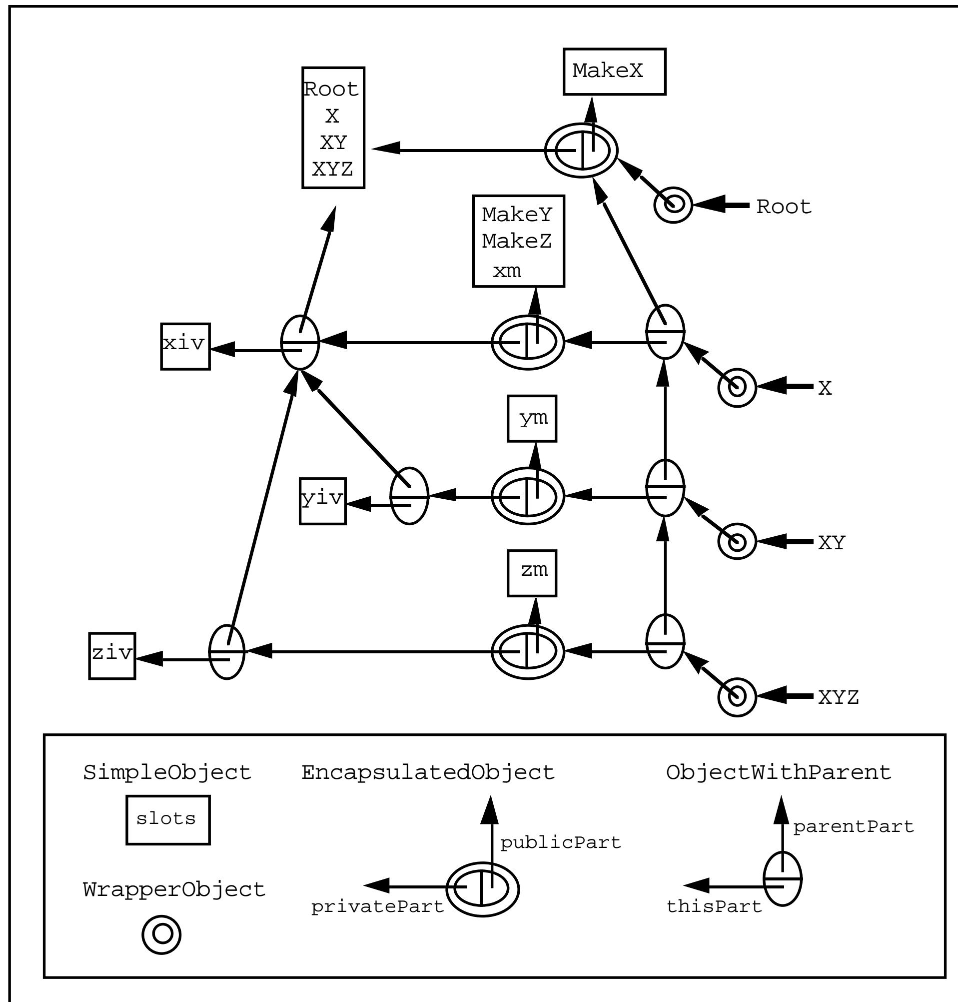


Figure 4.11

The way the scoping of nested mixins is dealt with is worth noting. The scoping of nested mixins is totally resolved with inheritance of encapsulated objects. In the figure both Z, and Y inherit in their encapsulated part from X, without inheriting from each other.

4.5.5 External Object Structures and Wrapper Objects

Internal objects can not be used directly in the evaluation of expressions. They do not hide enough details of their implementation, as opposed to meta-objects that only respond to the pure message passing protocol. Internal details of Agora internal objects are hidden by *wrapper objects*. Wrapper objects serve two purposes: 1) they act as holders of internal objects, thereby hiding their internal details 2) they are the explicit identity of objects. Wrapper objects are the only kind of meta-objects in use in the implementation of Agora. All variations on objects in Agora are due to variations in the internal object structures.

A wrapper object is essentially a forwarder of messages to its wrapped object. All accepted messages are delegated to the wrapped object, that is stored in the `delegate` instance variable. The wrapped object can be any constellation of internal object structures. Wrapper objects are responsible for generating recursive object structures, i.e. wrapper objects record themselves in the delegation context as receiver object. The `receiver` field of the delegation context can be used to interpret self expressions. The wrapped object is also put in the delegation

context. This has to do with the mechanism to extend a receiver object with mixins, as will be shown in the next section. A specialised delegation context with a `receiver` field and a `public` field is used in the implementation of wrapper objects.

Wrapper Objects

```
class WrapperObject extends AbstractMetaObject
  instance variables
   delegate: AbstractInternalObject
  methods
   concrete send:pattern client:client result AbstractMetaObject
     ^delegate delegate:pattern
               client:client
               context:(DelegationContext receiver:self
                                          public:delegate)
endclass
```

Agora Delegation Contexts that Record the Receiver

```
class DelegatingContext extends DelegationContext
  public instance variables
   receiver: AbstractMetaObject
   public: AbstractInternalObject
endclass
```

Wrapper objects encode the identity of Agora objects. As stated before, since Agora is an imperative programming language testing objects for identity is an important operation. Internally objects can have shared structures, but each object in Agora is represented by a unique wrapper. This allows dramatic changes to the internal object structures without changes to the identity of an object. An identity swap operation could easily be defined, for example. It suffices for two wrapper objects to swap their (private) wrapped objects. Other, more constructive, examples such as object reclassification can be implemented as easily. Note that it is this kind of variations that are the useful variations on wrapper objects.

For simplicity reasons we also expect, in the remainder of the text, that all internal objects have a wrap method. This wrap method puts a wrapper object around the receiving internal object, or in general hides the internal details of that object.

4.5.6 Extending Objects, Execution of Mixin Methods

Agora objects are extended by executing mixin methods. A mixin method is like an ordinary method except that its body is evaluated in a special mixin evaluation category (this is by the way, a good example of the usage of evaluation categories). Only block-expressions evaluate in the mixin evaluation category. They do so by extending the receiver with a new public and private part, and evaluating all the component expressions in this new receiver.

The evaluation of blocks in the mixin evaluation category is listed below. The mixin context used in this implementation indicates this evaluation category. This description may seem a bit involved, but what it actually does is adding an extra layer to the internal object structures to store public and private slots.

mixin method execution

```
reifierclass BlockExpression
   pattern [ exps:Sequence(AbstractExpression) ]
   extends AbstractExpression

  abstract class attributes
   ExtensibleSimpleObject ObjectWithParent EncapsulatedObject
  methods
   concrete eval:(context:MixinContext) result AbstractMetaObject
     local variables privateSlots publicSlots privatePart
             myPart publicPart newReceiver newContext
     privateSlots := ExtensibleSimpleObject new
     publicSlots := ExtensibleSimpleObject new
     privatePart := ObjectWithParent thisPart:privateSlots
                              parentPart:(context private)
     myPart := EncapsulatedObject publicPart:publicSlots
                              privatePart: privatePart
     publicPart := ObjectWithParent thisPart:myPart
                              parentPart:(context public)
     newReceiver := publicPart wrap
     newContext := (context asImperativeContext)
                    privateSlots:privateSlots
                    publicSlots:publicSlots
                    receiver:newReceiver
     for each exp in exps do
       exp eval:newContext
     ^newReceiver
endclass
```

Unlike Simple, Agora objects are not created by first collecting their slots and then creating an object with this collection of slots. In Agora slots are added to an object by means of declaration reifiers (e.g. method declaration, variable declaration, etc.). Declarations and other expressions may be mixed. For this reason, it must be possible to add new slots to the receiver after it has been created. This is possible with the following extension to the `SimpleObject` class of internal objects. With this extension slots can be added to this particular kind of internal objects. Also the mixin evaluation context must have two fields, each containing a reference: one to the object in which the public slots of the current extension are stored, and one to the object in which the private slots are stored.

Adding Slots to Agora Internal Objects

```
class ExtensibleSimpleObject extends SimpleObject
  methods
   concrete add:slot
     slots add:slot
endclass
```

Agora Delegation Contexts that Record the Public and Private Slots

```
class MixinContext extends DelegationContext
  public instance variables
   publicSlots: ExtensibleSimpleObject
   privateSlots: ExtensibleSimpleObject
endclass
```

Notice however that only the public and private slots of the *current* receiver object can be extended. The context only contains references to the extensible objects of the current receiver. All other objects can only be accessed via their wrapper, and they can only be extended by sending mixin messages (i.e. messages that result in the execution of a mixin method). Encapsulation is preserved !

4.5.7 Object Cloning

Object cloning plays an important role in Agora (as in any object-based programming language). In some form or another one can expect a cloning operation for objects. This might take the form of a simple clone method on objects, or, as we will see later on, more sophisticated constructions are possible.

Given the nature of how objects are represented internally, it is not evident how cloning must be implemented. First of all, internal objects use a non hierarchical sharing structure (e.g. in the realisation of nested scoping), this sharing structure must be preserved after cloning. Moreover, the cloning mechanism must be extensible, i.e. it must be possible to add new internal objects with their associated cloning strategy.

The cloning strategy for internal objects is made flexible through the introduction of *clone maps*. Clone maps (or a variant thereof) are typically used when copying circular pointer structures. They record the objects already copied, and associate each original object with its copy. It is obvious that this information can be used to copy objects while preserving circular or shared references.

Clone maps provide the classical operations for object cloning. A clone map can be asked to deep copy an object. Objects already present in the map will not be copied, rather the associated object in the map will take the place of the copy. A map can be asked to shallow copy an object. If this object is already in the map, the associated object is returned.

Clone Maps

```
class CloneMap
  methods
    concrete shallowClone:AbstractClonableInternalObject
        result AbstractClonableInternalObject
      … take a shallow clone of the argument if not present
      … in the map
    concrete deepClone:AbstractClonableInternalObject
        result AbstractClonableInternalObject
      … take a deep clone of the argument preserving the sharing
      … structure
      … all objects in the map will not be cloned
      … the map is extended with the newly copied objects
    concrete shallowNoClone:AbstractClonableInternalObject
      … extend the map with an identity association on the argument
    concrete deepNoClone:AbstractClonableInternalObject
      … extend the map with identity associations of all the direct
      … and indirect acquaintances of the argument
endclass
```

In our case clone maps are used to achieve a general and flexible object cloning mechanism. All clonable internal objects must provide three sorts of cloning operations. An object' s outline (i.e. acquaintances are not copied) is copied with the shallow copy operation. Deep cloning is used to copy an object' s acquaintances while respecting the clone map.

Abstract Class for Clonable Internal Object Classes

```
class AbstractClonableInternalObject extends AbstractInternalObject
  methods
   abstract shallowClone result AbstractClonableInternalObject
   abstract deepClone:CloneMap result AbstractClonableInternalObject
   abstract deepNoClone:CloneMap
endclass
```

An example of how compound objects are cloned will illustrate the above.

```
class ClonableObjectWithParent extends
                  AbstractClonableInternalObject,
                  CompoundObject
  methods
    concrete shallowClone result AbstractClonableInternalObject
      ^ClonableObjectWithParent
          publicPart:publicPart
          privatePart:privatePart.
    concrete deepClone:cloneMap result AbstractClonableInternalObject
      publicPart := cloneMap deepClone:publicPart
      privatePart := cloneMap deepClone:privatePart
    concrete deepNoClone:CloneMap
      cloneMap deepNoClone:publicPart
      cloneMap deepNoClone:privatePart
endclass
```

Finally note that, for now, clone maps are only used when copying the internal structure of objects. It is often desirable to use clone maps on the level of objects themselves, or even provide clone maps at the language level. Although this seems no problem in principle, this was not our initial motivation for the introduction of clone maps, and we did not further investigate this possibility. We refer the reader to [Mittal,Bobrow,Kahn86] for this matter.

4.5.8 Mixin, Method and Instance Variable Declaration Reifiers and Slots

A set of reifiers has been defined for adding slots to objects. Listed below are the mixin and other method declaration reifiers. Their implementation is straightforward. They just add a slot to the public part of the receiver object. This slot associates the method pattern to the method body.

Method and Mixin Declaration Reifiers on Patterns

```
class AbstractPattern
  abstract class attributes
   MethodSlot MixinSlot
  methods
   reifier Method:(rightHand:AbstractExpression)
       using (context:MixinContext)
     context publicSlots add:
       (MethodSlot key:self value:rightHand)

   reifier Mixin:(rightHand:AbstractExpression)
       using (context:MixinContext)
     context publicSlots add:
       (MixinSlot key:self value:rightHand)
endclass
```

Reifiers for declaring variables — either local variables or instance variables — are restricted to unary patterns. Only the define reifier has been listed below, all other variations have a similar implementation. A variable declaration adds two accessor slots to the private part of the receiver: one for reading the

variable, one for writing the variable. These slots share a reference to a variable holder that stores the value of the variable. As mentioned in the introduction to Agora, the assignment reifier is interpreted as a message that is sent to the private part of the receiver object. The equivalent message of assigning for example the value 3 to an identifier x is the receiverless message x:3.

Variable Declaration and Assignment Reifiers on Unary Patterns (Identifiers)

```
class UnaryPattern extends AbstractPattern
  abstract class attributes
   VariableHolder ReadVariableSlot WriteVariableSlot
  methods
   reifier define using (context:ImperativeContext)
     variableHolder := VariableHolder new.
     context privateSlots add:
       (ReadVariableSlot
         key:(self asFunctionalCategoryPattern)
         value:variableHolder)
     context privateSlots add:
       (WriteVariableSlot
           key:(self asImperativeCategoryPattern)
           value:variableHolder)

   reifier <- (rightHand:AbstractExpression)
       using (context:ImperativeContext)
     context privatePart
       send:(self asImperativeCategoryPattern)
       client:(rightHand eval:(context asFunctionalContext))
endclass
```

The slots that are used in Agora, have the same functionality as slots in the calculus. We will not go into the details of all the different slots introduced in the above description. Their implementation is a straightforward extension of previously defined slots. The context in which slot-bodies are evaluated in Agora are a direct derivation of the context used in delegating messages. This is logical. The body of a method is evaluated in a context that is essentially the receiver object.

4.5.9 Summary of the Application of the Framework to Agora

Whereas the implementation of Simple was used to improve our initial proposal for a framework (e.g. the introduction of client and context objects, the introduction of evaluation categories), the implementation of Agora indicates refinements and extensions to this improved framework.

The most important refinement is the introduction of internal object structures. This adds an extra layer to the framework. It is a partial concretisation of how meta-objects can be implemented. The notion of wrapper objects can be important for the implementation of flexible imperative objects. The framework was extended with notions such as object cloning. It was shown that a cloning facility can be constructed while preserving flexibility in the internal representation of objects.

Most importantly it was shown that the framework, albeit simple in nature, is general enough to form the basis for the construction of a full-fledged object-oriented language. It is also important to note that the notions of reifier methods and classes were consistently used for the entire definition of Agora, except for message passing. Message passing, which forms the kernel of Agora is the only built-in language construction.

4.6 Extensions to Agora

The framework introduced in the previous section can be used to define a set of extensions to Agora. The purpose of this section is to illustrate the flexibility of the framework. In our previous discussions we encountered an entire range of language concepts that should be supported, either for the construction of frameworks or for the construction of multiple inheritance, …. A selection of such language concepts is presented below. They are expressed in the framework.

4.6.1 Public Instance Variables and Private Methods

The standard set of reifiers for Agora does not include the declaration of neither *private methods* nor *public instance variables*. This is a straightforward extension however, due to two facts. One is that instance variables are represented as a "get instance variable", and a "set instance variable" slot, and that reading and writing instance variables is done through message passing. The second fact is that we already provide a mechanism to provoke private methods, i.e. receiverless messages. Their implementation can be found in the previous section.

```
[ MakeX Mixin:
    [ xiv publicdefine ;
      testPrint:aString PrivateMethod: [ aString print ] ;
      test Method: [ testPrint:"test" ] ] ;

  X define: Object MakeX ;
  X xiv:4 ;
  X xiv print                              ---- 4 on transcript
  X test                                   ---- "test" on transcript
]
```

The implementation of the declaration reifiers for public instance variables and private methods is straightforward.

Agora Extension: Private Method Declaration Reifier

```
class ExtendedAbstractPattern extends AbstractPattern
  methods
   reifier PrivateMethod:(rightHand:AbstractExpression)
       using (context:StandardContext)
     context privateSlots add:(MethodSlot key:self value:rightHand)
endclass
```

Agora Extension: Public Instance Variable Declaration Reifier

```
class ExtendedUnaryPattern extends UnaryPattern
  methods
   reifier publicdefine using (context:StandardContext)
     variableHolder := VariableHolder new.
     context publicSlots add: (ReadVariableSlot key:self value:variableHolder)
     context publicSlots add:
       (WriteVariableSlot key:(self asImperativeCategoryPattern)
                          value:variableHolder)
endclass
```

4.6.2 Cloning Methods

In object-based programming languages there is a need for sophisticated cloning operations (see for example [Mittal,Bobrow&Kahn86]). Up until now we only discussed a simple clone reifier in Agora. As an example of a more sophisticated cloning operation we will show an extension of Agora in which cloning and initialisation of objects are combined.

For encapsulation reasons it is often desirable to combine initialisation of the private state of an object and copying of that object. In pure object-based languages, where new instances can only be made by copying old instances, one usually needs to initialise the newly created instance after copying it. This is done with an initialisation method. In most cases this initialisation method must only be invoked on a newly created instance, but in most languages this is not enforced. The same problem occurs in class-based languages (in Smalltalk for example, as a convention between Smalltalk programmers, a special message category of "initialisation methods" or "private methods" is reserved for this purpose, in C++ a special copy constructor mechanism is available).

We propose the following alternative where a special category of methods, the category of *cloning methods*, is reserved for cloning objects. A cloning method contains initialisation code. When a cloning method is invoked, it will be invoked on a copy of the receiver object.

```
[ MakeX Mixin:
    [ xiv define ;
     xiv:newx CloningMethod: [ xiv <- newx ] ;
     xiv Method: xiv Result:Integer ] ] ;

  X define: Object MakeX ;

  y define: X xiv:3 ;
  z define: X xiv:5 ;
  y xiv print ;                    --- 3 on transcript
  z xiv print ;                    --- 5 on transcript
]
```

The implementation of cloning method slots is less straightforward than can be expected. This has to do essentially with the scoping rules of Agora and the way messages are delegated (rather than being looked up) internally. On reception of a message an object does not know whether this will result in the execution of an ordinary method or for example a cloning method. So, it can not decide at this point whether to proceed with a copy of itself or not. This decision can only be made when a method is found. The point is that by then the (delegation) context contains parts of the receiver object that is to be copied. Consequently the receiver object can not be copied as a whole, but needs to be copied part by part, whereby all parts need to be assembled. Cloning maps are a very suitable solution for this problem.

Agora Extension: Cloning Methods

```
class ExtendedAbstractPattern extends AbstractPattern
  methods
    reifier CloningMethod:(rightHand:AbstractExpression)
        using (context:StandardContext)
      context privateSlots add:
        CloningMethodSlot key:self value:rightHand
endclass
```

```
class CloningMethodSlot extends MethodSlot
  methods
    concrete valueIn:(context:StandardContext)
      local variables aMap parentPart privatePart publicPart
                      receiver
      aMap := CloneMap new.
      parentPart := (aMap clone:context parentPart).
      privatePart := (aMap clone:context privatePart).
      publicPart := (aMap clone:context publicPart).
      receiver := (publicPart wrap).
      super valueIn: (context  parentPart:parentPart
                               privatePart:privatePart
                               publicPart:publicPart
                               receiver:receiver).
      ^receiver
endclass
```

The positive point about this implementation is that it has some interesting variations. In some cases it is not desirable that the entire receiver is copied. One variation is that the receiver is only copied up to the point where the cloning method is found. This can easily be realised. Instead of copying the parent part from the context (i.e. `parentPart := (aMap clone:context parentPart)`), the parent part is inserted ' as is' in the cloning map (i.e.`Map noClone:context parentPart`). Such variations are useful for the introduction of shared instance variables or when handling ' split objects' .

4.6.3 Stubs for Multiple Inheritance

In the section on mixins with multiple parents, a *message qualification mechanism* for mixins was discussed. It was based on the notion of inserting *stubs* in the inheritance chain. This mechanism can be adapted for Agora. The car-toy example that illustrated the usage is translated to Agora, extended with stubs, as follows:

```
[ makeCar Mixin:
  [ fuel publicdefine:"gasoline" ;
    print Method:[ self fuel print ] ] ;

    makeToy Mixin:
     [ age publicdefine:2 ;
       print Method:[ self age print ] ] ;

    CarStub Stub:
     [ ToyStub Stub:
       [ MakeCarToy Mixin:
         [ print Method: [ CarStub super:print ;
                           ToyStub super:print ] ] ]
  ] ;

  Car <> Object makeCar ;
  Toy <> Object makeToy ;
  ToyCar <> Car CarStub makeToy ToyStub MakeCarToy ;
  aToyCar <> ToyCar ;
  aToyCar print  ]
```

Notice that to stay in Agora's philosophy, stubs are declared in the same way that methods and mixins are declared (they can even be nested as in the above example). A stub is inserted in between two mixin applications by sending a corresponding (stub) message. An object, upon reception of a message that leads to the selection of a stub-slot, inserts that stub in its inheritance chain. A special super invocation reifier is provided that takes stubs into account, i.e. to invoke operations of non-direct parents.

Remark that the above example can also be interpreted in a less operational manner. A mixin declares the number and formal names of its possible parents (with the stub declarations). These formal names can be used in parent invocations. The formal names are bound to actual parents in the mixin application chain by inserting references to the formal parent names (with stub applications).

A second remark is about the role of nesting stubs and mixins. A stub name serves two purposes. It is used in the declaration of a public stub attribute with which the stub can be inserted in the inheritance chain. When inserted it serves as the name of a private attribute used in the parent invocation. Since private names are lexically scoped in Agora, mixins must be nested in the stub declarations of the stubs they want to use in their parent invocations. In the above example, the nesting of stubs imposes an order on the mixin-applications with which car-toys can be made. First a car must be made and then, and only then, this car can be extended to a toy-car. An alternative stub declaration in which the order of stub-application is free, can be devised. It would take the form:

```
{CarStub, ToyStub} Stub:
   [ MakeCarToy Mixin: [ … ] ]
```

The notion of stubs fits well in the framework. Still, extending the framework with stubs is a bit more complicated than the previous extensions. We will confine ourselves to a brief overview.

Agora must be extended with two new reifiers: a reifier to declare stubs and a reifier to invoke operations of non-direct parents. Upon evaluation, the stub declaration reifier inserts a 'stub-slot' in the public part of an object. This stub slot can be invoked by a (stub) message. The effect is that the receiving object is extended with a stub-object. This stub-object is set to contain, in a private slot, a reference to the object to which it is applied. The contents of this private slot is used by the parent invocation reifier to delegate messages to (a parent operation invocation is implemented by delegation !). Therefore the stub-object must contain a reference to the internal representation of the object that it refers to.

The above implementation involves a number of technicalities. How can the stub-object get a hold on a non-encapsulated version of the receiver object (in fact the internal representation of the receiver) ? How can we avoid that stub-objects are passed around and see to it that they are only used in parent invocations ? How do stub-objects influence object cloning ? We will not go into the details. We only note that it is possible to solve them within the constraints of the framework.

4.6.4 Single Slot Nested Objects

The standard flavour of Agora includes a fixed set of control structures. In object-oriented programming languages the construction of user-defined control structures is an important issue. In general a derivative of closures is employed for this purpose. The extension of Agora with *single slot nested objects* goes along these ways. The idea is to create objects with a single slot (objects that respond to only

one message), and that are dependent on their creation context. An example is given below. It is the classical object-oriented definition of boolean values. The '@' reifier combines a pattern and a body for that pattern to a single slot nested object. The example only features single slot objects that respond to unary messages, in general operator and keyword patterns can also be used in the creation of single slot objects. Standard argument passing applies for single slot objects.

```
[ makeTrue Mixin:
  [ ifTrue:tBlock ifFalse:fBlock Method:[ tBlock true ]
  ] ;

  makeFalse Mixin:
  [ ifTrue:tBlock ifFalse:fBlock Method:[ fBlock false ]
  ] ;

  True <> Object makeTrue ;
  False <> Object makeFalse ;

  aBoolean define ;

  aBoolean <- True ;

  aBoolean ifTrue:(true@["this" print])
           ifFalse:(false@["that" print])
]
```

Again, we will only give a brief overview of how the framework must be extended. Single slot nested objects are best regarded upon as a variation of wrapper objects. They are objects that reinterpret the notion of self reference and private attributes. Single slot objects inherit their 'self' and their private attributes from the surrounding context. This is the basis for their implementation. They are wrapper objects that contain a reference to the context in which they are created, and a reference to a single slot. Upon reception of the appropriate message, the body of this single slot is evaluated in the stored creation context. Single slot nested objects are a good example of the useful variations on wrapper objects.

4.6.5 Classes

Agora is at its heart a prototype based programming language. Still, following the analysis of a previous section, classes can be reintroduced. It suffices to make a distinction between 'class' objects and 'instance' objects. In the following example this distinction is introduced by the 'class' reifier. Variables declared with the class reifier can only contain class objects. Class object can only be sent mixin and cloning messages. A clone of a class object is an instance object. Instance objects can not be sent mixin nor cloning messages.

```
MakePerson Mixin:
  [ name define ;
    name:newName CloningMethod:[name <- newName] ;
    name Method:name ] ;

MakeSportsman Mixin:
  [ cardnumber define ;
    number:newNr CloningMethod:[cardnumber <- newNr ] ;
    number Method:cardnumber ] ;

Person class: Object MakePerson ;
SportsPerson class: Person MakeSportsman ;

john define: Person name:'John';
```

Class objects are another good variation of wrapper objects. They are wrapper objects that filter the accepted messages. They do so by heavily relying on the notion of pattern-categories. All messages received by a class object are delegated to the internal object structures in a special pattern category that is only compatible with mixin and cloning patterns. All messages received by an instance object are delegated to the internal object structures in a special pattern category that is *not* compatible with mixin and cloning patterns. Accordingly, a mixin message sent to an instance object results in a 'message not understood' error.

4.6.6 Abstract Methods, and Abstract Class Attributes

Obviously the introduction of *abstract methods* is a very easy extension of the framework. It suffices to define an abstract method declaration reifier that stores an abstract method slot in the public part of an object. This slot responds with an error when selected. Or, even better, the cloning of objects could be adapted such that an object with an abstract method slot returns an error when cloned. Concretisation of an abstract method relies on method overriding. In a statically typed variant of Agora the information provided by an abstract method declaration could be useful.

```
MakeButton Mixin:
  [ draw:window AbstractMethod ]
```

The introduction of *abstract class attributes* is less trivial. Concretisation of abstract class attributes relies on overriding of private attributes. Since in standard Agora private attributes are lexically scoped (and not dynamically) this proves to be a problem. In fact, the lack of overridable private attributes is an open problem for Agora. Preliminary investigations have shown that the solution is closely related to introducing an explicit encapsulation operator in Agora (such as can be found in Simple), thereby simplifying the entire structure of internal and context objects. This needs further attention.

4.6.7 A Simple Form of Monotonic Reclassification

Mixins can be applied to objects. The result is a new extended object, that shares the parent object with all other such extensions. In some cases an object must be extended without resulting in a new object. This is a form of *monotonic reclassification*. Consider the following example. A person john becomes a doctor. Note that john truly *becomes* a doctor: all the references to john see john as a doctor after the object that represents john is extended. In an extension of Agora this is realised by applying a mixin in an imperative manner (previously all mixin applications returned a result: the extended object).

```
MakePerson Mixin:
  [ name define ;
    name:newName CloningMethod:[name <- newName] ;
    printname Method:[name print] ] ;

MakeDoctor Mixin:
  [ printname Method:['Dr. ' print ; super:printname] ] ;

Person define: Object MakePerson ;

john define: Person name:'John' ;
johny <- john ;

john MakeDoctor ;

johny printname              --- prints Dr. John
```

Wrapper objects play an important role in the implementation of the above imperative mixins. The idea is to extend the internal object structures of an object while keeping the same wrapper object. For the rest normal mixin application does it.

The above is a simplified form of *reclassification*. It allows for an object to gain new attributes. More powerful mechanisms are imaginable. A similar notion to stubs, for example, could be used as marker points to drop attributes from an object.

4.6.8 Classifiers

Mixins tend to split up the inheritance hierarchy into small chunks of behaviour. Generally, the number of attributes declared in a mixin is much smaller than the number of proper attributes declared in a class with a ' plain' inheritance mechanism. Therefore mixins form highly combinable primitives for the construction of objects. On the other hand a mechanism is needed to manage this combinatorial explosion. In the multiple inheritance literature the notion of *classifiers* [Hamer92] or *inheritance dimensions* [McGregor&Korson93] has already been proposed for this purpose. Classifiers can be easily adapted for mixin methods.

Consider the following example. The fact must be recorded that the mixin to turn a person into a female can not be combined with the mixin to turn a person into a male. The notion of gender is introduced at the level of persons. It is formally declared that a person can be classified according to its gender (i.e. a person can have a gender dimension in her/his inheritance chain), but can only be classified once according to the gender. Both the `MakeFemale` and the `MakeMale` mixins subscribe to the gender classifier. An attempt to apply both to the same person will result in an error.

```
MakePerson Mixin:
  [ gender ExclusiveClassification ;
    …
  ] ;

MakeFemale Mixin:
  [  Classification:gender ;
    …
  ] ;

MakeMale Mixin:
  [  Classification:gender ;
    …
  ]
```

The evaluation of classifier declarations results in the insertion of classification slots into an object. A classification slot will contain the mixin patterns of the mixins applied to the object that contains the slot. Each mixin application must check and update classification slots for possible conflicts. In an actual extension of the framework this can be realised by an ingenious system of delegating classifier information. We will not go into the details.

Other useful classifiers have been investigated. A covering classifier for example can be used to enforce the application of at least one mixin out of a selection of mixins. An object that is not complete with respect to a covering classifier is an abstract object.

4.7 Conclusion

In this chapter we discussed how our framework can be refined with an inheritance mechanism by adding a layer to it.

For this purpose we discussed the design issues that are involved, including issues such as multiple inheritance, constraining inheritance hierarchies, scoping issues, ... We came up with a novel inheritance mechanism on objects called mixin-methods, and discussed a language (Agora) that incorporated this inheritance mechanism. Other salient features of Agora are: it has a minimal syntax — essentially message passing syntax — due to its extensive use of reifier expressions; it is slot based; it features nesting of mixin-methods; it is prototype based.

We discussed a framework layer that handles mixin-method based inheritance. We showed that this layer is a refinement of our basic framework due to the fact that the mixin-method inheritance mechanism can be totally encapsulated, i.e. it has no effect on the interface of meta-objects. We discussed an internal object representation that can be used to implement mixin-methods. We also discussed cloning (or copying) mechanisms for objects.

Finally we showed how the refined framework can be used to define a set of useful extensions to Agora.

Chapter 5

A Reflective Framework

5.1 Introduction

Now that we have defined an open design for object-based and object-oriented programming languages, we can focus on how to turn this open design into a reflective system. Turning an open design into a reflective system differs in a fundamental way from the conventional approach of defining reflective languages.

Conventionally a reflective language is defined by giving a meta-circular interpreter for it. This meta-circular interpreter can contain various circularities that are resolved by imagining an entire tower of meta-circular interpreters. In an actual running system various mechanism are used to implement this tower. Resolving these circularities is an essential step in making a running reflective language.

We will take another approach. As we said in the chapter on reflection a programming language is turned into a reflective one by *extending* it with reflection operators. We will consider those open designs that are powerful enough so that reflection can be added as a full-fledged specialisation of the open design. The advantage is that in that case a formal relation exists between a reflective language and its open design. Reflection is added as an orthogonal language concept. Among others this means that the open design itself (meta-circularly defined or not) does not need to be altered in an ad hoc fashion to turn it into a running reflective system.

As discussed in the chapter on reflection, turning a system with an open design in a reflective system is a matter of 1) achieving a symbiosis of the implementation language of the open implementation and the engendered language, and 2) providing the necessary reflection operators that may or may not avoid reflective overlap. Our discussion will follow these steps.

A symbiosis between two object-oriented languages enables objects to freely travel from one language to the other. First we will show how an object-oriented language can achieve a symbiosis with its underlying object-oriented implementation language. We will also show that this can be done with a fairly general mechanism. A symbiosis between an object-oriented programming language and its implementation language will be achieved by the introduction of *conversion-methods* and objects that incorporate *reflection equations*. The properties of these will be discussed.

In practice, the choice of the reflection operators is an important issue. Reflection operators must give access to both the base- and meta-level interface of the meta-system. A fully reflective language must give access to the entire base- and meta-level interfaces. We will discuss different sets of operators each with different characteristics, and show that to a certain degree, making a choice between them is a matter of taste.

Exactly as discussed, both the linguistic symbiosis and the reflection operators are added to the framework as an extra layer, i.e. reflection operators are in some sense not different of any other extension of the framework that adds new sorts of expressions and new sorts of objects. In fact the extension of the framework with reflection involves extending the framework's object hierarchy (with conversion objects that realise the symbiosis) and the expression hierarchy (with the actual reflection operators).

5.2 Object-based Reflection

5.2.1 Linguistic Symbiosis

Both Agora and its implementation language are object-oriented languages. The purpose of this section is to show how objects from Agora's implementation language can be used as Agora objects, i.e. how messages, expressed in Agora, can be sent to implementation language objects. Vice versa, we will show how Agora objects can be used as objects from the implementation language, i.e. how messages, expressed in the implementation language, can be sent to Agora objects. This is depicted informally in the following figure.

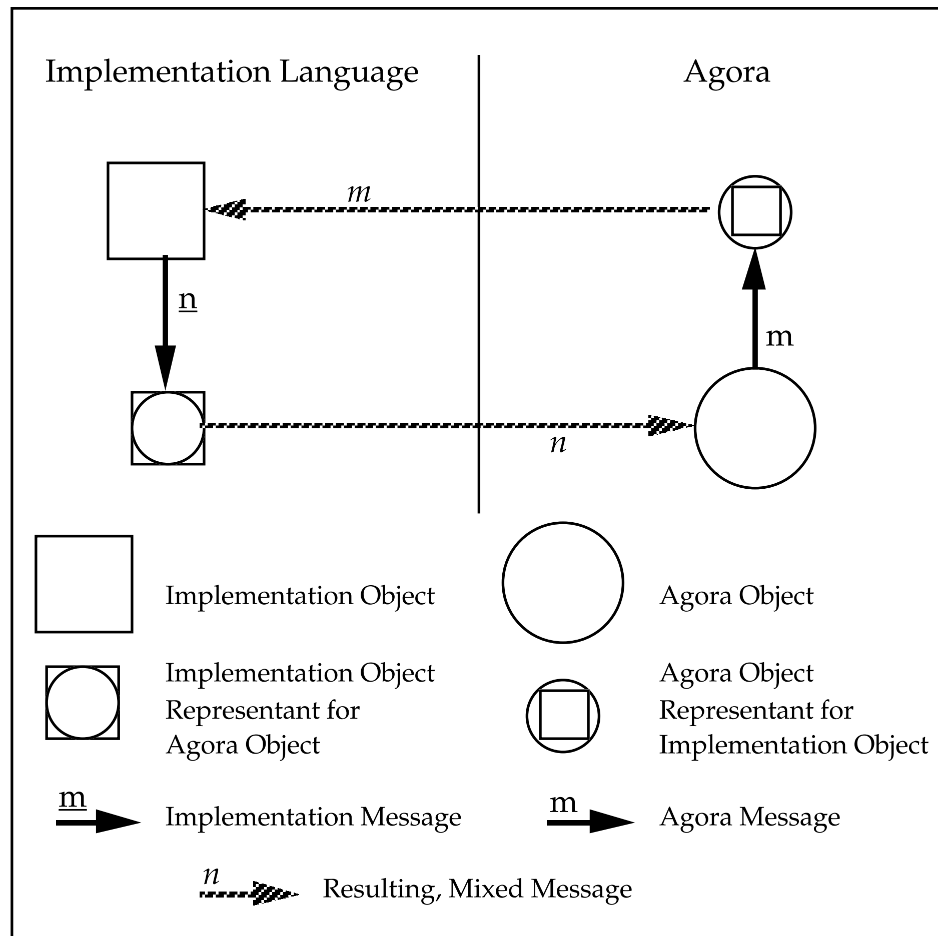


Figure 5.1

Before plunging into the technical details of the symbiosis of Agora and its implementation language, we will need some terminology. The distinction between Agora objects and implementation level objects will be blurred because after the symbiosis, objects will be able to travel between Agora and its implementation language. The simple terminological difference between Agora objects and implementation level objects is not good enough anymore. Therefor we will need a new terminology. The point is that we will need to make a distinction between the language in which an object is expressed and the language from which messages can be sent to an object. First of all we can make a distinction between *implicit messages* — messages expressed in the implementation language — and *explicit messages* — messages expressed in Agora. Secondly we will talk about an *explicitly encoded object* when this object is expressed in Agora, and about an *implicitly encoded object* when this object is expressed in the implementation language. Not every explicitly encoded object need to be referenced from within an Agora program. An object that can be sent implicit messages is called an *implicitly referable object*, an object that can be sent explicit messages is called an *explicitly referable object*. Finally we will simply talk about an implicit (explicit) object when this object is both implicitly (explicitly) encoded and referable. The following table summarises our terminology.

	Implementation Language	Agora
referable	Implicitly Referable Object	Explicitly Referable Object
encoded	Implicitly Encoded Object	Explicitly Encoded Object
referable & encoded	Implicit Object	Explicit Object

Figure 5.2

For one particular kind of objects this terminology can be interpreted in an ambiguous way. This is the source of much terminological confusion in object-oriented reflective programming languages. A meta-object is an object that is both implicitly and explicitly referable, albeit with two different protocols. To illustrate this, let us have a look at how explicit objects are represented in the implementation language. Each explicit object is represented at the implementation level by an implicitly referable meta-object. The latter will be called the *representation* of the former, the former will be called the *referent* of the latter. An explicit message to an explicit object is represented (or implemented) by an implicit message to the implicitly referable representation of that object, albeit a message with a different signature.

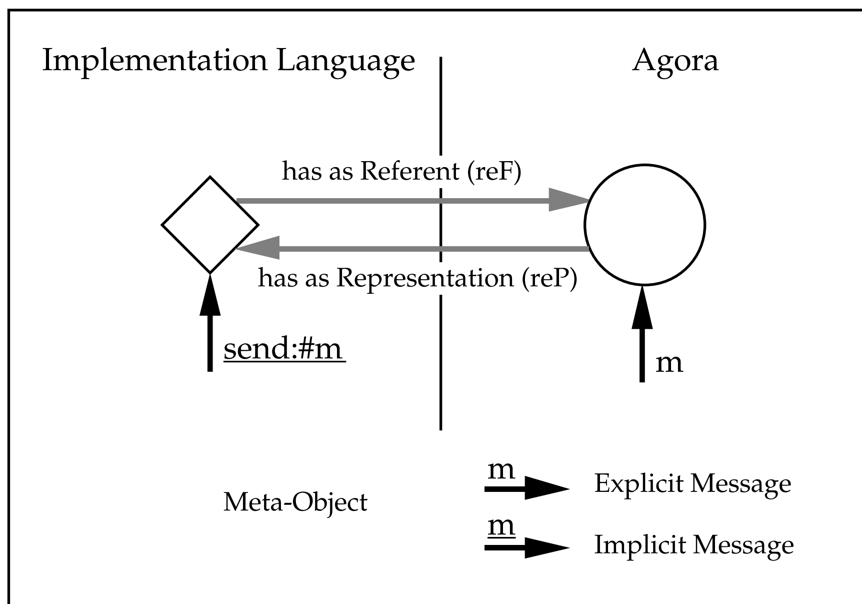


Figure 5.3

If the relation 'reP' associates each explicit object with its representation object, and the relation 'reF' associates each meta-object with its referent object, then the following holds for message passing between objects (depicted in the next table). Pattern objects are conveniently represented as '#x:y:z:', and argument lists as '{a1, … an}'.

<table>
<tr><th>Agora Objects and Their Representations (rule 1a)</th></tr>
<tr><td><pre>
                    reP[o x1:a1 x2:a2 … xn:an]
                               =
        reP[o] send:#x1:x2:…xn:
               client:(StandardClient private:{reP[a1], … reP[an]})

if o, a1, a2, … , an are explicitly referable objects,
and o x1:a1 x2:a2 … xn:an is an explicitly sent message
</pre></td></tr>
<tr><th>Meta-Objects and Their Referents (rule 1b)</th></tr>
<tr><td><pre>
         reP[reF[mo] x1:reF[ma1] x2:reF[ma2] … xn:reF[man]]
                               =
          mo send:#x1:x2:…xn:
              client:(StandardClient private:{ma1 … man})

if mo, ma1, ma2, … , man are implicitly referable meta-objects,
and mo send:(…) client:(…) is an implicitly sent message
</pre></td></tr>
<tr><th>Equality of Referents and Representations (rule 1c)</th></tr>
<tr><td><pre>
                 reF[mo1] = reF[mo2] ⇔ mo1 = mo2

                  reP[o1] = reP[o2] ⇔ o1 = o2

if mo1, mo2 are implicitly referable meta-objects, and
if o1, o2 are explicitly referable objects
</pre></td></tr>
</table>

Notice that it can be proved that the relations reF and reP are in a sense inverse relations with respect to message passing. It can be shown that an explicit message sent to 'reF[reP[o]]' has the same effect as an explicit message sent to 'o'.

```
reP[ reF[reP[o]] x1:reF[reP[a1]] … xn:reF[reP[an]] ]
=                                                          (rule 1b)
reP[o] send:#x1:x2:…xn:
       client:(StandardClient private:{rep[a1] … reP[an]})
=                                                          (rule 1a)
reP[o x1:a1 … xn:an]

therefore                                                  (rule 1c)
reF[reP[o]] x1:reF[reP[a1]] … xn:reF[reP[an]] = o x1:a1 … xn:an
```

Conversely it can be shown that an implicit 'send:client:' message to 'reP[reF[mo]]' has the same effect as an implicit 'send:client:' message to 'mo'.

```
reP[reF[mo]] send:#x1:x2:…xn:
             client:(StandardClient private:
                          {reP[reF[ma1]], … , reP[reF[an]]})
=                                                          (rule 1a)
reP[ reF[mo] x1:reF[ma1] … xn:reF[man] ]
=                                                          (rule 1b)
mo  send:#x1:x2:…xn:
    client:(StandardClient private:{ma1 … man})
```

Returning to the question of how to construct a symbiosis, we can now consider two conversion methods for implicitly referable objects. The first — named 'asImplicit' — turns a meta-object into an implicitly referable object with the protocol of the referent of the meta-object. The second — named 'asExplicit' — turns an arbitrary implicitly referable object into a meta-object, i.e. into a representation of an explicitly referable object with the same protocol as the initial implicitly referable object. Note that these conversion methods are implementation level methods, i.e. they can only be sent at the implementation

level. The two conversion methods are illustrated in the following figure.

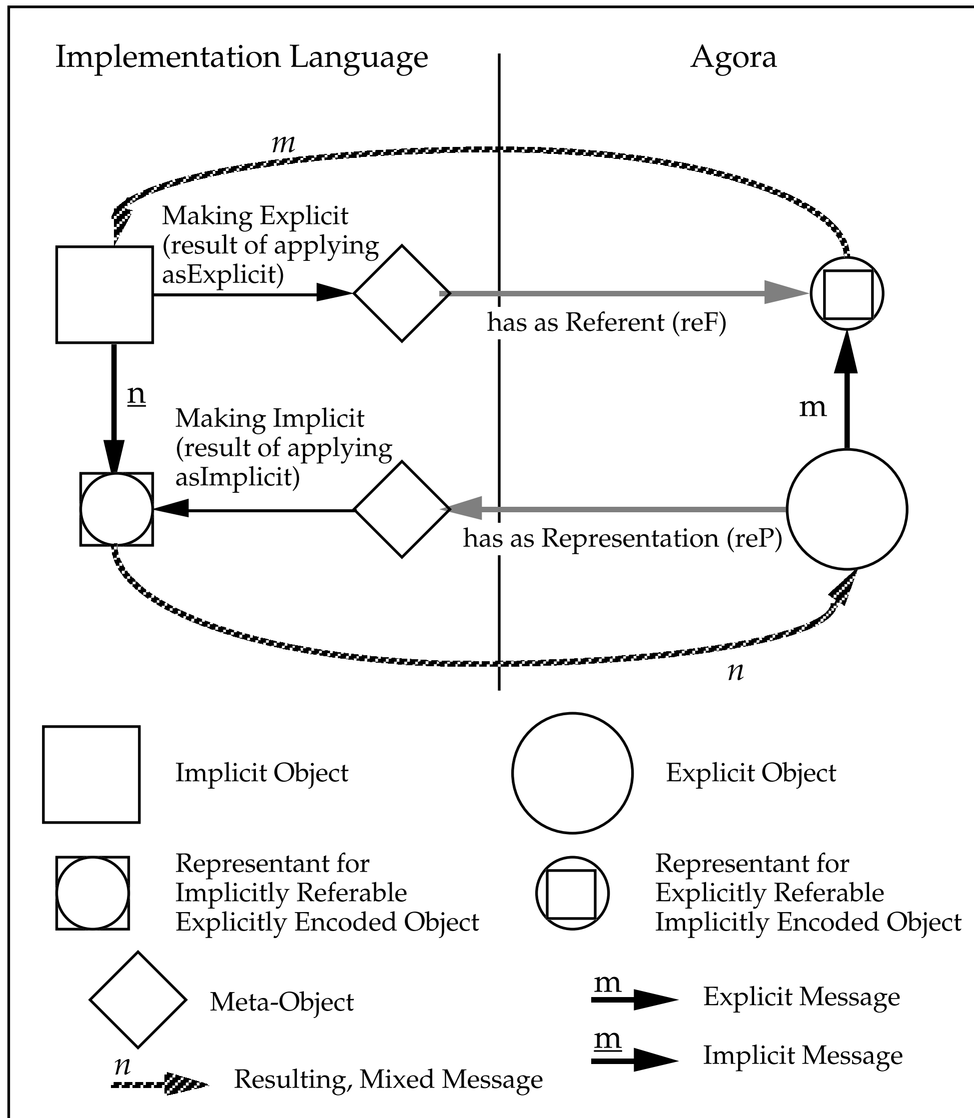


Figure 5.4

These conversion methods are crucial in achieving a symbiosis between Agora and its implementation language. The 'asImplicit' conversion method allows an object to travel from an Agora program to a program expressed in the implementation language. Conversely the 'asExplicit' conversion allows an object to travel from the implementation level to an Agora program. The conversion methods are defined by the following equalities :

Reflection Equations for Conversion Methods
reP[o *x1*:reF[a1 asExplicit] … *xn*:reF[an asExplicit]] asImplicit = (reP[o] asImplicit) *x1*:a1 … *xn*:an if o is an explicitly referable object a1, a2, … are implicitly referable objects and o *x1*:… … *xn*:… is an explicitly sent message
(reF[o asExplicit]) *x1*:a1 … *xn*:an = reF[(o *x1*:(reP[a1] asImplicit) … *xn*:(reP[an] asImplicit)) asExplicit] if o is an implicitly referable object a1, a2, … are explicitly referable objects and o *x1*:… … *xn*:… is an implicitly sent message
(o asExplicit asImplicit) = o = (o asImplicit asExplicit) if o is an implicitly referable meta-object

The 'asImplicit' and 'asExplicit' conversion methods are by axiom inverse methods for meta-objects. Intuitively, the previous equalities can be interpreted as a form of distribution of the 'asImplicit' and 'asExplicit' conversions over message passing.

The 'reP' and 'reF' relations on the one hand and the 'asImplicit' and 'asExplicit' conversion methods on the other hand should not be confused. The former are relations between implicit objects and explicit objects, i.e. a relation that can be observed to exist, or not. The latter are methods that must be explicitly applied. Furthermore the protocol changes involved are of a different nature. The 'reP' relation, for example, relates an explicitly referable object with an arbitrary protocol to an implicitly referable object with a protocol that is comprised of essentially a 'send:client:' method. The 'asImplicit' conversion method, however, converts the representation of an explicitly referable object with an arbitrary protocol to an implicitly referable object with the same protocol.

The role of the 'asImplicit' and the 'asExplicit' conversion methods will be illustrated with an example. Consider implicit expression objects. The method '`asExplicit`' for expressions converts an implicitly encoded expression object (i.e. an implementation object of type '`AbstractExpression`') into an explicitly referable object. The resulting object can receive explicit evaluation messages. Upon reception of an evaluation message '`eval:`', expressed in Agora's message passing, the converted object translates this message to an '`eval:`' message on the implementation level. It also takes care that the context argument is translated into an implicit referable object, and the result is translated back into an explicitly referable object. These translations are necessary since the message was sent from within an Agora program. Obviously the 'asExplicit' conversion method will play an important role in making primitive objects — that are present in the implementation — available to Agora programs.

Conversely, an Agora object that implements an '`eval:`' method, can be transformed to a implicitly referable object with the conversion method '`asImplicit`'. This method will translate this explicitly encoded object into an implicitly referable expression object, i.e. the implementation-language-representant (preferably of type '`AbstractExpression`') for explicitly encoded expression objects that can receive implicit evaluation messages. Upon reception of an implicit evaluation message '`eval:`' the converted object translates this

message to an Agora style 'eval:' message. Care is taken that the context argument is translated to an Agora object, and the result is translated back to an implementation object. Obviously, the 'asImplicit' conversion method will play an important role in the implementation of reflective operators.

So, we see that this form of symbiosis is but a mere extension of handling primitive data-types, as can be found in most implementations of programming languages. A primitive data-type is a data-type from the implementation language that is transported to the implemented language. In most languages, only the direction of implementation language to implemented language is supported; from the viewpoint of reflection, the other direction is much more interesting. In reflection terms the conversion methods allow objects to "shift levels".

The implementation of conversion methods for the symbiosis is straightforward in principle, but tedious in practice. Let us first look at the 'asExplicit' method. This method is for example defined for expression objects. It converts an implicit expression object into an explicitly referable expression object of which the definition is found below. Notice that the class of explicitly referable expression objects is a concretisation of the abstract class of meta-objects. Its implementation is one of translating explicit messages to implicit messages according to the following schema.

asExplicit Conversion Method

```
class AbstractExpression extends ObjectThatCanBeMadeExplicit
  abstract class attributes
   ExplicitlyReferableExpression
  methods
   concrete asExplicit result AbstractMetaObject
     ^ExplicitlyReferableExpression expression:self
endclass
```

Explicitly Referable Expressions

```
class ExplicitlyReferableExpression extends AbstractMetaObject
  constants
   EvalPattern = KeywordPattern name:"eval:"
  instance variables
   expression:AbstractExpression
  methods
   concrete send:pattern client:client result AbstractMetaObject
     if pattern = EvalPattern
       then
         ^(expression eval:(client arguments asImplicit))asExplicit
       else
         … raise an error
endclass
```

The implementation of the reverse 'asImplicit' conversion method is as straightforward as the previous one. This conversion method is defined only on meta-objects. In principle it translates an anonymous meta-object into an anonymous implementation level object. More specific variants of this conversion method can be useful. For example an 'asImplicitExpression' conversion method would translate a meta-object into an object of the 'ExplicitlyEncodedExpression' class of which the definition can be found below. Again, this class does a simple translation of messages.

asImplicit Conversion Method

```
class AbstractMetaObject
  abstract class attributes
    ExplicitlyEncodedExpression
  methods
    concrete asImplicitExpression result AbstractMetaObject
      ^ExplicitlyEncodedExpression object:self
endclass
```

Explicitly Encoded Expressions

```
class ExplicitlyEncodedExpression extends AbstractExpression
  constants
    EvalPattern = KeywordPattern name:"eval:"
  instance variables
    object:AbstractMetaObject
  methods
    concrete eval:context result AbstractMetaObject
      ^(object
          send:EvalPattern
          client:(StandardClient arguments:(context asExplicit))
        ) asImplicit
endclass
```

Finally, note that because meta-objects (not their referents) are implicitly referable objects, the 'asExplicit' conversion method should be defined for them.

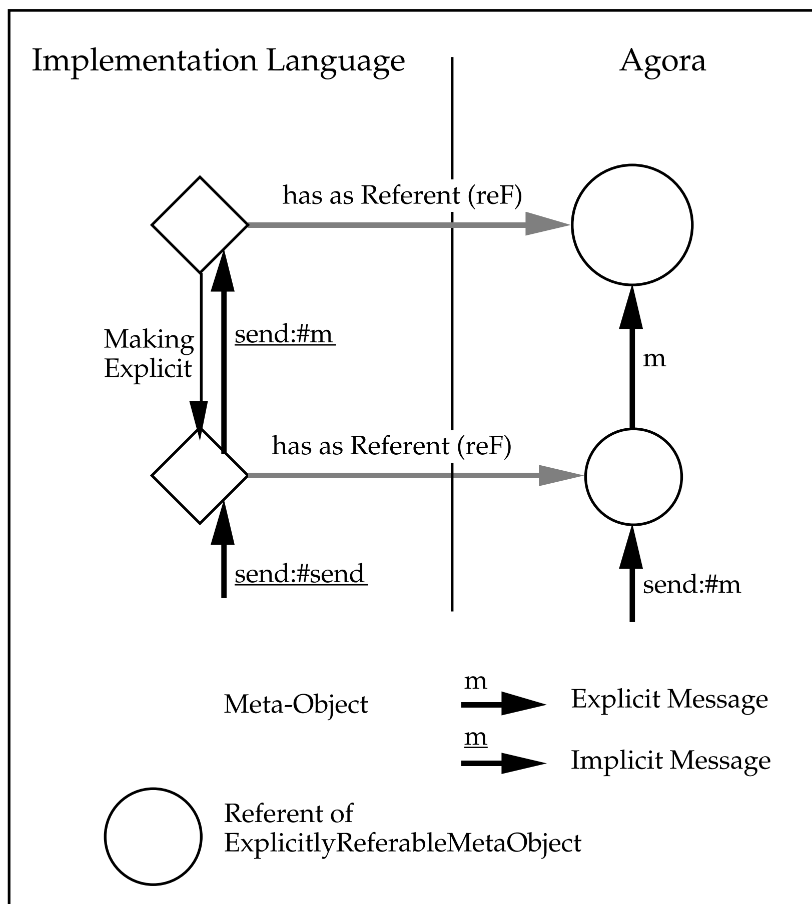


Figure 5.5

The according conversion class can be found below. It encodes meta-objects to which explicit 'send:client:' messages can be sent. Remark that, on meta-objects, the 'asExplicit' conversion can be applied an infinite number of times.

Explicitly Referable Meta-objects

```
class ExplicitlyReferableMetaObject extends AbstractMetaObject
  constants
   SendPattern = KeywordPattern name:"send:client:"
  instance variables
   aMeta:AbstractMetaObject
  methods
   concrete send:pattern client:client result AbstractMetaObject
     if pattern = SendPattern
       then
         ^(aMeta
            send:(client firstArgument asImplicit)
            client:(client scndArgument asImplicit))asExplicit
       else
         … raise an error
endclass
```

5.2.2 Simple Meta-Programming Operators for Agora

To illustrate the linguistic symbiosis we will discuss a set of reflection operators that is directly inspired by the above conversion methods. As we saw in the previous section the 'asImplicit' method can be used for example to convert an implicitly referable expression object into an explicitly referable expression object. This is called a quoting operator when provided as a language construct. Similarly, meta-objects can be converted into explicitly referable meta-objects. We will also illustrate how the inverse operations — that of converting an explicitly encoded expression or meta-object into an implicitly referable object — can be made useful.

The quoting reifier (form: 'e **quote**') allows us to get hold on expressions as Agora objects in what is usually called a *meta-program*. A quoted expression is an object that can be sent an explicit evaluation message, given a context as argument. The following meta-program evaluates the *object-level program* '"hello world" print' in an initially empty context. We presume that somewhere an appropriate prototype 'EmptyContext' has been defined. This prototype should conform to the protocol of standard contexts.

```
EmptyContext define: … ;    --- an initially empty context

aProgram define ;
aProgram <- ("hello world" print)quote ;
aProgram eval:(EmptyContext clone)  --- prints "hello world"
```

This is a typical example of *meta-programming*: allowing us to manipulate programs as first-class objects, but on the other hand absorbing (leaving implicit) the evaluator for these programs. Some remarks must be made. Consider the following example. The object-level program creates a point object that is returned as result[1].

1 Note that in Agora block expressions do *not* evaluate to something like closures (such as is the case in Smalltalk) but rather all component expressions are evaluated. The return reifier indicates what result must be returned.

```
aProgram  <- ( [ Point define: … ;   --- a point prototype
                 Point x:3 y:4 return ] )quote ;

p <- aProgram eval:(EmptyContext clone) ;

p x   --- ERROR: does not understand
      --- p is a result at the meta-level !
      --- therefore p is a meta-object
```

The first remark is that due to the fact that the evaluation is done in an explicitly given context the object-level program in the example can not refer to any of the prototypes defined in the meta-level program. Object-level programs must be 'self-contained' with respect to the referenced objects. Secondly, and more importantly, it must be noted that the result of an explicit evaluation is a meta-object. This is not only a direct result from the definition of our conversion methods, but it is also what we want. Whereas an object-level program deals with referents directly, the meta-program deals with the representations (meta-objects) of the objects of its object-level program. An evaluator (or a meta-system in general) that does not respect this is said to be a level-crossing evaluator [Smith82].

The implementation of the quoting operator is straightforward, and relies on the symbiosis of Agora and its implementation language. A quote reifier returns, upon evaluation, its receiver as an explicitly referable Agora object.

Quoting Expressions (without precautions to avoid reflective overlap)

```
class AbstractExpression
  methods
   reifier quote result AbstractMetaObject
         using (context:StandardContext)
      ^(self asExplicit)
endclass
```

Explicit meta-objects can be obtained in a way that is similar to the way expression objects are obtained. Similar to the quoting operator we introduce a reifier (form: `'e asMeta`') that transforms the representation of its evaluated receiver into an explicitly referable meta-object. Here again a typical example of a meta-program can be given. A meta-program that sends an explicit message to a meta-object. We presume that somewhere appropriate prototypes 'UnaryPattern' and 'EmptyClient' have been defined. These prototypes should conform to the protocols of respectively standard patterns and standard clients.

```
UnaryPattern define: … ;     --- a pattern prototype
EmptyClient define: … ;      --- a client prototype
Point define: … ;            --- a point prototype

metaOfP define ;
p define: Point x:3 y:4;
metaOfP  <- (p asMeta) ;
result <- metaOfP send:(UnaryPattern name:"x")
                  client:(EmptyClient clone)
```

Similarly to the above example, and for the same reasons, the result of an explicit message to a explicitly referable meta-object is a meta-object.

The definition of this new operator is as straightforward as the definition of the quote operator. It also relies on the symbiosis of Agora and its implementation language.

Accessing Meta-Objects

```
class AbstractExpression
  methods
   reifier asMeta result AbstractMetaObject
        using (context:StandardContext)
     ^((self eval:context) asExplicit)
endclass
```

Notice that, in contrast with other object - meta-object approaches, an object does not contain a reference to its meta-object but, rather, that object and meta-object are different views for communicating with an object (much in the style of the reP relation). Every object in Agora is implemented as an object in the implementation language (typically called the meta-object, and having a 'send:client:' method in its protocol). According to our symbiosis this latter object can be made explicit, via the 'asExplicit' conversion method. It is made explicit as an Agora object, such that Agora messages can be sent to it. The kind of messages that can be sent are 'send:client:' messages. The implementation level objects that implement Agora objects, are in that respect no different of, say, context-, or expression objects at the implementation level. Still, the fact that there is already a relationship (i.e. the reP and reF relation) between Agora objects, and implementation level meta-objects (which is not the case for context, or expression objects for example) can make this a bit confusing.

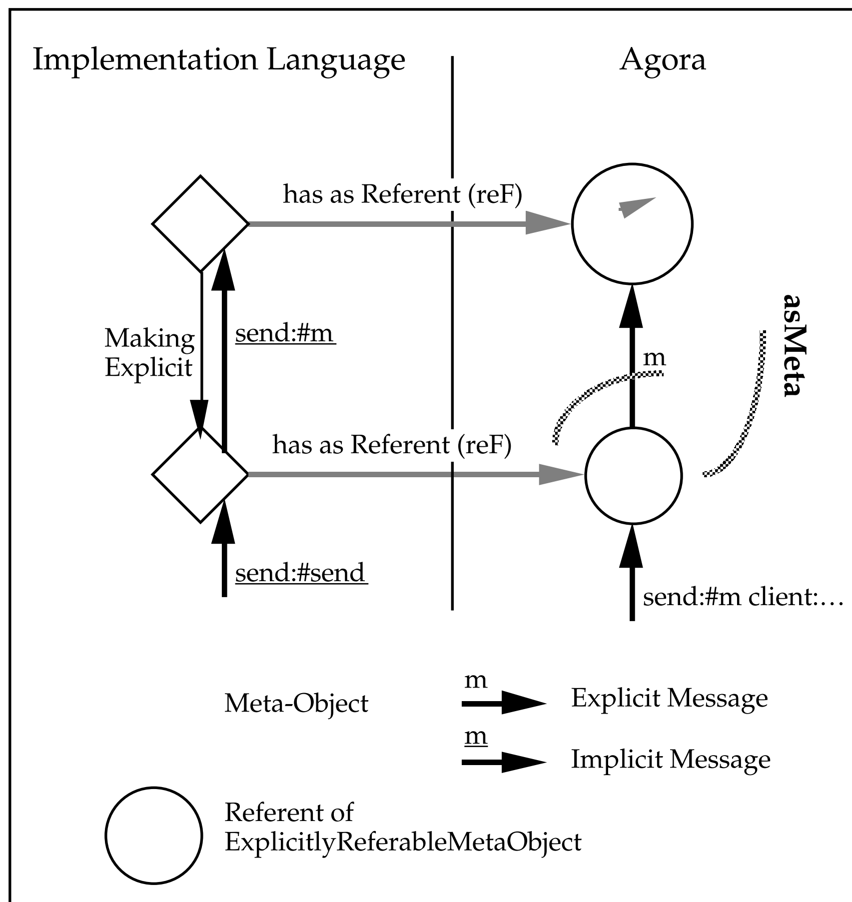


Figure 5.6

5.2.3 Simple Reflection Operators

The most important aspect of the meta-level interface of the open implementation of Agora is the extension of the class hierarchy of expressions and the class hierarchy of objects. In this section we will discuss two reflection operators that make this aspect of the meta-level interface available to Agora programs. These operators, also, are directly inspired by the conversion methods 'asImplicit' and 'asExplicit'. Whereas in the previous section we made use of the fact that expression and meta-objects can be made explicitly referable, we will now make use of the fact that explicitly encoded expression and meta-objects can be made implicitly referable by the 'asImplicit' conversion method.

The introduction of new sorts of expressions in an open implemented programming language, in principle, goes hand in hand with the introduction of new syntactic constructs. Mere extension of the expression class hierarchy is not enough, it is not even the goal. The goal is to be able to construct program trees that make use of the newly added expression objects. As we saw before, this can be realised, for example, with a generic syntax.

To keep things simple, however, the construction for reflectively adding new types of expressions, discussed here, will be of a flavour that avoids this complication. This construction is reminiscent of reifier functions in e.g. 3-Lisp [Smith82], albeit of a more primitive nature.

Furthermore it is our intention to illustrate the dynamic character of using the meta-level interface that comes with reflection. Previously, in the ‘plain’ open implementation, the usage of base- and meta-level interface were strictly separated in time. In case of reflection this need not be so.

The point is to offer a reflection operator that allows the dynamic extension of the program tree with explicitly encoded expression objects, i.e. an operator that, given a Agora object that implements an evaluation method, virtually installs this object in the program tree. The reifier (form: '`e` **`asExpression`**') we propose for this purpose is more or less the reverse of the quote operator. It evaluates its receiver expression '`e`' — that will be called the expression-definition of the absorbed expression —, transforms the result — that will be called the explicit expression-object of the absorbed expression — to an implicitly referable expression object, and sends an implicit evaluation method to this transformed object. Whereas the quote operator reifies parts of the program tree into explicitly referable objects, the asExpression operator absorbs explicitly encoded expression objects into the program tree.

Consider the following example. The goal is to construct an expression type that reifies the current context. For this purpose an appropriate expression object is defined. Each time this explicit expression object is absorbed in the evaluation process, by means of the 'asExpression' reifier, it reifies the current context.

```
Point define: … ;             --- a point prototype

MakeCurrentContextExp Mixin:
    [ eval:context Method:[ context return ] ] ;
CurrentContextExp define: Object MakeCurrentContextExp ;

currentContext define;
currentContext <- (CurrentContextExp asExpression) ;

a <- ( Point x:3 y:4 )quote ;
p <- a eval:currentContext ;
(p asObject) x print                    --- prints 3
```

This example features two different forms of reflective overlap. Firstly, the context that is reified by the 'currentContextExp' expression, is both reified and left implicit. This is apparent in the fact that the variable that points to the reified context also is part of this reified context. Secondly, and more importantly, the evaluation method of the explicit expression object is evaluated in the same context that it reifies. This evaluation method has, for example, access to the point prototype, both directly and via its context argument. Whereas the first kind of reflective overlap is the result of how the above program is formulated, the second kind is a direct result of the definition of the 'asExpression' reifier. We will see in the next section how an alternative set of reflection operators that avoid reflective overlap can be defined.

Explicitly encoded meta-objects can also be made implicitly referable. The reifier 'asObject' allows the absorption of explicitly encoded meta-objects. In the following example meta-objects are constructed that reply lazily to messages. The result of a message sent to a lazy object is computed only if a message is sent to this result. Therefore two different sorts of meta-objects are defined. The first kind ('Lazy' objects) that contains a reference to the object that is made lazy. The second kind ('ResultHolder' objects) that act as representants for the results of the messages sent to a lazy object. Notice that in the example, the first kind of meta-objects is put to use by an explicit application of the 'asObject' reifier. Whereas the second kind is created in the execution of an explicitly encoded 'send:client:' message. Since this latter is executed at the meta-level the so created meta-object is automatically absorbed.

```
MakeResultHolder Mixin:
  [ receiver define ; pattern define ; client define ;
    receiver:r pattern:p client:c CloningMethod:
      [ receiver <- r ; pattern <- p ; client <- c ] ;
    send:p client:c Method:
      [ (ResultHolder
            receiver: (receiver send:pattern client:client)
            pattern:p
            client:c) return ]
  ] ;
ResultHolder define: Object MakeResultHolder ;

MakeLazy Mixin:
  [ who define ;
    who:w CloningMethod: [ who <- w ] ;
    send:p client:c Method:
        [ (ResultHolder receiver:who pattern:p client:c) return ]
  ] ;
Lazy define: Object MakeLazy ;

Point define: … ;

p define: (Lazy who:((Point x:3 y:4) asMeta)) asObject ;
--- p contains a lazy point now
```

The definitions of both the 'asObject' and 'asExpression' operators are straightforward.

Installing Expressions and Meta-Objects (without precautions to avoid reflective overlap)

```
class AbstractExpression
  methods
   reifier asObject result AbstractMetaObject
        using (context:StandardContext)
      ^((self eval:context) asImplicit)
endclass
```

```
class AbstractExpression
  methods
   reifier asExpression result AbstractMetaObject
        using (context:StandardContext)
      ^(((self eval:context) asImplicit) eval:context)
endclass
```

This last definition is an interesting one since it illustrates two important aspects of a reflective system. The first is the notion of reflection levels. In the definition of the 'asExpression' reifier it is apparent that two evaluation messages are sent. Unlike a recursive call to the evaluator function, the above calls to the evaluator are 'cascaded'. The second call to the evaluator is sent to the result of the first evaluation. This obviously gives rise to layers of evaluation (reflection levels). All usages of the 'asExpression' reifier need two layers of evaluation. Closely connected is the notion of reflective overlap. The 'asExpression' reifier suffers from it since both layers of evaluation use the same context.

5.2.4 Nature of Meta-Programs and Reflective Overlap

As illustrated in the example in which contexts where reified, the 'asExpression' reifier introduces a form of reflective overlap. An explicitly encoded expression object is evaluated in the same context that will be passed as argument of the implicitly sent evaluation message. This reflective overlap can, but must not necessarily, be avoided. If we do want to avoid reflective overlap, the question of what should be considered part of the meta-program and what should be considered part of the object-level program must be answered.

As we saw in the first quoting example, quoting introduces a natural boundary between meta-program and object-level program. The result of a quote expression is a new object-level program. Conversely the expression-definition used in an 'asExpression' reifier should be part of the meta-program since, conceptually, it adds an expression type to the open implementation. Consider the following example. Obviously the main part of the program can be interpreted as a meta-level program. It defines a 'constant 3' expression, and evaluates some quoted object-level program. It is more than natural that, in the object-level program, this newly defined expression type can be used. Therefore the expression-definitions used in calls to the asExpression reifier from within the object-level program should be evaluated in the context of the meta-program rather than in the context of the object-level program. Such a strict separation of the meta-context and the object-level context also solves our problem of reflective overlap. The following example features the variant reifiers 'cleanQuote' and 'cleanAsExpression' that avoid reflective overlap.

```
MakeConstant3Exp Mixin:
    [ eval:context Method: [(3 asMeta) return ] ] ;
Constant3Exp define: Object MakeConstant3Exp ;

EmptyContext define: … ;    --- an initially empty context

aProgram define ;
aProgram <- ((Constant3Exp cleanAsExpression) print ) cleanQuote ;
aProgram eval:(EmptyContext clone)  --- prints "3"
```

Trivially, reflective overlap can be avoided by keeping track of meta-contexts by means of a stack mechanism. The explicitly given context in which a quoted expression is evaluated is pushed on the stack of meta-contexts. This stack is popped to return to the meta-level when the receiver expression of the asExpression reifier is evaluated.

Quoting Expressions

```
class QuoteExpression extends AbstractExpression
  instance variables
   quote: AbstractExpression
   inContext:StandardContext
  methods
   concrete eval:context result AbstractMetaObject
     ^(quote eval:(context push:inContext))
endclass
```

```
class AbstractExpression
  abstract class attributes
   QuoteExpression
  methods
   reifier cleanQuote result AbstractMetaObject
        using (context:StandardContext)
     ^((QuoteExpression quote:self inContext:context) asExplicit)
endclass
```

Installing Expressions and Meta-Objects

```
class AbstractExpression
  methods
   reifier cleanAsExpression result AbstractMetaObject
        using (context:StandardContext)
     ^(((self eval:(context pop)) asImplicit) eval:context)
endclass
```

5.2.5 Dynamic Reflection and Infinite Regress

As we said in the previous section, reflective overlap need not necessarily be avoided. One particular case where reflective overlap comes in handy is in the definition of *dynamic reflection*. As we explained in the second chapter, dynamic reflection is characterised by the fact that the number of times a program regresses is dynamically determined. Below is an example of a program that regresses infinitely (reflecting upon one's own behaviour in an infinitely recursive way). A regression expression is used in the evaluation of its own evaluation method. This definition looks very similar to a meta-circular definition, except for the fact that in this case there is no special provision to 'bottom out' of the circularity. The meta-circular definition is effectively used in its own interpretation.

```
MakeRegressingExpression Mixin:
  [ eval:context Method:
      [(RegressingExpression asExpression) return ] ] ;
RegressingExpression define: Object MakeRegressingExpression;

(RegressingExpression asExpression) --- infinite regression
```

The definition above is only possible in the case where the version of the 'asExpression' reifier is used that suffers from reflective overlap. Obviously, infinite meta-regress is easily constructed with such reifiers. Less obvious is how this sort of reflection can be applied to practical situations. This is reminiscent of what we said in our introduction of reflection: what use is it to keep on reasoning about one's self if this does not improve one's reasoning about the world. The use of dynamic reflection is in fact an open question. Here, we only point out a possible candidate that uses dynamic reflection in a useful way.

We talk about dynamic reflection when the number of 'reflection levels' is dynamically determined. One particular form of dynamic reflection occurs when the number of explicit meta-objects a particular object has, is dynamically determined — i.e. if the 'asObject' reifier has been applied a dynamically determined number of times. Notice that this is another kind of dynamic reflection than the above reifier that uses itself recursively in its evaluation method (leading to an undetermined number of evaluation levels). The dynamic aspect here has to do with levels of explicitly encoded 'send:client:' methods that are used in sending 'send:client:' messages (as depicted if figure 5.6). We will try to show that this can occur in a practical situation.

Consider writing a meta-circular definition for the Agora framework. One part of the job in doing so, is implementing a linguistic symbiosis between the newly defined Agora and its implementation language, the already defined Agora. The fact that both sorts of objects — the implicit Agora objects, and the explicit Agora objects — are so closely related doesn't seem to help. The problem is that objects need to shift levels. But, this is exactly the functionality provided by the 'asObject' and 'asMeta' reifiers. So, in the reflective variant of Agora, level shifting of objects can be absorbed. This is illustrated in the following sample of a hypothetical meta-circular Agora definition.

```
MakeMetaObject Mixin:
  [ …

    send:pattern client:client Method: [ … ] ;

    asImplicit Method: [ (self asObject) return ]

    … ] ;

MakeObjectThatCanBeMadeExplicit Mixin:
  [ asExplicit Method: [ (self asMeta) return ] ]
```

Notice that with each application of the 'asObject' reifier (to some object) in a program executed by the meta-circular Agora interpreter corresponds an application of the 'asObject' reifier in the code of the meta-circular interpreter (in fact to the meta-object, see figure 5.7). The number of times the 'asObject' reifier is applied to some object in the meta-circular interpreter is determined by the program it is evaluating. From the standpoint of this meta-circular definition this number is dynamically determined.

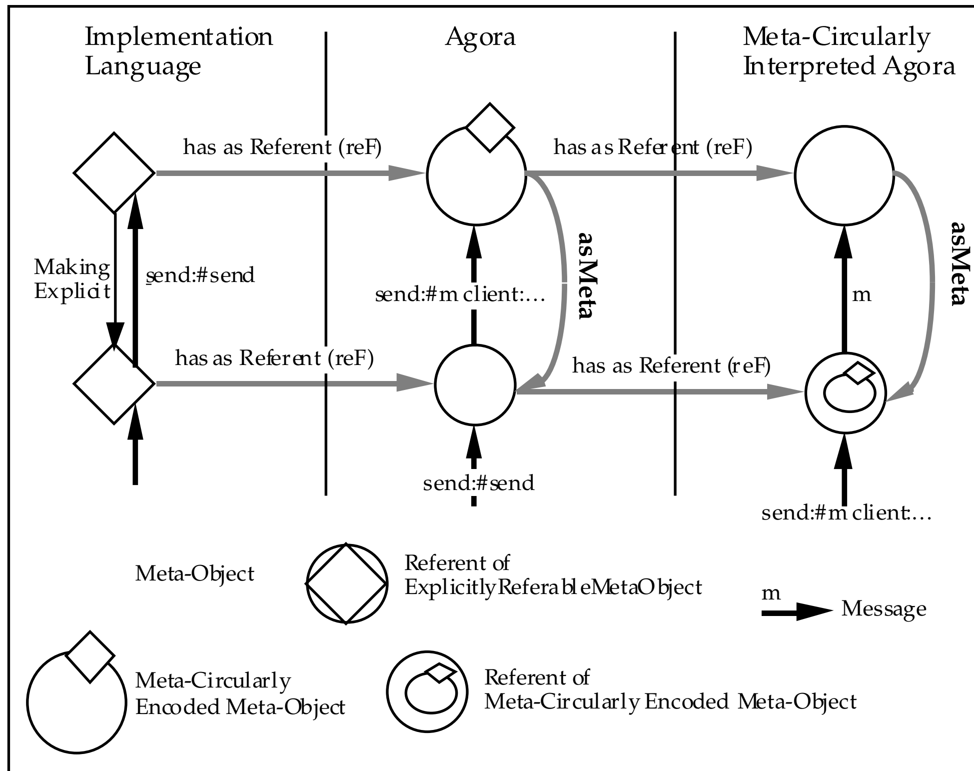


Figure 5.7

5.2.6 Full Abstraction and Compositionality

A final issue that is partially left open, is the role of compositionality and full abstraction in reflective programming languages. Apart from extensibility issues (as discussed before), it is clear that both concepts have an important role to play.

First of all full abstraction guarantees us that the meta-level programmer can not 'mess things up' more than is possible at the object level. For example, in a reflective object-oriented programming language where objects are not represented fully abstractly, a programmer can always break the encapsulation of objects. Consider again the non-abstract representation of meta-objects in the form of slot objects (see chapter 3). Imagine a reflective programming variant of Agora based on this alternative kind of meta-objects. As shown in the next example, in that case encapsulation of objects can not be ensured. The example features a turtle class that tries to encapsulate its location and heading variables. However, at the meta-level these instance variables can be freely accessed.

```
UnaryPattern define: … ;    --- a pattern prototype

MakeTurtle Mixin:
  [ location define: Point rho:0 theta:0*pi ;
    heading define: 0*pi ;
    home CloningMethod:[ … ];
    turn:turn Method: [ … ] ;
    forward:distance Method: [ …  ] ] ;
Turtle define: Object MakeTurtle ;

turtleLocation define ;
aTurtle define: Turtle home ;
```

```
--- breaking the turtle's encapsulation
turtleLocation <-
  (((aTurtle asMeta)
      lookup:(UnaryPattern name:"location")) value) asObject
```

Another way to look at this is that in a non abstract implementation, the view one has on objects at the implementation level is a finer view (finer in the sense that one sees more implementation details) than one has at the programming level. Since in a reflective system it is possible to switch between these levels, it is always possible to take the finer view. This makes reasoning about programs more difficult, and diminishes reusability. Similar remarks apply for compositionality.

Compositionality and full abstraction also have implications on possible optimisation techniques. It is for example, easier to provide alternative implementation strategies for abstract object representations than for non abstract object representations. Also reification of expressions in a compositional way is a prerequisite for a compositional semantic definition of a reflective language [Malmkjær90].

So, compositionality and abstraction, indeed play an important role in reflective programming languages that transcends the role they play in the definition of open implementations. However, it is not yet possible to estimate the full consequences of both.

5.3 Object-Oriented Reflection

The above reflection operators do not give full access to the open implementation of Agora. Although an operator was presented to add new expression types one important aspect of Agora was ignored: that of reifiers. Another element that was ignored is the ability to extend the existing classes from the framework that constitutes Agora's implementation. We will show that a more fine-grained linguistic symbiosis is needed.

5.3.1. The Evaluation and Declaration of Reifiers

Since Agora itself is a full-fledged object-oriented programming language, considerable freedom exists in the choice of reflection operators. Moreover, since Agora aims to be a general purpose programming language, an important factor in the choice of reflection operators, apart from being complete, is their practical applicability, and ease of use.

Agora is best extended with new expression types by the addition of new reifiers. We will discuss the different characteristics of two sorts of reifiers that exist in Agora: that of reifier classes, and that of reifier methods. We will see that reifier classes are more appropriate for dynamic reflection and that reifier methods are more appropriate for static reflection. The evaluation of reifier expressions (both messages and receiverless messages) has been left open until now. All that was said is that somehow each time a reifier expression is evaluated, a corresponding reifier method or class has to be evaluated. Given the above symbiosis, we are now ready to show a possible interpretation of reifier expressions.

Reifier Classes
The first kind of reifier expressions that will be discussed are the receiverless reifier expressions, or also called reifier pattern expressions. An example reifier pattern expression is given below. The receiverless 'trace' reifier will first give rise to the creation of an implicit trace expression object that is then sent an implicit evaluation message.

```
p define: Point ;
p x:3 y:4 ;

p <- trace:p ;
--- from here on all messages to p will be traced
```

We are now ready to show how reifier pattern expressions can be interpreted. Two choices exist according to whether we want to avoid reflective overlap or not. We will take the latter choice. The idea is that the expression object that corresponds to a reifier pattern is to be found in the evaluation context. The corresponding expression object is looked up by sending a message to the evaluation context. The arguments of this message are explicit expression objects. When a correct expression object is found, it is made implicit and it is sent an evaluation message. Notice that the pattern with which the expression object is looked up is a reifier pattern.

Agora Receiverless Reifier Message Passing

```
class ReifierPatternExpression extends AbstractExpression
  abstract class attributes
   StandardClient
  instance variables
   pattern:AbstractReifierPattern
  methods
   concrete eval:context result AbstractMetaObject
     local variables
       reifierArguments:ArgumentList
       reifierInstance:AbstractMetaObject
     for each argument in pattern do
        reifierArguments add:(argument asExplicit)

     reifierInstance :=
         context private
           send:(pattern asCategory:context)
           client:(StandardClient arguments:reifierArguments)
     ^reifierInstance asImplicit eval:context
endclass
```

The following example shows how to declare reifier expression objects. The definition of a reifier object is that of any explicitly encoded expression object. In this case a tracing expression is implemented that defines how to make a tracing meta-object. The mapping between the '**trace**:' reifier pattern and the trace expression object takes the form of a private method declaration. This private method is executed each time the trace reifier is evaluated. Notice that it is declared with a reifier pattern as head.

```
MakeTracingObject Mixin:
  [ who define ;
    who:w CloningMethod: [who <- w ] ;
    send:pattern client:client
      method:
        [ … put trace information on screen …
          (who send:pattern client:client) return
        ] ] ;
TracingObject define: Object MakeTracingObject ;
```

```
MakeTracingExpression Mixin:
    [  exp define ;
       trace:e CloningMethod: [exp <- e ] ;
       eval:context Method:
          [ (TracingObject who:(e eval:context)) return ] ];
TracingExpression define: Object MakeTracingExpression ;

(trace:e) privateMethod: TracingExpression trace:e ;

p define: Point ;
p x:3 y:4 ;
p <- trace:p ;
--- from here on all messages to p will be traced
```

With the above mechanism local extensions to Agora can be made at run-time. A program is evaluated under a local extension of the class hierarchy. Such a local extension takes the form of a set of reifier declarations.

Other such mechanisms can be devised (e.g. global reifier declarations, recursive reifier declarations). They all share the property that programs can, during execution time, extend the set of reifiers that can be used. Obviously, in some cases this ability must go hand in hand with a mechanism for handling reflective overlap. The notion of meta-contexts can be reused for these purposes.

Reifier Methods

The evaluation of reifier expressions has been left open until now. All that was said is that somehow each time a reifier message is sent, the correct reifier method has to be executed. We are now ready to show how reifier messages can be interpreted as a special kind of messages. What is needed for this special interpretation is the notion of a linguistic symbiosis. Reifier messages are nothing but messages sent to converted expression objects. Besides passing all converted component expressions as arguments, the context also needs to be passed to the receiver expression object.

Agora Reifier Message Passing

```
class ReifierMessageExpression extends AbstractExpression
  abstract class attributes
    ReifierClient
  instance variables
    receiver:AbstractExpression,
    pattern:AbstractReifierPattern
  methods
    concrete eval:context result AbstractMetaObject
      local variables reifierArguments:ArgumentList
      for each argument in pattern do
         reifierArgumentsadd:(argument asExplicit)
      reifierArguments add:(context asExplicit)

      ^(receiver asExplicit)
          send:(pattern asCategory:context)
          client:(ReifierClient arguments:reifierArguments)
endclass
```

Thus, reifier declarations in Agora are nothing but special method declarations within expression objects. What differentiates a reifier from any other method is that it has an implicit context argument[2]. An example reifier declaration can be found below. It should be declared in a pattern class.

2 In fact message passing to explicitly referable objects must be adapted accordingly. Also the client that carries the reifier arguments must allow one extra argument.

```
(PrivateMethod:righthand)
    using:context
    reifier:
     [ context privateSlots add: (MethodSlot key:self value:righthand) ]
```

The implementation of the "**using**:**reifier**:" reifier simply adds a reifier slot to the public part of the object in which this declaration took place. A reifier slot differs from other method slots by the fact that it can handle hidden context arguments (the formal argument name of the hidden argument is stored in this slot).

```
class AbstractReifierPattern
  abstract class attributes
   ReifierSlot
  methods
   reifier using:contextPattern reifier:body
      result AbstractMetaObject
      using (declarationContext:StandardContext)
     declarationContext publicSlots add:
       (ReifierSlot key:self value:body using:contextPattern)
endclass
```

The disadvantage of reifier methods is that they necessarily lead to a more static form of reflection. Reifiers can be declared in newly created expression classes. The problem is that when a program is being executed all expression-objects are already instantiated. One either has to devise some mechanism whereby reifier methods can be added to already existing expression objects — for example in the form of some reclassification mechanism — or one falls back upon a more static form of reflection. Let us consider the latter.

Expression classes must be defined prior to using them in the construction of program trees. This leads us to a form of reflection highly inspired upon the pragmatics of the existing open implementation of Agora. In practice the open implementation of Agora is used as follows. First, new kinds of expressions, objects, etc. are constructed by inheriting from, and extending the existing class hierarchies. Then, these new classes are used by the programming environment (e.g. the parser) to construct a program tree. Finally this program tree is executed. The difference is that now expression classes can be expressed in Agora.

The advantage of the above approach is that all the power that comes with nested mixins can be used in structuring the class hierarchy of for example expressions. Particular extensions of Agora can be grouped together in mixins, and nesting and overriding of mixins can be used to record dependencies between such groups of extensions.

The disadvantage of this approach is that, if we literally follow the above, the entire programming environment for Agora must be made explicit in Agora. In some cases this may be just what we wanted (the Smalltalk programming environment for example is explicitly encoded in Smalltalk). In other cases this may be a dramatic overhead. Other solutions exist however. The techniques used in opening up the implementation of Agora's evaluator may equally well be applied to a programming environment. It is then only a question of making the interface of this open implemented programming environment available to the Agora programmer. For the time being, this issue remains open.

Another, more important disadvantage is that reflection, in the above, is reduced to a static mechanism. Although in a literal sense programs implicitly reflect during run-time, all extensions to Agora are made prior to running a program with this extended version of Agora.

5.3.1. Need for a More Fine-Grained Linguistic Symbiosis

The above reflection operators do not give full access to the open implementation of Agora. What is lacking is the ability to extend existing classes from the framework that constitutes Agora's implementation. Agora's implementation hierarchy must be made accessible from within Agora. In practice this class hierarchy can be made available as a library of mixins. For the expression hierarchy, for example, this means that the following library of nested mixins — of which the root mixins are applicable to the root object — are made available:

```
MakeAbstractExpression
  MakeMessageExpression
  MakeReifierMessageExpression
  MakeAggregateExpression
  MakeLiteralExpression
  MakePatternExpression
  MakeReifierPatternExpression
MakeAbstractPattern
  MakeUnaryPattern
  MakeOperatorPattern
  MakeKeywordPattern
MakeAbstractReifierPattern
  MakeUnaryReifierPattern
  MakeOperatorReifierPattern
  MakeKeywordReifierPattern
```

Similar libraries of mixins must be made available for all other class hierarchies in the implementation. Each of these mixins can then be used to extend the implementation hierarchy of Agora.

The symbiosis of Agora and its implementation language that was achieved in the previous section is not sufficient for this purpose. Given our goal, a more fine-grained symbiosis of Agora and its implementation language is called for. In particular, what is needed is that Agora objects can inherit from implicitly encoded objects. If we divide an object into sub-objects corresponding to the inheritance structure, then the following figure depicts what is needed.

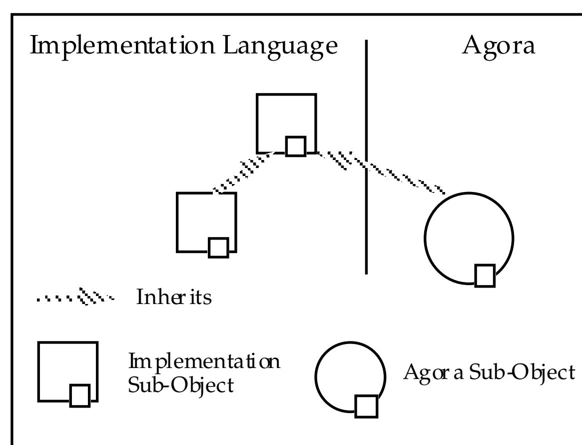


Figure 5.8

Since Agora's inheritance mechanism is prototype-, and mixin-based, technically, this can become non-trivial. We will not go into the technical details, but just give some indications of the problems involved.

Similarly to the symbiosis on the level of objects we need to identify how sub-objects are represented in the implementation. As we saw in chapter 4, sub-objects are represented internally by instances of concrete subclasses of 'AbstractInternalObject' that communicate with 'delegate:client:' messages. We can adopt the terminology of implicitly and explicitly encoded and referable sub-objects. We can also adopt a referent and representation relation on the level of sub-objects. This is shown in the next figure.

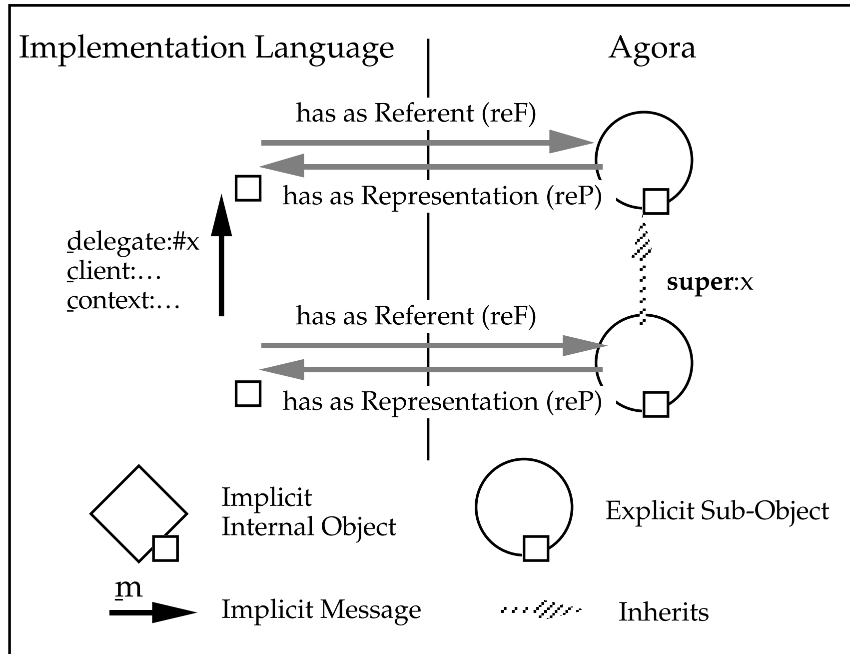


Figure 5.9

The problem now is that if we want explicitly encoded objects to inherit from implicitly encoded objects, then we need implicitly encoded objects that can be delegated to. Consider the following ideal situation. Presume that all implicitly encoded objects (e.g. 'AbstractExpression', 'Pattern', 'AbstractMetaObject', …) are such that messages can be delegated to them. In that case a variant of the 'asExplicit' conversion method can be devised such that implicit objects can be made explicit as sub-objects of explicit objects. This is depicted in figure 5.10.

So, in principle if implicitly encoded objects are objects that accept delegated messages then a symbiosis on the level of sub-objects between Agora and its implementations language can be realised. In practice however it is not possible to delegate messages to implicitly encoded objects. For this to be possible either the implementation language must be a language that supports message delegation or all objects that are part of the implementation must be encoded such that all methods have an extra set of delegation arguments (such as the 'self'). The former kind of languages were discarded in our analysis of object-orientation. The latter is an ad hoc implementation. An elegant solution to this problem remains an open issue.

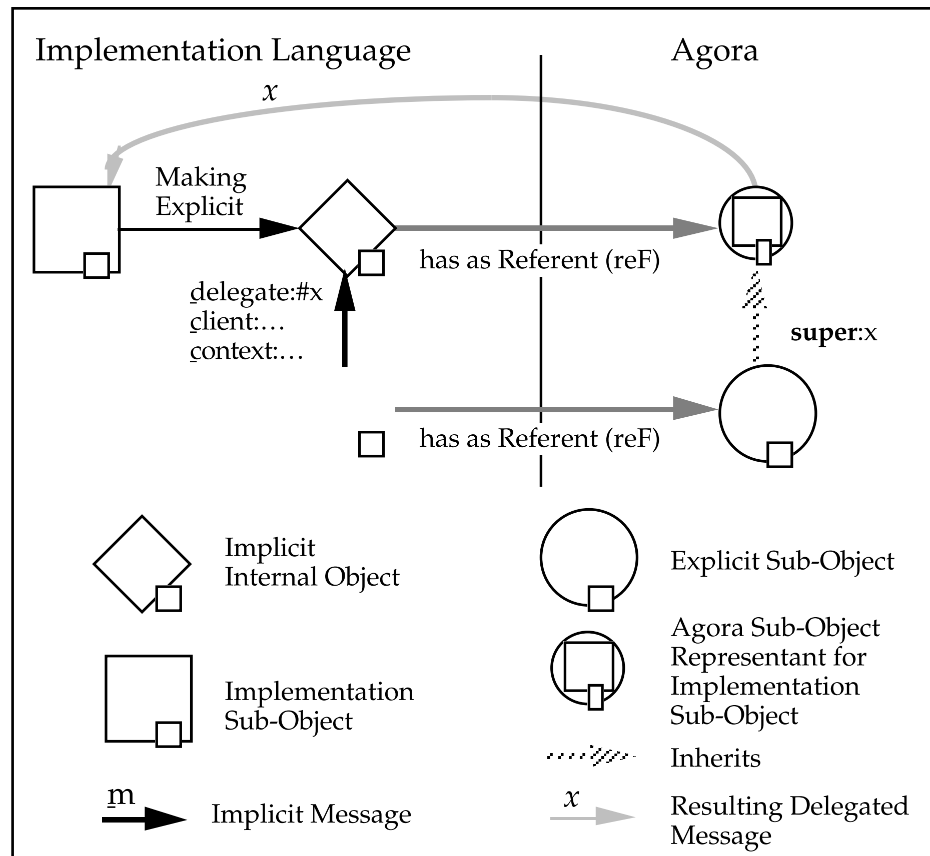


Figure 5.10

5.4 Conclusion and Open Issues

We conclude this section with some open issues regarding the introduction of reflection operators in an open implemented programming language.

Both of the above approaches to reflection in Agora (i.e. receiverless and other reifier messages) have their merits and their drawbacks. The second approach is too static but has the nice property of seamlessly integrating reflection and meta-programming. In the first approach reflection can be made more dynamic, but no explicit provisions are made for meta-programming.

Another apparent distinction is the management of extensions. The first approach is directed towards combining program pieces that are evaluated under different versions of the base language. The second approach may have better capabilities to combine different versions, but in its straightforward usage, programs, as a whole, must be expressed in the same extended language. In this situation differences may be trivially resolved, in the general case management of extensions of the base language may become an issue [Simmons&Friedman92] [Simmons&Friedman93].

We saw that for a considerably complex open implementation the choice of

reflection operators can become less than trivial. The design choices one has to make largely depend on the dynamic nature of reflection, and on how extensions of the base language are managed. One question that can be asked is whether reflection operators themselves can be used to introduce new reflection operators. It is trivial to see that all possible reflective facilities can be reconstructed in a reflective architecture (an example of a quote reifier is given below).

```
(AbstractExpression quote)
    using:context
    reifier: [ self return ]
```

More interestingly, in the case of Agora for example, the reifier declaration reifier itself can be reconstructed (meta-circularly):

```
(AbstractReifierPattern using:contextPattern reifier:body)
    using:declarationContext
    reifier:
      [declarationContext publicSlots add:
              (ReifierSlot  key:self
                            value:body
                            using:contextPattern) ]
```

It remains open to what extent and what useful reflective operators can be introduced reflectively.

Chapter 6

Related Work

6.1 Reflection and Open Systems

Open implementations were first discussed in [Rao90] [Rao91]. Rao defines what an open implementation is and describes an open implementation for a windowing system. Although Rao relates his work to what he calls implementational reflection, he does not give a thorough account of the relation between open implementations and reflective systems.

A more general introduction to the intuitions surrounding and motivations for investigating open implementations is given in [Kiczales92].

The most widely accepted account of computational reflection in programming languages was given by Smith in [Smith82] [Smith84]. The intuitions behind and motivations for the introduction of reflection are adopted in this text. Still, in Smith's and later Maes's [Maes87] account of reflection the notion of meta-circular interpreters plays an essential role. We motivated another approach to reflection where meta-circular interpreters are substituted by open implemented language processors.

In [Smith82] the notion of towers of meta-circular interpreters is used, and implementation techniques for such towers are discussed in [des Rivières&Smith84]. A comparison with towers of open implementations is given in the appendix.

6.2 Object-Oriented Reflection

3-KRS, ObjVlisp and Others

3-KRS [Maes87] is probably the first structured account of how to build an object-oriented programming language with a reflective architecture. Although the merits of object-orientation — encapsulation, modularity and abstraction are cited — are acknowledged for reflective programming, and although a serious attempt was made to be as complete and uniform as possible in the self-representation — 3-KRS has for example meta-objects for both data and program objects —, the reflective architecture of 3-KRS is highly inspired by one particular implementation of one particular object-oriented language (KRS).

3-KRS and Agora seem to have some commonalities in the choice of what must be represented in the self-representation, i.e. meta-objects, expressions, slots, etc. The important difference lies in *how* meta-objects, expressions, slots and so on are represented. As has been amply discussed in the previous chapters compositionality and full abstraction play an important role in Agora in the representation of objects and expressions. Equally important is the fact that for Agora the choice of what is made part of the self-representation and how the self-representation is structured is motivated by an in-depth study of the major design issues involved in designing an object-oriented programming language.

It should also be noted that since the connection between open implementations, frameworks and reflection had not been made at the time 3-KRS was being defined, 3-KRS does not have the layered structure of specialisation of language concepts as is the case for Agora.

ObjVlisp as presented in [Cointe87] discusses the introduction of reflection in a class-based language. It is based on a set of postulates, for example stating that the only protocol to activate an object is message passing. Still, it is easily shown that meta-classes alone are not enough to obtain a fully reflective programming language.

Other efforts are worth mentioning in this context. In [Ibrahim&Cummins88] a reflective programming language is developed with a strong emphasis on the representation of expressions. In [Jagannathan&Agha92] a reflective model of inheritance is presented. It is shown how, by reification of name-binding environments, reflection can be used in providing modularity structures in programming languages. In [Ferber89] various approaches to reflection in class-based languages are described. The approaches described vary in the different aspects of the computation that are reified. It should be mentioned that at least the reification of meta-objects and the reification of messages is covered by the reflection operators of Agora. Both [Mulet&Cointe93] and [Malenfant,Cointe&Dony91] study the issue of reflection in a prototype-based programming language. Although in some ways they have similar goals as ours — for example, defining inheritance or delegation as a specialisation —, but they do so by reifying meta-objects in a non abstract manner.

Finally, it should be noted that Agora's reifier expressions (i.e. reifier messages, receiverless reifier messages and reifier aggregates) are based on reifier functions from 3-Lisp [Smith84]. To the author's knowledge no other reflective object-oriented programming languages exist that are so heavily based on reifier expressions as Agora.

Metaobject Protocols

A major thrust in current day research on object-oriented reflection is directed towards metaobject protocols, of which the CLOS metaobject protocol (MOP) [Kiczales,des Rivières&Bobrow91] is the most prominent. Our work is related most to the work on MOPs. It shares the concerns of defining open languages by means of well-documented class hierarchies (e.g. [Kiczales&Lamping92]).

The CLOS MOP defines an open implementation for the Common Lisp Object System. This object system is heavily based on 'generic functions'. Accordingly the object model that is made explicit in the CLOS MOP is a model based on overloaded functions. This results in an entirely different view on object-orientation than the one adopted in this dissertation.

Although object-oriented software engineering practices — protocol design and documentation — play a prominent role in the work on metaobject protocols, it is difficult to say whether the CLOS MOP defines a true open design. Therefore a more thorough account of the design issues of object-oriented languages based on overloaded functions would be needed.

The differences, with our work, in handling reflection must also be pointed out. In [Kiczales,des Rivières&Bobrow91] the MOP is meta-circularly defined. This obviously leads to various circularities. These circularities are resolved in an actual implementation (behind the scenes). Resolving these circularities is an essential step in making a running reflective CLOS MOP. In our work we do not attempt to give a meta-circular definition of our framework. But, on the other hand, reflection is added as a full-fledged specialisation of the framework. The notion of a linguistic symbiosis is an essential ingredient in this process. We do not have a meta-circular definition of for example meta-objects, but what we do have is a way to make meta-objects explicitly referable (exactly this kind of differences was our motivation for making the terminological distinction between explicitly encoded and explicitly referable objects).

Linguistic Symbiosis of Object-Oriented Languages

The notion of a linguistic symbiosis between a programming language and its implementation language and its role in implementing a reflective programming language was first discussed in [Ichisugi&al.92]. There a description is given of a symbiosis, on the level of objects, between RbCl (a concurrent reflective object-oriented programming language) and C++ (the implementation language).

6.3 Object-Oriented Systems

The most important sources of inspiration for our analysis of object-oriented programming language concepts are [Wegner90] [Stein,Lieberman&Ungar89] [Dony,Malenfant&Cointe92] and [Cardelli88] [Cardelli&Wegner86] [Cook89] [Ghelli90] [Lieberman86] [Wegner&Zdonik88] for models of object-orientation and inheritance. For an alternative look on object-orientation based on overloaded functions the reader is referred to [Castagna&al.92] [Chambers92].

6.3.1 Object-Oriented Frameworks

In [Johnson&Foote88] and [Johnson&Russo91] a general description is given of the role of reuse and abstract classes in object-oriented programs. In [Johnson&Foote88] the emphasis is put on how to design (abstract) classes so that they become reusable. We extended the notion of abstract classes to include classes with virtual class attributes. In [Johnson&Russo91] the issue of how entire designs are reused is illustrated by means of a larger example. The distinction we made between a framework's internal and external interface comes originally from [Deutsch87]. In this dissertation object-oriented frameworks were studied in the context of open designs. We emphasised different kinds of specialisations of object-oriented frameworks that preserve the design of the framework. The relation between open implementations and object-oriented frameworks has already been noted in [Holland92]. In [Helm&al90] and [Holland92] a description is given of contracts — high level constructs for the specification of interactions among groups of objects — and how to refine and reuse contracts in a conforming way. We gave a more intuitive explanation of how to specialise a framework entirely based on substitutability [Liskov87] [Wegner&Zdonik88] of objects. Of course this can not substitute a formal description of specifying behaviour compositions in frameworks, but should rather be an intuitive basis for it.

6.3.2 Mixin-Based Inheritance

Our work on mixin-methods is an extension of mixin-based inheritance as was introduced in [Bracha&Cook90]. Mixin-based inheritance is an inheritance mechanism that is directly based on the model of inheritance as an incremental modification mechanism. It makes wrappers and wrapper application explicit.

Our work is based on a generalisation of mixin-based inheritance. Firstly, our mixins are based on a more general form of wrappers, where wrappers can have multiple parents. The notion of wrappers with multiple parents has already been pointed out in [Cook89]. Secondly, we extend the use of mixins to object-based inheritance. This sort of object-based inheritance is similar to implicit anticipated delegation [Stein,Lieberman,Ungar89], the resulting objects are comparable to split objects of [Dony,Malenfant,Cointe92]. Furthermore, we add dynamic application of mixins, mixins as attributes and the resulting scope rules for nested mixins. The extra polymorphism gained by viewing mixins as attributes seems to us an important enhancement to mixin-based inheritance. In contrast with [Bracha&Cook90] the mixin-methods used in Agora remain dynamically typed at the moment.

The relation to nested classes [Buhr&Zarnke88][Madsen87] has been discussed in the dissertation. The correspondence between so called virtual superclasses [Madsen&Møller-Pedersen89] in BETA and mixins has already been noted [Bracha92]. The same remarks as in the previous paragraph apply to the relation between mixin-methods and virtual superclasses.

Having mixins as instance attributes is very similar to "enhancements" described in [Hendler86]. We agree that being able to associate functionality with instances rather than classes has several advantages. The advantages of dynamic classification have also been discussed in the classifier approach of [Hamer92]. Both approaches lack the equivalent of late binding of mixin attributes. Our approach lacks the equivalent of having classifiers as first class values (which would amount to first class mixin "patterns"). Traces [Kiczales93] and first class mixin patterns are comparable mechanisms, they both are a step in solving the "make isn't generic" problem. First class mixin patterns are more general but also more primitive.

Our analysis of the problems involved in multiple inheritance is a résumé of [Snyder87], [Knudsen88] and [Carré,Geib90].

Although it is shown in [Bracha&Cook90] that mixin-based inheritance subsumes different inheritance strategies, no complete analysis is given on its relation with multiple inheritance. Other approaches that share our view of fragmenting the functionality of inheritance, can be found in [Bracha92], [Hauck93], and [Hense92]. Neither of them gives a complete analysis of how the multiple inheritance problems are addressed in it. In [Bracha92] name collisions are resolved with renaming. Neither of them supports object-based inheritance, which is an important part of our solution to multiple inheritance problems.

Chapter **7**

Conclusion

In this dissertation we showed what it means for an object-oriented programming language to have an open design. The role of object-oriented frameworks, full abstraction and compositionality in this was shown. We discussed how frameworks can be used in expressing open designs. A framework for an object-based, and later on for an object-oriented programming language was presented. In this framework the major components of an object-oriented programming language were represented as abstract classes. A compositional and extensible representation of expressions and expression evaluation was presented. We showed that reifier expressions are an essential ingredient in making program representations extensible. Furthermore, a fully abstract object representation was discussed. This object representation was shown to represent encapsulated, polymorphic objects with a well-defined behaviour. It was shown that this particular fully abstract representation of objects could be specialised to include an object-based inheritance mechanism based on mixin-methods. This inheritance mechanism was shown to be a particularly useful one in constructing and controlling the evolution of multiple inheritance hierarchies. The resulting framework was used to express two important programming languages. The first language was based on a calculus for objects. The second was a full-fledged programming language with as sole built in construct that of message passing. It was shown that, while remaining in the constraints of the framework, fairly sophisticated extensions to this programming language could be expressed.

In a larger context we showed that reflective programming languages can be defined in a less ad hoc way than is usually the case. The importance of open designs in the context of defining reflective programming languages was made apparent. It was shown that opening up a programming language does not necessarily mean that all control about the language concepts, that are made explicit, is lost. The relation between open designs and reflective systems was discussed in general, and illustrated by making the above discussed framework reflective. In general terms we can say that open designs take up the role meta-circular interpreters conventionally play in reflective programming languages.

7.1 Contributions

The major contribution of this dissertation is showing that merely opening up an implementation of a programming language is not sufficient as a basis for defining expressive and safe reflective programming languages. The concept of open designs was proposed as an alternative. In particular an intuitive explanation of object-oriented frameworks and operations on object-oriented frameworks that preserve the major design issues was given. Furthermore, the importance of full abstraction and compositionality was discussed.

The second contribution is an alternative account of reflection based on open systems. It was shown how a computational system can be made reflective by opening up the meta-system, and by providing effective access to the resulting meta- and object-level interfaces of the meta-system. The properties of open systems with reflective potential have been discussed. A thorough account of a linguistic symbiosis — an essential ingredient in constructing an open system with reflective potential — has been given.

Apart from these general contributions more technical results were obtained.

The definition of a particular open design for object-based and object-oriented programming languages was given. This open design was based on respectively the notion of encapsulated polymorphic objects with a well-defined behaviour and mixin-methods. It was shown how a framework for object-oriented programming languages can be seen as a specialisation of a framework for object-based programming languages. The role of message passing was stressed, not only as a basic control structure, but also for making an extensible expression hierarchy and for making objects extensible. Reifier messages were used to make the expression hierarchy extensible. Mixin messages played an important role in extending objects.

A calculus for objects that features an explicit encapsulation operator and message passing as a primitive, atomic control structure was presented. Although not fully formalised yet, it provided a good basis for defining the basic structures of our framework.

A full fledged object-oriented programming language was presented (Agora). What distinguishes Agora from other object-oriented programming languages with a reflective architecture is its simple design. It features message passing as its only built in language construct. Furthermore a vanilla flavour of Agora was defined in the form of a standard set of reifier expressions. This standard set of reifiers was characterised by its usage of mixin-methods for inheritance.

Mixin-methods were shown to be an object-based inheritance mechanism that combines the advantages of class-based inheritance with the advantages of object-based delegation. It was shown that for an inheritance mechanism based on mixin-methods, inheritance can be entirely encapsulated. Motivated by a thorough analysis of the problems involved in constructing inheritance hierarchies a proposal was made for a generalisation of mixin-methods. It was shown that generalised mixin-methods can be used to express an entire range of multiple inheritance hierarchies in a simple and effective way. Furthermore it was shown how mixin-methods can be used in controlling the evolution of inheritance hierarchies.

7.2 Future work

Expressing Open Designs
In this dissertation we investigated the importance of open designs for reflective programming languages. The converse question, the importance of reflection in expressing open designs has been left open. As we saw in the section on object-oriented frameworks, the major design issues of an object-oriented framework are made explicit in the form of abstract classes. The programming language used to express a framework must support, for this reason, abstract classes as a language concept. In this text we stressed the importance of abstract attributes for this purpose. This is only one aspect of expressing object-oriented frameworks. Other aspects include information about the sharing structure of objects, information about the order of method invocations, information about the possible evolution of the class hierarchy — in the form of classifiers, for example — and many more. Obviously it is impossible to give a complete list of all such language concepts that must be provided, let alone to construct a single monolithic programming language that incorporates this list. Reflection and programming languages with an open design could prove to be an essential ingredient in realising and expressing open designs.

Open implementations, object-oriented frameworks, reflective systems are relatively young research areas. Formalisation of the major issues of each of them is an ongoing research topic. The work presented in this text is no exception to this. Although at some places formal techniques were used and hinted at, the main part of the work is in need of a more formal treatment. Formal semantics for conventional programming languages have been thoroughly studied. To a certain extent the formal semantics of reflective programming languages, also, has been investigated. Still, what is needed for a programming language with an open design is a formal description of a design space of programming languages. To the author's knowledge this issue has been left unexplored in the research community. What can be expected is that formal work on object-oriented frameworks will play an important role in this. However, formal descriptions of object-oriented frameworks too are still a hotly debated research topic.

A Model of Objects Based on Atomic Message Passing
In our analysis of object-orientation we came to the conclusion that a theory of object-orientation can be based upon the notion of encapsulated polymorphic objects with a well-defined behaviour. Subsequently a calculus was presented that featured objects of the above kind. Other such calculi for objects are being proposed in the literature. It remains to be investigated how our calculus relates to the others.

A Model of Inheritance Based on Mixin-Methods
Mixin-methods play an important role in this dissertation. As we already said, in the absence of mixin-methods, one has essentially two choices in picking an inheritance mechanism. Either class-based inheritance or classless delegation can be chosen. The one involves a notion different from objects, the other a notion different from normal message passing. Mixin-methods have been shown to solve this dilemma. Still, some work needs to be done. A more formal treatment of mixin-methods needs to be given. This could, for example, take the form of giving a denotational semantics of mixin-methods. Another remark is that mixin-methods are in some cases too limited. With mixin-methods an object must incorporate, in advance, all its extensions. Sometimes an object must be extended in a way that can not be predicted. Therefore mixin-methods must be generalised.

Implementation Issues
The Agora framework as presented in the text has been given a full implementation. This implementation is based on an interpretative approach. Furthermore efficiency issues are totally neglected. The problems involved in defining efficient implementations of reflective and open programming languages have been studied elsewhere. However, it needs to be investigated to what extent compositionality and full abstraction aid in optimising efficiency of reflective languages.

Bibliography

[Abadi&Cardelli94] M. Abadi and L. Cardelli: **A Theory of Primitive Objects**, DEC Tech Report, 1994.

[Abelson&Sussman84] H. Abelson, G. J. Sussman, and J. Sussman: **Structure and Interpretation of Computer Programs**, MIT Press, 1984.

[Agha86] G. Agha: **Actors: A Model of Concurrent Computation in Distributed Systems**, MIT Press, 1986.

[Agha90] G. Agha: **The Structure and Semantics of Actor Languages**, In Foundations of Object-Oriented Programming Languages, Proceedings of REX School/Workshop, pp. 1-59, LNCS 489, Springer-Verlag 1990.

[America87] P. America: **POOL-T: A Parallel Object-Oriented Language**, In A. Yonezawa, M. Tokoro (Eds.), Object-Oriented Concurrent Programming, pp. 199-220, MIT Press 1987.

[Asai,Matsuoka&Yonezawa93] K. Asai, S. Matsuoka, and A. Yonezawa: **Duplication and Partial Evaluation to Implement Reflective Languages**, In Informal Proceedings of the OOPSLA'93 Workshop on Object-Oriented Reflection and Metalevel Architectures, October 1993.

[Bobrow&al.86] D. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, F. Zdybel: **CommonLoops Merging Lisp and Object-Oriented Programming**, In Proceedings of OOPSLA'86, pp17-29.

[Bracha&Cook90] G. Bracha and W. Cook: **Mixin-based Inheritance**, In Proc. of ACM Joint OOPSLA/ECOOP'90 Conference Proceedings, pp.303-311, ACM Press 1990.

[Bracha&Lindstrom92] G. Bracha and G. Lindstrom: **Modularity meets Inheritance**, In Proc. of IEEE Computer Society International Conference on Computer Languages, pp.282-290, 1992.

[Bracha92] G. Bracha: The Programming Language Jigsaw: **Mixins, Modularity and Multiple Inheritance**, PhD thesis, Department of Computer Science, University of Utah, March 1992.

[Buhr&Zarnke88] P.A. Buhr, C.R. Zarnke: **Nesting in an Object-Oriented Language is NOT for the Birds**. In Proc. of ECOOP'88 European Conference on Object-Oriented Programming, pp.128-143, Springer-Verlag 1988.

[Canning,Cook,Hill&Olthoff89] P. S. Canning, W. R. Cook, W. L. Hill, W. G. Olthoff: **Interfaces for Strongly-Typed Object-Oriented Programming**, In OOPSLA'89 Conference Proceedings, ACM Sigplan Notices Vol. 24(10), pp. 457-468, ACM Press 1989.

[Cardelli&Mitchell89] L. Cardelli & J. C. Mitchell: **Operations on Records**, Proceedings on Mathematical Foundations of Programming Semantics, LNCS 442

[Cardelli&Wegner86] L. Cardelli and P. Wegner: **On Understanding Types, Data Abstraction and Polymorphism**, Computing Surveys Vol 17(4), pp. 471-522, 1988.

[Cardelli88] L. Cardelli: **A semantics of multiple inheritance**, Information and Computation 76, pp. 138-164, 1988

[Carré&Geib90] B. Carré and J. Geib: **The Point of View notion for Multiple Inheritance**. In Proc. of ACM Joint OOPSLA/ECOOP'90 Conference Proceedings, pp.312-321, ACM Press 1990.

[Castagna&al.92] G. Castagna, G. Ghelli&G. Longo: **A Calculus for Overloaded Functions with Subtyping**, In ACM Conference on Lisp and Functional Programming, pp.182-182, ACM Press, 1992.

[Chambers92] C. Chambers: **Object-Oriented Multi-Methods in Cecil**, ECOOP'92 European Conference on Object-Oriented Programming, Proceedings, Ed. O. Lehrmann Madsen. Springer-Verlag (615), pp33-56, 1992.

[Chambers93] C. Chambers: **Predicate Classes**, ECOOP'93 European Conference on Object-Oriented Programming, Proceedings, Ed. O. Nierstaz, Springer-Verlag (707), pp268-296, 1993.

[Chomsky56] N. Chomsky: **Three Models for the Description of Language**, IEEE Trans. Information Theory, Vol. 2, pp. 113-124, 1956.

[Codenie,Steyaert,Lucas92] W. Codenie, P. Steyaert, C. Lucas.: **Nested Mixins in AGORA**, Position Paper to the ECOOP '92 Workshop on Multiple Inheritance and Multiple Subtyping, pp. 29-31, 1992.

[Cointe87a] P. Cointe: **The ObjVlisp kernel: A reflexive Lisp architecture to define a uniform object-oriented system**, In P. Maes and D. Nardi (eds.), Meta-Level Architectures and Reflection, pp. 155-176, North-Holland, 1987.

[Cointe87b] P. Cointe: **Metaclasses are First Class: the ObjVlisp Model**, In Proceedings of OOPSLA'87 Conference, Volume 22, pp. 156-167, SIGPLAN Notices, ACM Press, October 1987.

[Cook&Palsberg89] W. Cook and J. Palsberg: **A Denotational Semantics of Inheritance and its Correctness**, In OOPSLA'89 Conference Proceedings, ACM Sigplan Notices Vol. 24(10), pp433-443, ACM Press 1989.

[Cook89] W. Cook: **A denotational semantics of Inheritance**, PhD thesis, Department of Computer Science, Brown University, May 1989.

[Cook90] W. Cook: **Object-Oriented Programming Versus Abstract Data Types**, In Foundations of Object-Oriented Programming Languages, Proceedings of REX School/Workshop, pp. 151-178, LNCS 489, Springer-Verlag 1990.

[Dami93a] L. Dami: **Extensible Lambda Expressions: A Lambda Calculus with Names, Combinations and Alternations**, Technical Report, University of Geneva, 1993

[Dami93b] L. Dami: **The HOP Calculus**, Technical Report, University of Geneva, 1993

[Dami94] L. Dami: **Named Parameters Subsume Records and Variants**, Submitted to OOPSLA '94, University of Geneva, 1994

[De Hondt93] K. De Hondt: **A Customizable, Ergonomic, Hybrid Structure-Oriented Editor**, Master Thesis, Vrije Universiteit Brussel, August 1993.

[De Volder&Steyaert94] K. De Volder, and P. Steyaert: **Construction of the Reflective Tower Based on Open Implementations**, VUB-PROG Technical Report, 1994.

[des Rivières&Smith84] J. des Rivières and B. C. Smith: **The Implementation of Procedurally Reflective Languages**, In Proceedings of the ACM Conference on LISP and Functional Programming, pp. 331-347, ACM 1984.

[Deutsch87] L. P. Deutsch: **Levels of Reuse in the Smalltalk-80 Programming System**, In Peter Freeman (Ed.) Tutorial: Software Reusability, IEEE Computer Society Press, 1987.

[Dony,Malenfant&Cointe92] Christophe Dony, Jacques Malenfant, Pierre Cointe: **Prototype-Based Languages: From a New Taxonomy to Constructive Proposals and Their Validation**, In Proceedings of the OOPSLA'92 Conference, ACM Sigplan Notices 27(10), pp.201-217, ACM Press 1992.

[Donzeau-Gouge&al.84] V. Donzeau-Gouge, G. Kahn, B. Lang, B. Mélèse: **Document Structure and Modularity in Mentor**, Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Sigplan Notices Vol 9(3), May, 1984.

[Ellis&Stroustrup90] M. A. Ellis and B. Stroustrup: **The Annotated C++ Reference Manual**, Addison-Wesley, 1990.

[Ferber89] J. Ferber: **Computational Reflection in Class based Object Oriented Languages**, In OOPSLA'89 Conference Proceedings, ACM Sigplan Notices Vol. 24(10), pp. 317-326, ACM Press 1989.

[Frege92] G. Frege: **Über Sinn und Bedeutung**, (On Sense and Meaning), Zeitschrift für Philosphie und Philosophische Kritik, 100:25-50, 1892. Englisch translation in Readings in Philosophical Analysis (eds. H. Feigl and W. Sellars), pp. 85-102, Appleton-Century-Crofts, New York (1949), and Translations from the Philosophical Writings of Gottlob Frege (eds. P. T. Geach and M. Balck), pp. 56-78, Blackwell, Oxford (1960).

[Friedman&Wand84] D. P. Friedman, and M. Wand: **Reification: Reflection without Metaphysics**, In Proceedings of the 1984 Conference on Lisp and Functional Programming, pp. 348-355, 1984.

[Gamma&al.93] E. Gamma, R. Helm, R. Johnson, J. Vlissides: **A Catalog of Object-Oriented Design Patterns**, Technical Report, Draft of September 7, 1993, Book forthcoming.

[Ghelli90] G. Ghelli: **Modelling Features of Object-Oriented Languages in Second Order Functional Languages with Subtypes**, In Foundations of Object-Oriented Programming Languages, Proceedings of REX School/Workshop, pp. 311-340, LNCS 489, Springer-Verlag 1990.

[Ghelli91] G. Ghelli: **A Static Type System for Message Passing**, In Proc. of OOPSLA'91 Conference, ACM Sigplan Notices, Volume 26, No 11, pp. 129-145, ACM Press 1991.

[Goldberg&Robson89] A. Goldberg, and D. Robson:**Smalltalk-80, The Language**, Addison-Wesley Publishing Company, Reading Massachusetts, 1989.

[Hamer92] J. Hamer: **Un-Mixing Inheritance with Classifiers**, In Multiple Inheritance and Multiple Subtyping: Position Papers of the ECOOP'92 Workshop W1, pp. 6-9, Utrecht, the Netherlands, June/July, 1992. Also available as working paper WP-23, Markku Sakkinen (Ed.), Dept of Computer Science and Information Systems, University of Jyvaskylä, Finland, May 1992.

[Hauck93] F. Hauck: **Inheritance Modeled with Explicit Bindings: An Approach to Typed Inheritance**. In Proceedings of the OOPSLA '93 Conference ACM Sigplan Notices, pp. 231-239, ACM Press 1993.

[Hedin89] Görel Hedin: **An Object-Oriented Notation for Attribute Grammars**, Proceedings of the European Conference of Object-Oriented Programming (ECOOP'89), Nottingham U.K, 1989.

[Helm&al90] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay: **Contracts: Specifying Behavioural Composition in Object-Oriented Systems**, In Proceedings of the OOPSLA-ECOOP'90 Conference, ACM Sigplan Notices 25(10), pp.169-180, ACM Press 1990.

[Hendler86] J. Hendler: **Enhancement for Multiple Inheritance**. In Proc. of Object-Oriented Programming Workshop 86, Sigplan Notices Vol 21 (10), pp.98-106, October 1986.

[Hense92] A.V. Hense: **Denotational Semantics of an Object-oriented Programming Language with Explicit Wrappers**. Formal Aspects of Computing (1992) 3:1-000.

[Hofmann&Pierce92] M. Hofmann & B. C. Pierce: **An abstract view of objects and subtpying**, Technical Report ECS-LFCS-92-226, University of Edinburgh, 1992

[Holland92] Ian M. Holland: **Specifying reusable components using Contracts**, In ECOOP'92 European Conference on Object-Oriented Programming, Proceedings, Ed. O. Lehrmann Madsen. Springer-Verlag (615), pp.287-308, 1992.

[Ibrahim&Cummins88] M. H. Ibrahim and F. A. Cummins: **KSL: A Reflective Object-Oriented Programming Language**, In Proceedings of the IEEE Computer Society International Conference on Computer Languages, pp. 186-193, 1988.

[Ichisugi&al.92] Y. Ichisugi, S. Matsuoka, and A. Yonezawa: **RbCl: A Reflective Concurent Language without a Run-time Kernel**, In Informal Proc. ofThe ECOOP'92 Object-Oriented Reflection and Metalevel Architectures , Utrecht, the Netherlands, 30 June, 1992.

[Jagannathan&Agha92] S. Jagannathan and G. Agha: **A Reflective Model of Inheritance**, In ECOOP'92 European Conference on Object-Oriented Programming, Proceedings, Ed. O. Lehrmann Madsen. Springer-Verlag (615), pp350-372, 1992.

[Johnson&Foote88] R. E. Johnson, B. Foote: **Designing Reusable Classes**, Journal of Object-Oriented Programming, 1(2), pp. 22-35, 1988.

[Johnson&Russo91] R. E. Johnson, Vincent F. Russo: **Reusing Object-Oriented Design**, University of Illinois tech report UIUCDCS 91-1696,1991.

[Johnson90] R. E. Johnson: Position Statement in [Wirfs-Brock90].

[Kamin88] S. Kamin: **Inheritance in SMALLTALK-80: A Denotational Definition**, In ACM Symposium on Principles of Programming Languages, January 1988.

[Kiczales&Lamping92] G. Kiczales and J. Lamping: **Issues in the Design and Specification of Class Libraries**, In Proc. of the OOPSLA'92 Conference, ACM Sigplan Notices 27(10), pp.435-451, ACM Press, 1992.

[Kiczales&Paepcke93] G. Kiczales, and A. Paepcke: **Metaobject Protocols and Open Implementations**, Tutorial Notes, OOPSLA'93, 1993.

[Kiczales,des Rivières&Bobrow91] G. Kiczales, J. des Rivières, and D. G. Bobrow: **The Art of the Metaobject Protocol**, The MIT Press, Cambridge, Massachusetts, 1991.

[Kiczales92] G. Kiczales: **Towards a New Model of Abstraction in the Engineering of Software**, In Proceedings of IMSA'92 Conference.

[Kiczales93] G. Kiczales: **Traces (A Cut at the "Make Isn't Generic" Problem**, In Proceedings of First International Symposium on Object Technologies for Advanced Software (ISOSTAS), S. Nishio and A. Yonezawa (Eds.), LNCS Vol. 742, 1993.

[Knudsen88] J. Lindskov Knudsen: **Name Collision in Multiple Classification Hierarchies**, In Proc. of ECOOP'88 European Conference on Object-Oriented Programming, pp. 93-109, Springer-Verlag 1988.

[Kristensen&al.87] B. B. Kristensen, O. L. Madsen, B. Møller-Pedersen, and K. Nygaard: **The Beta Programming Language**, In Bruce Shriver and Peter Wegner (Eds.) Research Directions in Object Oriented Programming, pp. 7-48, MIT Press 1987.

[LaLonde,Thomas&Pugh86] W. R. LaLonde, D. Thomas, J. R. Pugh: **An Exemplar Based Smalltalk**, , In Proceedings of OOPSLA'86 Conference, ACM Sigplan Notices, 21(11), pp. 322-330, ACM Press, 1986.

[Lamping93] J. Lamping: **Typing the Specialization Interface**, In Proc. of the OOPSLA'93 Conference, ACM Sigplan Notices, pp.201-214, ACM Press, 1993.

[Lieberman86] H. Lieberman: **Using Prototypical Objects to Implement Shared Behaviour in an Object-Oriented System**, In Proceedings of OOPSLA'86 Conf., ACM Sigplan Notices, 21(11), pp. 214-223, ACM Press, 1986.

[Linton&al.89] Mark A. Linton, John M. Vlissides, and Paul R. Cadler: **Composing user interfaces with Interviews**, IEEE Computer, 22(2), pp.8-22, February, 1989.

[Liskov87] Barbara Liskov: **Data Abstraction and Hierarchy**, OOPSLA'87 addendum to the proceedings, Leigh Power, Zvi Weiss (eds.), available as Sigplan Notices Special Issue Vol. 23(5) May 1988, pp.17-34.

[Madsen&Møller-Pedersen89] O. L. Madsen B. and B. Møller-Pedersen. **Virtual Classes, A Powerful Mechanism in Object-Oriented Programming**, In Proc. of ACM Conf. on Object-Oriented Programming, Languages, and Systems, pp.397-406, ACM Press 1989.

[Madsen&Nørgaard88] O. L. Madsen, C. Nørgaard: **An Object-Oriented Metaprogramming System**, Proceedings of the 21 Annual Hawaii International Conference on System Sciences, January 1988. Vol II. pp.406-415.

[Madsen,Magnusson&Møller-Pedersen90] O. L. Madsen, B. Magnusson, B. Møller-Pedersen: **Strong Typing of Object-Oriented Languages Revisited**, In Proceedings of the OOPSLA/ECOOP'90 Conference, ACM Sigplan Notices 25(10), pp.140-150, ACM Press, 1990.

[Madsen87] O. L. Madsen: **Block Structure and Object-Oriented Languages**, Research Directions in Object-Oriented Programming B. Shriver and P. Wegner (eds), pp 113-128, MIT Press 1987.

[Maes87a] P. Maes: **Computational Reflection**, PhD Thesis, AI-Lab VUB, Technical Report 87-2.

[Maes87b] P. Maes: **Concepts and Experiments in Computational Reflection**, In Proceedings of OOPSLA'87 Conference, ACM Sigplan Notices 22, pp. 147-155, ACM Press, October 1987.

[Malenfant,Cointe&Dony91] J. Malenfant, P. Cointe and C. Dony: **Reflection in Prototype-Based Object-Oriented Programming Languages**, In Informal Proceedings of the OOPSLA'91 Workshop on Reflection and Metalevel Architectures, October, 1991.

[Malmkjær90] K. Malmkjær: **On Some Semantic Issues in the Reflective Tower**, In Fifth Conference on Mathematical Foundations of Programming Semantics, LNCS 442, Springer Verlag, 1990.

[McGregor&Korson93] J. D. McGregor&T. Korson: **Supporting Dimensions of Classification in Object-Oriented Design**, In Journal of Object-Oriented Programming, pp.25-30, SIGS Publication, February 1993.

[Mens,Mens&Steyaert94] T. Mens, K. Mens, and P. Steyaert: **OPUS: A Formal Approach to Object-Orientation**, Tech Report VUB-TINF-94-02, 1994

[Meyer88] B. Meyer: **Object-oriented Software Construction**, Prentice Hall, 1988.

[Milner91] R. Milner: **The Poliadic pi-calculus: A tutorial**, Technical Report ECS-LFCS-91-180, University of Edinburgh, 1991

[Mittal,Bobrow&Kahn86] S. Mittal, D. G. Bobrow, K. Kahn: **Virtual Copies: At the Boundary Between Classes and Instances**, In Proc. of the OOPSLA'86 Conf., pp 159-166. ACM Press, 1986

[Moon89] David A. Moon: **The COMMON LISP Object-Oriented Programming Language Standard**, Object-Oriented Concepts, Databases, and Applications, Won Kim and Frederick H. Lochovsky (Eds.), pp. 79-126, ACM Press 1989.

[Mulet&Cointe93] Philippe Mulet, Pierre Cointe: **Definition of a Reflective Kernel for a Prototype-Based Language**, In Proceedings of First International Symposium on Object Technologies for Advanced Software, S. Nishio and A. Yonezawa (Eds.), LNCS Vol. 742, pp. 128-144, 1993.

[Nierstrasz92] O. Nierstrasz: **Towards an Object Calculus**, ECOOP Workshop on Object-Based Concurrent Computing, LNCS 612, 1992

[Opdyke&Johnson90] W. F. Opdyke and R. E. Johnson: **Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems**, In Proceedings of Symposium on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA), September 1990.

[Opdyke92] W. F. Opdyke: Refactoring Object-**Oriented Frameworks**, PhD Thesis Univesity of Illinois at Urbana-Champaign, 1992.

[Palay&al88] A.J. Palay, W.J. Hansen, M.L. Kazar, M. Sherman, M.G. Wadlow, T.P. Neuendorffer, Z. Stern, M. Bade, and T. Petre: **The Andrew Toolkit – an Overview**, In, USENIX Assoc. Winter Conf., Dallas, 1988.

[Palsberg&Schwartzbach90] J. PalsBerg, M. I. Schwartzbach: **Type Substitutions for Object-Oriented Programming**, In Proceedings of the OOPSLA/ECOOP'90 Conference, ACM Sigplan Notices 25(10), pp.151-160, ACM Press, 1990.

[Rao90] R. Rao: **Implementational Reflection in Silica**, In Informal Proceedings of the ECOOP/OOPSLA'90 Workshop on Reflection and Metalevel Architectures, October, 1990.

[Rao91] R. Rao: **Implementational Reflection in Silica**, In ECOOP'91 Proceedings, Lecture Notes in Computer Science, P. America (Ed.), pp. 251-267, Springer-Verlag, 1991.

[Reddy88] U. S. Reddy: **Objects as Closures: Abstract Semantics of Object-Oriented Languages**, In Proc. of the ACM Conference on Lisp and Functional Programming, pp. 289-297, 1988.

[Revesz88] G.E. Revesz: **Lambda-calculus, combinators and functional programming**, Cambridge Tracts in Theoretical Computer Science 4, Cambridge University Press, 1988

[Reynolds75] J. C. Reynolds: **User Defined Types and Procedural Data Structures as Complementary Approaches to Data Abstraction**, In David Gries (Ed.), Programming Methodology, A Collection of Articles by IFIP WG2.3, pp. 309-317, Reprinted from S.A. Schuman (Ed.), New Advances in Algorithmic Languages 1975, INRIA, Rocquencourt, 1975, pp. 157-168.

[Ruf93] E. Ruf: **Partial Evaluation in Reflective System Implementations**, In Informal Proceedings of the OOPSLA'93 Workshop on Object-Oriented Reflection and Metalevel Architectures, October 1993.

[Schmidt86] D. A. Schmidt: **Denotational Semantics, A Methodology for Language Development**, Allyn And Bacon Inc, Massachusetts, USA, 1986.

[Schmucker86] Kurt J. Schmucker: **Object-Oriented Programming for the Macintosch**, Hayden Book Company, 1986.

[Shan&al.93] Y. Shan, T. Cargil, B.Cox, W. Cook, M. Loomis, A. Snyder: **Is Multiple Inheritance Essential to OOP**, In Proc. of OOPSLA'93 Conference, ACM Sigplan Notices, pp. 360-363, ACM Press, 1993.

[Simmons II&al.92] J. W. Simmons II, S. Jefferson and D. P. Friedman: **Language Extensions via First-class Interpreters**, Indiana University Computer Science Departement Technical Report #362, 1992.

[Simmons&Friedman92] J. W. Simmons II and D. P. Friedman: **A Reflective System is as Extensible as its Internal Representations: An Illustration**, Indiana University Computer Science Technical Report #362, 1992.

[Simmons&Friedman93] J. W. Simmons II and D. P. Friedman: **First-class Interpreters: Illustrating the Limits Imposed by Representation in a Reflective Language**, In Informal Proceedings of the OOPSLA'93 Workshop on Object-Oriented Reflection and Metalevel Architectures, October 1993.

[Smith82] B.C. Smith: **Procedural Reflection in Programming Languages**, PhD Thesis, Massachusetts Institute of Technology, available as MIT Laboratory of Computer Science Technical Report 272, Cambridge Massachusetts, 1982.

[Smith84] B.C. Smith: **Reflection and Semantics in Lisp**, Conf. Rec. of the 11th ACM Symposium on Principles of Programming Languages, pp. 23-35, January 1984.

[Smith86] B.C. Smith: **Varieties of Self-Reference**, Form: Reasoning about Knowledge, Proc. of the 1986 Conf., J. Halpren ed., Los Altos, CA, Morgan Kaufman, 1986.

[Snyder87] A. Snyder: **Inheritance and the Development of Encapsulated Software Components**, In Research Directions in Object-Oriented Programming, B. Shriver and P. Wegner (eds), pp 165-188, MIT Press 1987.

[Stein,Lieberman&Ungar89] L. Stein, H. Lieberman, D. Ungar: **A Shared View of Sharing: The Treaty of Orlando**, In Object-Oriented Concepts, Databases, and Applications, W. Kim and F.H. Lochovsky, pp. 31-48, ACM Press 1989

[Stein87] L. A. Stein: **Delegation is Inheritance**, In Proceedings of the OOPSLA'87 Conf., ACM Sigplan Notices, 22(12), pp. 138-146.

[Steyaert&al.93] P. Steyaert, W. Codenie, T. D'Hondt, K. De Hondt, C. Lucas, M. Van Limberghen: **Nested Mixin-Methods in Agora**, ECOOP '93 European Conference on Object-Oriented Programming, pp. 197-219, Springer-Verlag.

[Steyaert92] P. Steyaert: **Towards a Calculus for Objects and its Reflective Variant**, Extended Abstract (unpublished), printed at ECOOP '92 workshop on reflection and metalevel architectures, 1992

[Szyperski92] C. A. Szyperski: **Import Is not Inheritance – Why We Need Both: Modules and Classes**, In ECOOP'92 European Conference on Object-Oriented Programming, Proceedings, Ed. O. Lehrmann Madsen. Springer-Verlag (615), pp.19-32, !992.

[Teitelbaum&Reps81] T. Teitelbaum and T. Reps: **The Cornell Program Synthesizer: A Syntax-Directed Programming Environment**, Communications of the ACM, Vol 24(9), September 1981.

[Tennent91] R. D. Tennent, **Semantics of Programming Languages**, Prentice Hall, 1991.

[Ungar&Smith87] D. Ungar and R. B. Smith: **Self: The Power of Simplicity**, In Proceedings of OOPSLA'87 Conf. pp. 227-242, ACM Press,1987.

[Ungar,Chambers,Chang&Hölzle91] D. Ungar, C. Chambers, B-W. Chang, and U. Hölzle: **Organising Programs without Classes**, in Lisp and Symbolic Computation, Vol. 4 no. 3, 1991.

[Wand&Friedman88] M. Wand, and D. P. Friedman: **The Mystery of the Reflective Tower Revealed: A Nin-Reflective Description of the Reflective Tower**, Meta-Level Architectures and Reflection, P. MAes and D. Nardi (eds.), Elsevier Publishers B.V. (North Holland), 1988.

[Wegner&Zdonik88] P. Wegner, S. B. Zdonik: **Inheritance as an Incremental Modification Mechanism, or What Like is and Isn't Like**, In Proc. of ECOOP'88 European Conference on Object-Oriented Programming, pp.55-77, Springer-Verlag 1988.

[Wegner87] P. Wegner: **Dimensions of Object-Based Language Design**, In Proceedings of the OOPSLA'87 Conferenece, Sigplan Notices, pp.168-181, ACM Press, 1987.

[Wegner90] P. Wegner: **Concepts and Paradigms of Object-Oriented Programming**, OOPS Messenger Vol 1 Nr 1, ACM Press, August, 1990.

[Weinand&al88] A. Weinand, E. Gamma, and R. Marty: **ET++: an object-oriented application framework in C++**, In Proceedings of OOPSLA'88 conf., pp. 46-57, November 1988, printed as SIGPLAN Notices, 23(11).

[Wirfs-Brock90] Allen Wirfs-Brock: **Panel: Designing Reusable Designs: Experiences Designing Object-Oriented Frameworks**, Sigplan Notices Special Issue OOPSLA-ECOOP'90 Addendum to the Proceedings (Jerry L. Archibald and K.C. Burgess Yakemovic eds.), pp.19-24, 1990.

Index

A
absorbed 15, 24
abstract acquaintance 72
abstract class 65, 71, 72
abstract class attributes 70, 72, 162
abstract grammars 81
abstract methods 71, 162
abstract object representation 147
abstract parent 53
abstract representation of objects 142
abstract representations 27
abstract syntax trees 81
acquaintances 42
ADT encapsulation 48
aggregate expressions 92, 143
Agora 59, 132
anticipated delegation 57
applicability of mixins 122, 123
autistic object 42

B
base-level interface 14
behaviourally compatible inheritance 67
bounded polymorphism 67

C
class-based encapsulation 39, 42
class-based inheritance 57, 104
class-based languages 37
classifiers 163
client objects 85, 94
clone maps 154
clones 136
cloning 55
cloning methods 158
common ancestor duplication problem 107, 124
composite expressions 81
compositionality 29, 84, 182
compound expressions 92
compound objects 78
computational system 12
concrete subclasses 65
concretisation of an abstract class 66, 73
context objects 94

D
deification 24
delegation 37, 55, 59
delegation-based languages 55
design reuse 71
diamond problem 107
duplicate parent operation invocation 108
dynamic reclassification 50, 65
dynamic reflection 23, 180

E
encapsulated attributes 42, 61
encapsulated generator functions 58
encapsulated self 58
encapsulation 61
encapsulation of inheritance 70
encapsulation operators 61
ensemble 75
evaluation categories 97
explicit delegation 57
explicit interfaces 40, 41
explicit messages 167

explicitly encoded meta-objects 178
explicitly encoded object 167
explicitly linearised inheritance 124
explicitly referable object 167
expression kinds 97
expression objects 84
extensible syntax 92
extension of an abstract class 75

F

framework's external interface 66
framework’s internal interface 66
full abstraction 182
fully abstract 86
fully abstract semantics 30

G

generator function 54
generic class 68
generic expression 93
generic expressions 142
generic syntax 92
genericity inhibition problem 114
genericity mechanisms 67
graph multiple inheritance 107

I

implicit anticipated delegation 57
implicit delegation 57
implicit messages 167
implicitly encoded object 167
implicitly referable object 167
incremental changes 105
incremental definition 37
incremental modification 51, 57, 67, 118
incremental modifications of templates 49
infinite meta-regress 181
inheritance 37
inheritance dimensions 163
inheritance graphs 106
inheritance structure of objects 147
inheritance trees 106
inherited attributes 51
inheriting clients 43, 104
instance objects 60
instantiating clients 42
instantiation messages 60
internal objects 147
interoperability constraint for extension of abstract classes 75

L

language processor 13
late binding of self 53
late-binding polymorphism 38, 40, 44, 66
layered abstract classes 73
linear multiple inheritance 109
linearisation 109
linguistic symbiosis 24, 166

M

message expressions 133, 144
message passing 41
message qualification mechanism 159
meta-circular 13, 165, 180
meta-classes 51
meta-level architecture 13
meta-level interface 66
meta-object 86, 151, 168
meta-program 13, 174
meta-programming 174
meta-regression 23
meta-system 12
method associations 78
method dispatcher 45
method invocation 41
method lookup 41
method lookup semantics 53
minimal templates 61
mixin attributes 122
mixin method 136
mixin-attributes 59
mixin-based inheritance 59, 111, 112, 118
mixin-class 118
mixin-classes 111
mixin-method based inheritance 122
mixin-methods 59
mixins 111
module based encapsulation 128
module-based encapsulation 42, 50
monotonic reclassification 50, 61, 162
multi-methods 44
multiple inheritance 104, 105, 106
mutual exclusion constraint on subclasses 123

N

name collisions 106
name conflicts 108
nested classes 51, 129
nested mixins 119, 130, 151
nested objects 61
non-encapsulated representation of objects 85
nonencapsulated object 42

O

object cloning 154
object creation 55
object factories 70
object identity 37, 64

object representations 85, 147
object-based encapsulation 39, 40, 42
object-based inheritance 57, 58, 104
object-based languages 37
Object-based Programming calculUS 77
object-level interface 66
object-level program 174
object-level programs 13
object-oriented frameworks 53, 65
object-oriented languages 37
open designs 12, 27, 165
open implementation 12, 18
open implementations with reflective potential 22, 24
open implemented programming language 19, 21
OPUS 77
overriding method 52

P

point of view notion of multiple inheritance 114
private methods 61, 157
private attributes 42
program 12
proliferating subclass problem 105
proper attributes 51
prototype 57
prototype-based languages 55, 59
prototypes 136
public instance variables 42, 157

Q

qualified message passing 114

R

receiverless message expressions 146
receiverless messages 133
reclassification 163
referents 168
refinement 66
refinement constraint for abstract classes 74
refinement inhibition problem 114
refinement of an abstract class 74
reflect 23
reflection levels 179
reflection operators 23, 166
reflective architecture 22, 23
reflective facilities 22, 24
reflective overlap 23, 25, 179
reflective program 23
reflective programming language 23
reflective system 12, 22, 165
reification 15, 24
reifier class 94, 143
reifier expression 184
reifier message 94, 133, 143
reifier method 94, 186, 143
reifier patterns 143
reifiers 183
representation of objects 141
reusability 66

S

selectors 40
'self' pseudo-variable 52
signature compatible inheritance 67
Simple 77
single slot nested objects 160
slot-based language 43, 127
software reuse 37
static reflection 23
strict templates 49, 61
strictly encapsulated inheritance 104
strong identity 49, 64
stubs 121, 159
subobject 104
substitutability 67
substitutability rule 74, 75
substitutable objects 40

T

tangled inheritance hierarchies 105
template 49
template methods 71
template objects 60
tree multiple inheritance 112
type substitution 68

U

unanticipated delegation 57
unencapsulated self 58
unrestricted qualified message passing 114

V

virtual class attributes 69
virtual methods 71

W

weak identity 64
wrapper objects 151

Appendix A

Notes
In this appendix the most important abstract and concrete classes are listed of the framework that was described in the dissertation. First remark that the classes listed here can differ in minor details of their counterpart in the dissertation itself. The names of some classes have been changed to avoid name collisions. Remark also that this is by no means a complete listing of all the classes in the framework.

Furthermore some notes must be made about the descriptions of these classes. In principle the language used is self-explanatory. It uses a Smalltalk-like message passing, and class descriptions. However, two notations have been introduced that deal with overloading. The superscript "+" is used to indicate in a (abstract) class that an argument can be — apart from being overridden — overloaded in later subclasses. In fact it indicates that the method is overloaded on the indicated argument with all subclasses of the formal parameter. Also, a formal parameter that takes the form of a "name:class" couple indicates overloading on that argument. Finally note that in the class descriptions no instantiation methods are listed. We presume the existence of instantiation methods for all classes. Instantiation methods take the form of a keyword message composed of the instance variable names.

Kernel Abstract Classes of the Framework

Abstract Expressions with Overloading on the Context Argument

```
class AbstractExpression
  methods
    abstract eval:StandardContext+ result AbstractMetaObject
endclass
```

Abstract Meta-Object with Overloading on the Pattern Argument

```
class AbstractMetaObject
  methods
   abstract send:AbstractPattern+ client:StandardClient
      result AbstractMetaObject
endclass
```

Abstract Pattern Class

```
class AbstractPattern
  methods
   abstract = pattern result: Boolean
endclass
```

Standard Client Object

```
class StandardClient
endclass
```

Standard Context Object

```
class StandardContext
endclass
```

Kernel Classes of the Simple Framework

Message Passing with Evaluation and Pattern Categories

```
class UnaryMessageExpression extends AbstractExpression
  abstract class attributes
   EmptyClient
  instance variables
   receiver:AbstractExpression
   pattern:AbstractPattern
  methods
   concrete eval:(context:StandardContext+) result AbstractMetaObject
     ^(receiver eval:context)
        send:(pattern asCategory:context)
        client:EmptyClient
endclass
```

Receiverless Message Passing with Evaluation and Pattern Categories

```
class PatternExpression extends AbstractExpression
  abstract class attributes
   EmptyClient
  instance variables
   pattern:AbstractPattern
  methods
   concrete eval:(context:StandardContext+) result AbstractMetaObject
     ^context send:(pattern asCategory:context) client:EmptyClient
endclass
```

Standard Client Object for Simple, Grouping all Send Arguments

```
class SimpleStandardClient extends StandardClient
 public instance variables
  private:AbstractMetaObject
endclass
```

Standard Context Object for Simple, Grouping All Evaluation Arguments

```
class SimpleStandardContext extends StandardContext
 public instance variables
  private:AbstractMetaObject
endclass
```

Adapted Abstract Pattern Class for Simple

```
class SimpleAbstractPattern extends AbstractPattern
  methods
   abstract asCategory:StandardContext result: AbstractPattern
endclass
```

Abstract Class for Slots

```
class Slot
  instance variables
   key:AbstractPattern
  methods
   abstract valueIn:context result AbstractMetaObject
   concrete key result AbstractPattern
     ^key
endclass
```

Other Classes of the Simple Framework

Simple Expression Class Hierarchy

```
AbstractExpression
  AbstractionExpression
   AbstractCompoundObjectExpression (ACompoundObject)
     CompoundObjectExpression (publicPart:AbstractExpression,
                               privatePart:AbstractExpression)
   AbstractBaseObjectExpression (ABaseObject)
     BaseObjectExpression (associations:Sequence(AssociationExpression))
  UnaryMessageExpression (receiver:AbstractExpression,
                          pattern:AbstractPattern)
  PatternExpression (pattern:AbstractPattern)
AbstractAssociationExpression (AAssociation)
  AssociationExpression (pattern:AbstractPattern, value:AbstractExpression)
   VariableAssociationExpression
   MethodAssociationExpression
AbstractPattern
  Pattern(name:String)
```

Simple Meta-Object Class Hierarchy

```
AbstractMetaObject
     CompoundObject (publicPart:AbstractMetaObject
                     privatePart:AbstractMetaObject)
     BaseObject (slots:slots:Sequence(Slot))
```

Simple Slot Class Hierarchy

```
Slot
  VariableSlot (value:AbstractMetaObject)
  MethodSlot (value:AbstractExpression)
```

Agora Message Passing

```
class MessageExpression extends AbstractExpression
  abstract class attributes
   StandardClient
  instance variables
   receiver:AbstractExpression,
   pattern:AbstractPattern
  methods
   concrete eval:(context:StandarContext+)
       result AbstractMetaObject
     local variables arguments:ArgumentList

     for each argument in pattern do
        arguments add:(argument eval:(context asFunctionalContext))

     ^(receiver eval:(context asFunctionalContext))
         send:(pattern asCategory:context)
         client:(StandardClient arguments:arguments)
endclass
```

Evaluation of Receiverless Messages

```
class PatternExpression extends AbstractExpression
  abstract class attributes
   StandardClient
  instance variables
   pattern:AbstractPattern
  methods
   concrete eval:(context:StandaredContext+)
       result AbstractMetaObject
     local variables arguments:ArgumentList

     for each argument in pattern do
        arguments add:(argument eval:(context asFunctionalContext))

     ^(context privatePart)
         send:(pattern asCategory:context)
         client:(StandardClient arguments:arguments)
endclass
```

Pattern Classes Used in Expressions Versus Pattern Classes Used in Messages

```
class AbstractPattern
  methods
   abstract asCategory:StandardContext+ result AbstractCategoryPattern
endclass
```

```
class AbstractCategoryPattern
  methods
   abstract = AbstractCategoryPattern+ result Boolean
endclass
```

Agora Standard Client

```
class AgoraStandardClient extends StandardClient
  public instance variables
   arguments:ArgumentList
endclass
```

Agora Standard Context

```
class AgoraStandardContext extends StandardContext
  methods
   abstract asFunctionalContext result StandardContext
     … return an instance of functional context with the same
     … content
endclass
```

Agora Slot Class Hierarchy

```
Slot
  ReadVariableSlot (value:AbstractMetaObject)
  WriteVariableSlot (value:AbstractMetaObject)
  MethodSlot (value:AbstractExpression)
  MixinSlot (value:AbstractExpression)
  RiefierSlot (value:AbstractExpression)
```

Agora Expression Class Hierarchy

```
AbstractExpression
  MessageExpression (receiver:AbstractExpression, pattern:AbstractPattern)
  ReifierMessageExpression (receiver:AbstractExpression,
                                      pattern:AbstractReifierPattern)
  AggregateExpression (expressions:Array(AbstractExpression),
                              delimiter:Delimiter)
  LiteralExpression (value:LiteralValue)
  PatternExpression (pattern:AbstractPattern)
  ReifierPatternExpression (pattern:AbstractReifierPattern)

AbstractPattern
  UnaryPattern (id:Identifier)
  OperatorPattern (op:Operator, argument:AbstractExpression)
  KeywordPattern (keys:Array(Keyword), arguments:Array(AbstractExpression))
AbstractReifierPattern
  UnaryReifierPattern (id:Identifier)
  OperatorReifierPattern (op:Operator, argument:AbstractExpression)
  KeywordReifierPattern (keys:Array(Keyword),
                              arguments:Array(AbstractExpression))

Identifier (name:String)
Operator (name:String)
Keyword (name:String)
Delimiter (left:String, right:String)
```

Kernel Classes for Handling Inheritance in the Agora Framework

Wrapper Objects

```
class WrapperObject extends AbstractMetaObject
  abstract class attributes
   DelegationContext
  instance variables
   delegate: AbstractInternalObject
  methods
   concrete send:pattern client:client result AbstractMetaObject
     ^delegate delegate:pattern
              client:client
              context:(DelegationContext receiver:self
                                        public:delegate)
endclass
```

Root of the Abstract Internal Object Classes

```
class AbstractInternalObject
  methods
    abstract  delegate:CategoryPattern
              client:StandardClient
              context:DelegationContext+
        result AbstractMetaObject
endclass
```

Delegation Contexts Used in the Realisation of Mixin methods

```
class DelegationContext extends StandardContext
  methods
    concrete parent:AbstractInternalObject result DelegationContext
      --- returns a copy of the context with a new parent field
    concrete private:AbstractInternalObject result DelegationContext
      --- returns a copy of the context with a new private field
    concrete noParent result DelegationContext
      --- returns a copy of the context with a empty parent field
    concrete noPrivate result DelegationContext
      --- returns a copy of the context with a empty private field
    concrete receiver:AbstractMetaObject result DelegationContext
      --- returns a copy of the context with a new receiver field
    concrete public:AbstractInternalObject result DelegationContext
      --- returns a copy of the context with a new public field
endclass
```

Agora Delegation Contexts that Record the Public and Private Slots

```
class MixinContext extends DelegationContext
  methods
    concrete publicSlots:ExtensibleSimpleObjectresult DelegationContext
      --- returns a copy of the context with a new publicSlots field
    concrete privateSlots:ExtensibleSimpleObject result DelegationContext
      --- returns a copy of the context with a new privateSlots field
endclass
```

Concrete Internal Object Classes

```
DelegationContext
  ObjectWithParent (thisPart: AbstractInternalObject
                             parentPart: AbstractInternalObject)
  EncapsulatedObject (publicPart: AbstractInternalObject
                             privatePart: AbstractInternalObject)
  SimpleObject (thisPart:Sequence(Slot))
     ExtensibleSimpleObject
```

Appendix B

Construction of the Reflective Tower Based on Open Implementations

Kris De Volder[1] , Patrick Steyaert[2]

--- DRAFT (1.1) ---

Abstract

It is our opinion that the traditional view on reflection, the notion of towers of interpreters interpreting each other, is not sufficiently detailed to give a thorough understanding of reflection. Expressions such as "somehow the levels must be connected" and "adding lines to the interpreter above" are typical when talking about towers of meta-circular interpreters. This alone gives an indication that the model is not detailed enough, lacking a way to formalise the relation between levels of the tower. The connection remains a magical ingredient in the recipe to cook up a reflective system. This is the main reason why reflection has hitherto remained covered in a mystical veil.

We present an alternative view on reflection. Rather than being based on meta-circular interpreters, this model is based on open implementations. An open implementation hides the implementation details of the interpreter, but shows how the interpreter can be extended/adapted. In this approach reflection is obtained by explicitly generating the limit of an infinitely ascending chain of open implemented interpreters through a fix-point operation.

We argue that the connection between interpreters in the traditional view is ad-hoc and counter-intuitive. The open implementation point of view yields a notion of reflection which is highly similar to the traditional view, but improves upon the ad-hoc way of relating interpreters at different levels. In a tower of open implementations the connection between levels is established in a natural way through the parameters for the open-implementation which are provided by the implementing level above.

1 Motivation

Every reflective system needs an accessible, causally connected self-representation. As every representation defines a certain terminology to talk about the entities it represents, so does this self-representation define a terminology for the system to talk about itself. The self-representation determines the system’s aspects that can be reasoned about and modified by the system. As is true for any representation the self-representation can not be “complete”, i.e. any representation will always ignore certain aspects of the system it represent. For reflective systems this is known as the “theory relativity” of reflective systems [Maes87].

For procedurally reflective languages it is said that the procedural code in the meta-circular processor serves as the “theory” or causally connected self-representation [Smith84]. It is our opinion that this is a misleading, or even wrong, statement. And that exactly this statement hampers our true understanding of reflective systems. In this paper we will discard with meta-circular processors as self-representations. Moreover, since the meta-circular processors are used in the tower model, we will also discard the notion of towers of meta-circular processors. We will not discard with the notion of tower-architectures! Only with towers of meta-circular processors.

One of the motivations for our work is the demystification of the magical "link between levels" ingredient. As stated before, we believe that the traditional model is not sufficiently detailed in this respect.

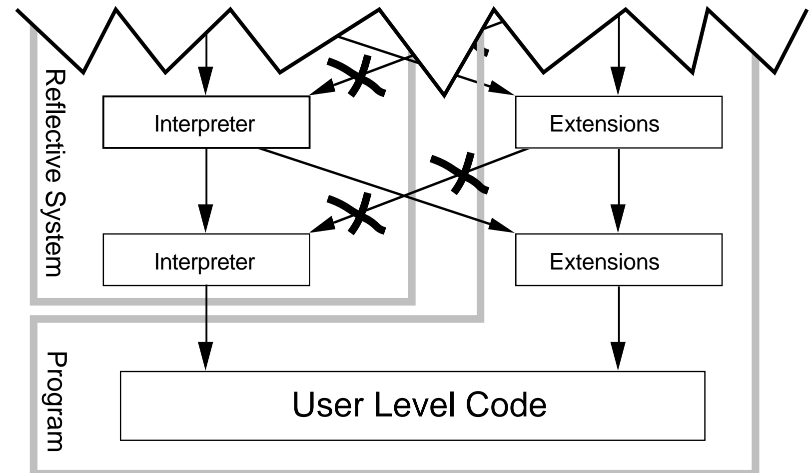
Another and perhaps more important consideration is the fact that the traditional view defies the notion of theory

1 Programming Technology Lab; Computer Science Department; Vrije Universiteit Brussel; Pleinlaan 2, B-1050 Brussel, Belgium; email: kdvolder@vnet3.vub.ac.be

2 Programming Technology Lab; Computer Science Department; Vrije Universiteit Brussel, Pleinlaan 2, B-1050 Brussel, Belgium; email: prsteyae@vnet3.vub.ac.be

relativity. When using the procedural code in the meta-circular processor for a self representation it is tempting to think that if one can change the code of the meta-circular interpreter in any way one likes, one must be able to do just about anything. This of course is not true because a meta-circular evaluator alone does not define a programming language. An external processor is needed to process the meta-circular interpreter.

If the procedural code of the meta-circular processor is to serve as a causally connected self-representation then any modification to this code must not only affect the interpretation of user programs but also the interpretation of the meta-circular processor's code itself. In practice—in any existing implementation—this is not the case. Thus the meta-circular code does not truly serve as a self representation. The following figure tries to illustrate this.



This picture represents a traditional infinite tower of meta-circular interpreters. The arrows indicate the "… plays a part in the interpretation of …" relationship. The reflective system is a tower of meta-circular interpreters. This tower is used to interpret a program. The program usually contains some normal non reflective "User Level" code but also some reflective code that will be installed as part of the interpreter (shown in the drawing as "Extensions"). The traditional model is misleading because it gives the impression that the extended interpreter is used to interpret itself meta-circularly. This is not true, the extended interpreter is only used to interpret the extensions, while the "core" of the interpreter is not affected by the extensions. That is why we have crossed out the arrows leading from "extensions" to "interpreter".

In any existing implementation, extensions to the language do not affect the interpretation of the "core interpreter". The extensions only affect evaluation of a) “User Level Code” and b) the code implementing the extensions themselves. They *do not* affect the core-interpreter. In many systems this is so because the "core interpreter" is explicitly written in a subset of the language that can not be altered by reflective programming (e.g. a variant of Scheme with reifier-procedures, whereby the implementation does not make use of reifiers, nor is it possible for reifiers to override the pre-defined special forms). In other implementations the core interpreter is written in a part of the language that can be changed by reflective programming (e.g. a variant of Scheme whereby the pre-defined special forms can be redefined), but even then the scope of the changes will only include the extension's implementation and the user level code, but never the actual interpreter itself.

The traditional model is deceiving because it does not distinguish the extensions (added by reflective programming) and the core of the interpreter from each other. Our approach attempts to remedy this by dividing the interpreter into a fixed and a parameterised part.

In this paper we will start with building an open implemented interpreter. This open implementation will be

written meta-circularly, meaning that it can be evaluated with some “basic” evaluator obtained from that open implementation itself. Then we will experiment a little with finite[3] literal towers of open implementations. The finite tower experiment serves as a stepping stone towards reflection, providing an easy way to experiment with towers of open implementations. After playing around with finite towers for a while we will introduce “real” reflection characterised by a fix-point equation and show the relation between this equation and infinite towers.

2 The Open Implemented interpreter

Our approach is an attempt to refine the traditional model. We will represent a level of the tower by an open-implemented interpreter, explicitly representing the "fixed-core" of the system as a separate entity.

We will use the following simple example language (ASEL), which is a subset of Scheme.

```
<exp>          =  <var> | <constant> | <lambda> | <if> | <definition> |
                  <assignment> | <application>
<var>          =  <scheme-symbol>
<constant>     =  <scheme-literal> | <quoted>
<quoted>       =  '<scheme-value>
<lambda>       =  (lambda <formals> <sequence>)
<sequence>     =  <exp>+
<formals>      =  () | <var> | (<var>+ [. <formals>])
<if>           =  (if <exp> <exp> <exp>)
<definition>   =  (define <var> <exp>)
<assignment>   =  (set! <var> exp>)
<application>  =  (<exp> <exp>*)
```

An open implementation is in essence nothing more than a parameterised interpreter. The parameterised interpreter will take the form of a function we will name *meta*. Applying *meta* to a parameter will return an evaluator based on that parameter. Thus we can obtain a range of evaluators, by applying *meta* to a variety of parameters. The fixed core is explicitly represented by *meta*.

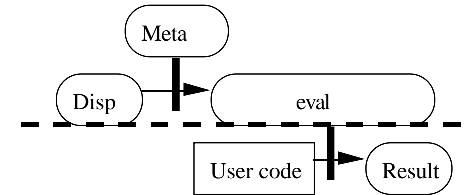
The way we write *meta*, the choice of parameterisation, establishes beforehand in what way we can adapt the evaluator. In the system we will implement here as an example, the goal is to be able to extend the evaluator so that it can handle new kinds of syntax structured as follows:

```
(<syntactic-keyword> <arg>*)
```

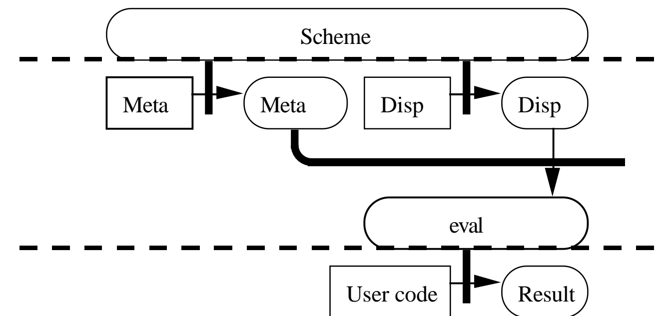
For example we could extend the evaluator with *cond*, *let*, *let**, … expressions. The parameter that is passed to *meta* will take the form of an assoc-list which associates an appropriate evaluation procedure with a *<syntactic-keyword>*. We will also provide a standard parameter, yielding the basic evaluator which handles the vanilla version of ASEL. The definition of *meta* will be written meta-circularly, which in our case means that it is implemented in vanilla ASEL and thus can be evaluated with the basic-evaluator.

The following figure gives a schematic view of what our open implementation looks like.

[3] This approach is inspired by the work of Jefferson and Friedman [Jefferson&Friedman92]. They introduce reflection through explicitly constructing finite towers meta-circular interpreters that are interpreting each other literally. This gives poor performance, but it does give a very clear, simple and understandable account of procedural reflection. One can observe the behaviour of a finite tower and “extrapolate” upon this to understand how an infinite (procedurally) reflective tower behaves.



The convention is that rectangular objects represent syntactic entities, i.e. pieces of source code. Round objects represent functions. The application of a function is represented by a thick black line, an arrow points through the black line, pointing from the argument to the result of the application. This diagram shows the open implementation as a function called *meta*, that is applied to a dispatcher. The result of this application is a function called *eval*, this is the evaluator. The evaluator is then used to evaluate user code. One last remark: the parameter to our meta function is called *disp*, in our system this is an association list, which associates a syntactic keyword with an evaluation function. Thus, strictly spoken, *disp* is not a function but to keep the figures simpler we will treat it as such. Regarding *disp* as a function does not change anything in an essential way. Similarly *result* is also drawn as a function although it usually is something else like a number, a list, ….



The previous figure gives a more detailed view of the open implementation, this time also showing the open implementation's source code, and the dispatcher's source code. Both are evaluated by the underlying scheme's evaluation function, yielding the *meta* and *disp* functional objects.

We can view *meta* as establishing the *meta-theory*. It determines what aspects of the interpreter we can talk about, and how we must do this (i.e. what parameters must be supplied to *meta*). In this respect we can consider the argument(s) to *meta* as the *representation* for some evaluator. The meta-theory relates the representation to the evaluator it represents. Without a proper meta-theory, the representation is meaningless. This model explicitly exhibits *theory relativity*. The representation is not complete it only determines an interpreter in the context of some meta-theory: some of the interpreters aspects are contingent to this meta-theory.

The *representation* (the round Disp in the drawing) is a semantical object. We must denote this semantical object by some syntactical structure (the rectangular Disp in the drawing). An interpretation function will be needed to relate the syntactical *description* with the corresponding semantical *representation*. In this drawing that interpretation function is the Scheme evaluator. From here on we will use "*representation of E*" for referring to the semantical object that represents an evaluator *E* in the sense described above. Respectively we will use "*description of E*" for referring to the syntactical structure denoting the representation of *E*.

The overall structure of *meta* is displayed below. It takes a dispatch-table assoc-list as argument and returns an evaluator. The implementation of the evaluator, which is hidden inside the body of *meta*, is written in continuation passing style. All evaluation procedures like evaluate, basic-evaluate, evaluate-constant, … take 3 arguments *e*, *r* and *k*. These are respectively the expression to be evaluated, the current environment (= set of bindings of variables to values) and the current continuation. The evaluation procedures in the dispatcher take another extra argument: *evaluate*. This is the evaluator itself, passed as an argument to dispatcher procedures so

that they may use it to evaluate sub-expressions.

```
(define meta
  (lambda (dispatch-table)

    (define evaluate
      (lambda (e r k)
        (if (pair? e)
           (find-pair (car e) dispatch-table
              (lambda (success-pair)
                ((cdr success-pair) evaluate e r k))
              (lambda ()
                (basic-evaluate e r k)))
           (basic-evaluate e r k))))

    (define basic-evaluate
      (lambda (e r k)
        ((if (constant? e)
           evaluate-constant
           (if (variable? e)
             evaluate-variable
             (if (if? e)
               evaluate-if
               (if (assignment? e)
                 evaluate-assignment
                 (if (definition? e)
                   evaluate-definition
                   (if (abstraction? e)
                     evaluate-abstraction
                     evaluate-combination))))))
         e r k)))

    (define evaluate-constant …)
    (define evaluate-variable …)
    …
    (define …)

    evaluate))
```

The main procedure, *evaluate*, checks the dispatch table for an appropriate evaluation procedure to call. If one is found, then it will be used to evaluate the expression. If the dispatcher does not contain a procedure for this type of expression, then *basic-evaluate* gets called. *Basic-eval* handles all "vanilla" ASEL expressions, it distinguishes between different types of expressions and dispatches to an appropriate evaluation procedure for that particular expression type.

Most of the sub-task procedures that *basic-eval* dispatches to are rather straightforward, so we won't explain all of them here. The full source code can be found in appendix A. Since some of the rest of this paper will involve dealing with complications that arise when procedures are moving up and down in the tower of interpreters, and inter level (in)compatibility of procedures, we will now take a look at the code of the evaluator involving procedure creation (evaluation of lambda expressions) and procedure calls.

```
(define evaluate-abstraction
  (lambda (e r k)
    (k (make-compound (formals-part e) (body-part e) r))))
```

Evaluation of a lambda expression (abstraction) is very straightforward: create a representation for a procedure by calling *make-compound*, and pass the result to the continuation.

```
(define make-compound
  (lambda (formals body r)
    (lambda (k . args)
```



```
      (evaluate-sequence body (extend r formals args) k))))
```

As can be seen from the definition of *make-compound*, procedures are represented by procedures. The representation procedure has an extra first argument. This extra argument is a continuation that will receive the result of the procedure-call. The remaining arguments are the "real" arguments. Application of a procedure represented in this way is written as follows.

```
(define apply-procedure
      (lambda (proc args k)
        (if (procedure? proc)
          (apply proc (cons k args))
          (wrong "operator is not a procedure" proc))))
```

The *evaluate-combination* procedure is the one that is called for evaluating procedure calls. It first evaluates the operator part (the expressions that yields the procedure to be called). Then it evaluates the operands one by one by calling the procedure *evaluate-operands*. Finally the procedure (*proc* = result from evaluating the operator part) is applied to the list of evaluated arguments.

```
(define evaluate-combination
      (lambda (e r k)
        (evaluate (operator-part e) r
           (lambda (proc)
             (evaluate-operands (operands-part e) r
                (lambda (args)
                  (apply-procedure proc args k)))))))
```

We also provide a read-eval-print loop mechanism, so that we can type in expressions to be evaluated and see the result printed on the screen. A read-eval-print loop can be started by calling the function *openloop* passing the evaluator as an argument.

```
(define openloop
  (lambda (evaluate read-prompt write-prompt)
    (display read-prompt)
    (evaluate (read) global-env
      (lambda (v)
        (display write-prompt)
        (if (eq? v (void))
          "Nothing is displayed"
          (write v))
        (newline)
        (openloop evaluate read-prompt write-prompt)))))
```

Now we have everything we need to start a session, we can open a read-eval-print loop on a basic evaluator, or variation of the evaluator we create by applying meta to a parameter. The following is an example of how one might create an extended evaluator that understands a special *exit* expression. When an *exit* expression is evaluated, it causes the evaluator to terminate promptly, discarding all pending computations. The example shows the evaluation of some simple expressions, an invocation of the *exit* construct ends the session. Things typed in **bold** where typed in by the user. Things in normal font where responses or prompts printed by the read-eval-print loop or by the underlying scheme system.

```
scheme> (initialise-global-env)
scheme> (define exit-evaluator
            (meta (list (cons 'exit (lambda (evaluate e r k) e)))))
scheme:
scheme> (openloop exit-evaluator "exit> " "exit: ")
exit> (* 3 4)
exit: 12
exit> (define foo (lambda (x) (* x x)))
exit:
exit> foo
exit: #[procedure #x8B2D2]
```

```
exit> (foo 5)
exit: 25
exit> (exit)
scheme: (exit)
scheme> …
```

3 Finite Towers

3.1 Construction

The next step towards a reflective tower will be to use the code from (2) meta-circularly, building a finite tower of a fixed number of literal levels of open implementations. For this purpose we add a procedure *loadfile*, that enables us to read a file from disk and interpret its expressions one by one with an interpreter of our choice. *loadfile* is very similar to a read-eval-print loop, but also checks for *end-of-file* and reads expressions from a file instead of from the keyboard.

```
(define loadfile
  (lambda (evaluate file)
    ((lambda (port)
       ((lambda (loop)
          (set! loop
            (lambda (v)
              (if (eof-object? v)
                (close-input-port port)
                (evaluate v global-env
                   (lambda (ignore)
                     (loop (read port)))))))
          (loop (read port)))
        '*))
      (open-input-file file))))
```

Now it's easy to build a finite tower of open implementations. In the following example we will build a tower of 2 levels, with at every level of the tower an interpreter that is extended with a *climb* construct. The climb construct takes one argument. This argument must evaluate to a strictly positive integer value. Evaluation of climb will cause an exit from exactly the number of levels indicated by the argument. The climb construct is not a very useful thing, but we employ it because it is a simple example of a construct that needs an arbitrary number of reflection levels (depending on the argument).

First we load the file "open-simple.scm", which contains the definitions for *meta*, *loadfile*, *openloop*, … . Then we initialise the level 1 global environment for the first level and load "open-simple.scm" again, but this time into the level 1 global environment, using the level 1 basic evaluator.

```
scheme> (load "open-simple.scm")
scheme:
scheme> (initialise-global-env)
scheme:
scheme> (loadfile basic-eval "open-simple.scm")
scheme:
```

Next we define the dispatcher and evaluator for level 1. The evaluator for level 1 is an evaluator extended to handle *climb*.

```
scheme> (define climb-proc
          (lambda (evaluate e r k)
               (evaluate (car (cdr e)) r
                   (lambda (how-many)
                     (if (= 1 how-many)
                      how-many
                        'cannot-climb-further)))))
scheme:
```



```
scheme> (define climb-dispatcher (list (cons 'climb climb-proc)))
scheme:
scheme> (define climb-evaluator (meta climb-dispatcher))
scheme:
```

After doing all of the above, the system is ready to start the level 1 read-eval-print-loop with a call to *openloop*.

```
scheme> (openloop climb-evaluator "1> " "1: ")
1>
```

To add another level to the tower we simply go through the same steps again, defining the dispatcher and evaluator and starting a read-eval-print loop. This time we can skip loading "open-simple.scm" because we do not intend to add a third level below level 2, so we don't need to load another meta-circular open implementation.

```
1> (initialise-global-env)
1:
1> (define climb-proc
           (lambda (evaluate e r k)
                (evaluate (car (cdr e)) r
                    (lambda (how-many)
                      (if (= 1 how-many)
                       how-many
                         (climb (- how-many 1)))))))
1:
1> (define climb-dispatcher (list (cons 'climb climb-proc)))
1:
1> (define climb-evaluator (meta climb-dispatcher))
1:
1> (openloop climb-evaluator "2> " "2: ")
2>
```

Notice that the definitions of climb for level 1 and level 2 are not identical. Normally climb calls "itself" recursively when it needs to climb more than one level. Strictly spoken "itself" is not correct, since the *climb* that is called and the one that is being implemented are in different levels of the tower. The level 2 *climb* is taken care of by *climb-proc* at level 1, which relies on level 1's climb. Since scheme, which coincides with level 0 in our tower, does not understand *climb*, the level 0 *climb-proc* (implementing level 1 climb), cannot rely on it. This is the reason why the level 0 *climb-proc* instead of calling *climb* "recursively" returns the symbol "cannot-climb-further". The result is that on level 2 we can climb at most 2 levels, and at level one we can climb only one level. Trying to climb more than this number of levels leaves us in Scheme, with the message "cannot-climb-further".

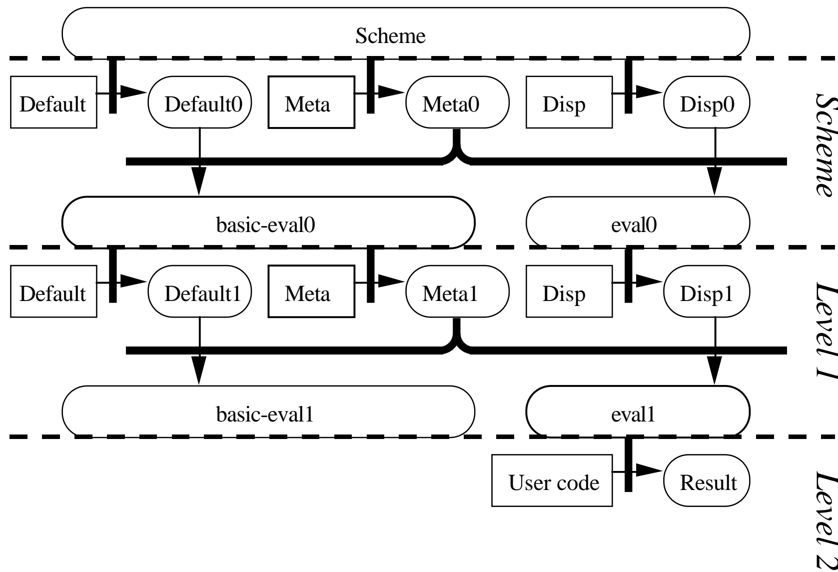
```
2> (* 3 4)
2: 12
2> (climb 1)
1: 1
1> (climb 2)
scheme: cannot-climb-further
scheme> …
```

3.2 Explanation

The following figure illustrates the configuration of the finite tower we just created. A little explanation is in order here. Dashed horizontal lines separate the different levels of the tower. Evaluation functions are drawn at the boundary, just above the dashed lines. Their application protrusions point through the dashed line into the level below, because they "reason about" objects from the level below, the level they are implementing. Notice that we actually have 2 towers here, standing right next to each other. On the left is a tower of basic evaluators, and on the right a tower of customised evaluators. The end product, the evaluator that is used to evaluate user code is the bottom of the tower of customised evaluators.

Although this is only a finite tower, and not a real reflective system, it already illustrates some interesting

things. By representing a level of the tower as an open implemented interpreter, we have exactly the right amount of detail to be able to represent the independence of the fixed core of the system from the extensions. *Meta* and *default* are evaluated explicitly with *basic-eval*. This ensures that changes to the language introduced by the reflective parts of the user program (*disp* in the drawing) will not affect the interpretation of the fixed core.



In this implementation, the dashed line boundaries are very strict, function representations at different levels are incompatible, a function of one level cannot be used at another level. To see that this is true let's examine the representation of procedures at different levels, for an example let's consider the representation of the + procedures. At scheme's level this is simply represented by the native primitive addition procedure. On every level procedures are represented as procedures of the implementing level with one extra continuation argument in front of the real argument list (remember the definition of *make-compound* discussed in section 3). So at level 1 the representation for the primitive addition procedure will roughly correspond to the result of evaluating the following expression in native scheme:

```
(lambda (k1 . args) (k (apply + args)))
```

Similarly, the addition procedure at level 2 will correspond to the evaluation of the following at level 1:

```
(lambda (k2 . args) (k2 (apply + args)))
```

Which in turn corresponds to the following evaluated in scheme:

```
(lambda (k1 k2 . args) (k2 k1 (apply + args)))
```

Every level of the tower introduces an extra continuation, thus a procedure representation at level 1 takes one extra continuation argument, and a procedure at level 2 takes 2 extra continuation arguments. This obviously shows that procedures at different levels differ in the number of hidden continuation arguments they expect and are therefor not interchangeable.

4 Reflection

4.1 Why the Finite Tower is not Reflective

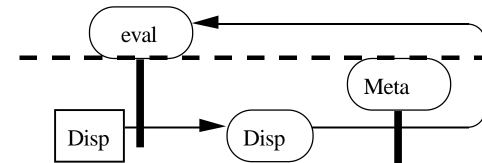
There are strong arguments to say that it would be a mistake to call the finite tower architecture just presented a reflective system. It isn't much more than an open implemented interpreter, it is just a number of open implemented interpreters executing each other. An open implemented interpreter in itself is not reflection, it is merely an interpreter that can be varied upon by supplying different parameters. The most important aspect of reflection, the ability of a program/interpreter to reason about itself is completely absent, an interpreter cannot reason about itself, it can only reason about the interpreter below. The interpreter below is a similar, but nevertheless a different interpreter, it need not even be extended in the same way.

4.2 The Fix-point Equation of Reflection

Merely an open implementation by itself is not reflection. What do we expect from a "reflective language/interpreter"? First of all, we need some meta-theory to talk about the interpreter. Second a *representation* of the evaluator is needed under this meta-theory. Obviously these 2 things are not sufficient to get a reflective system because both of these are present in open implementations: there is a meta-theory established by *meta* and the parameters to *meta* serve as representation for the evaluator. One essential thing is missing however. In a reflective system we want to be able to express the description for the interpreter in the language implemented by that interpreter itself! That is the essence of reflection! We express this in the following equation:

$$E = (M\ (E\ d))$$

Our convention is to write functional objects (round objects in the drawings) with capitals, and "source code" objects (rectangular in the drawings) with small letters. In this equation the evaluator, E, that is being created by applying the meta function, M, to a dispatcher function (representation of the evaluator) is the same as the evaluator that is being used to create the dispatcher by evaluating its source code (description) d. This fix-point equation is the key to reflection in a system of open implemented interpreters. Schematically we can draw this fix-point equation as follows:



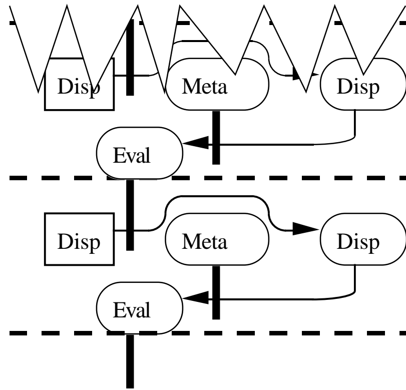
The "recursiveness" of the fix-point equation shows up in this diagram under the form of a circularity in the drawing. Notice that the arrow that leaves from the dispatcher goes through the level boundary. Remember that we pointed out before there cannot be arrows crossing level boundaries, the boundaries are strict. This is a complication we will deal with in the following 2 sections.

4.3 Reflection and Infinite Towers

The equation $E = (M\ (E\ d))$ actually implies an infinite tower of open implementations. This can be seen when rewriting the equation into longer and longer equivalent equations as follows:

$$E = (M\ (E\ d))$$
$$E = (M\ ((M\ (E\ d))\ d))$$
$$E = (M\ ((M\ ((M\ (E\ d))\ d))\ d))$$
$$\ldots$$
$$E = (M\ ((M\ ((M\ ((M\ ((M\ (\ldots\ d))\ d))\ d))\ d))\ d))$$

The last equation, the result of substituting $(M\ (E\ d))$ an infinite number of times into $E = (M\ (E\ d))$ can be regarded as an infinite tower, corresponding to the next figure.



In this figure there is an infinite number of levels (imagine that the level is repeated ad infinitum). Every level contains an evaluator, a meta and a dispatcher. You should consider all rounded rectangles containing the word "meta" to stand for one and the same meta-function, simply drawn multiple times. Similarly consider the rectangles and rounded rectangles representing dispatcher, dispatcher source code and evaluator as multiple drawings of the same objects. It is not difficult to see that actually the 2 drawings, the infinite tower and the one with the circularity, are "isomorphic", that is, if we simply look at the way the arrows go, and do not distinguish between the multiple copies at different levels, but regard them as identical, the two drawings are the same. In both drawings there are 2 arrows. One that starts from dispatcher source code goes through the evaluator and points to the dispatcher function, representing the application $(E\,D)$. The other arrow starts from the dispatcher function goes through the meta-function and points to the evaluator, representing the application $(M(E\,D))$.

Notice that in the infinite tower figure there are no arrows crossing level boundaries. This shows that we can think of the fix-point equation as an infinite tower of open implementations, without the discrepancy of level crossing procedures.

We consider both drawings to represent equivalent views of the fix-point equation. The circular one directly represents the recursion by a circularity in the drawing, thus looking at the equation from a rather direct, "implementational" viewpoint. While the infinite one looks at it from a conceptual, behavioural angle, viewing it as the representation for an infinite tower of identical open implementations. The first drawing we conceive as being of an implementational nature, because we are going to implement reflection directly by supplying an suitable *M*, that can be used to define an evaluator by directly expressing the fix-point equation, using recursion in ASEL to define *E* in terms of itself. The infinite tower drawing gives the more conceptual view, expressing that the result will be something that behaves like an infinite tower of identical open implementations, with identical extensions at every level.

5 Implementing the Infinite Tower

This section gives a brief description of how to implement an infinite tower of open implemented interpreters. This can be done in 2 stages.

5.1 Stage 1: Implementing *M*

The level crossing arrow in the figure from section (4.2) indicates that there is something strange about the *M* in the fix-point equation. Normally level crossing arrows are not possible.

Our implementation of *M* is inspired upon the left side of the finite tower from section 4, extended into infinity. An important aspect of the infinite tower is that procedures at different levels (near the bottom of the tower) are interchangeable. This is different from the situation in finite towers where procedures at different levels differ in the number of continuation arguments they receive as hidden arguments. At the bottom of an infinite tower, a procedure expects an infinite number of continuation arguments. Informally we could argue that adding one more continuation to an already infinite number doesn't make much of a difference, there still are an infinite number of them.

We have built a "level shifting" implementation, based on the idea of a *meta* at the bottom of an infinite tower. For technical reasons it was not possible to mimic the behaviour of an infinite tower exactly, but what matters is that we were able to ensure interlevel compatibility of procedures. The whole thing boils down to simulating an infinite tower by maintaining a stack of meta-continuations that is virtually infinite, but from which only a finite number of the topmost levels will actually be used. We will not go into detail because this is very similar[4] to the traditional implementation of a level shifting interpreter as previously described in [desRivières&Smith84] and [Smith84]. The result is an implementation of *meta* (from here on called $meta_\infty$) as an "infinite level procedure". It can be applied on dispatchers that contain "infinite level" procedures and returns an evaluator that also is an "infinite level" procedure.

5.2 Stage 2: Fix-point Equation

Given $meta_\infty$ we can use the fix-point equation given under (4.2) to spawn an infinite tower of customised interpreters. The system we have implemented provides a read-eval-print loop, in which the user can type in expressions. The evaluator used to evaluate the expressions is an instance of *basic-eval*, created by applying $meta_\infty$ to the default-dispatcher. The global environment contains a reference to $meta_\infty$ in a variable called "meta*". Thus the user can create his own dispatcher and pass it to $meta_\infty$ to create an infinite tower of customised evaluators.

The following is an example showing the creation of an infinite tower of evaluators that understand the *climb* syntax. The code is a bit more complicated than expected but this is merely the result of some technical matters. A dispatcher is not really a function, but an assoc-list, containing functions, this makes things a bit more intricate. Another complication is that Scheme, and ASEL (a subset of scheme) do not have delayed evaluation so we have to throw in some extra η-redexes here and there to avoid infinite loops.

First the variable “climb-code” is bound to the source code (notice the quote) of the climb-dispatch procedure. Thus the contents of “climb-code” corresponds to *d* in the equation.

```
0> (define climb-code ;; d
      '(lambda (evaluate e r k)
          (evaluate (car (cdr e)) r
             (lambda (how-many)
                (if (= 1 how-many)
                   how-many
                     (climb (- how-many 1)))))))
0:
```

Next we construct a dispatcher assoc-list that contains the evaluation of the source code from “climb-code” and store that in a variable “climb-dispatcher”. The evaluator that should be used for evaluating the source code should be “eval-climb” the evaluator we are constructing. Since this evaluator will be declared later and is still unavailable, we must delay the evaluation with an extra η-redex.

```
0> (define climb-dispatcher ;; (E d)
      (list (cons 'climb
                  (lambda (eval e r k) ;; extra η-redex
                     (eval-climb climb-code global-env
```

4 Actually our implementation looks simpler and more elegant than similar things written for the traditional model because the ad-hoc notion of things moving up and down the tower has disappeared. The link between levels are through the parameters for *M*. Source code can be found in the Appendix.

```
                    (lambda (ED)
                      (ED eval e r k)))))))
0:
```

Subsequently we obtain the evaluator by giving the dispatcher created above as an argument to “meta*”. Here we also need an extra η-redex, this time to avoid infinite looping.

```
0> (define eval-climb ;; E = (M (E D))
     (lambda (e r k)   ;; extra η-redex
       ((meta* climb-dispatcher) e r k)))
0:
```

Finally we can use the evaluator. For example we can start a read-eval-print loop and evaluate some expressions.

```
0> (openloop eval-climb "1> " "1: ")
0:
1> (climb 3)
-2: 0
-2> …
```

6 Why the Infinite Tower of Meta-Functions is Reflective

It can be argued, on the basis of the time of definition and installation of dispatcher functions, that the tower of meta-functions is still a weaker form of reflection than the more common reifier functions sort of reflection. We will argue that the difference is merely a matter of 1) a lower degree of reflective overlap (something one really tries to avoid) and 2) pragmatics and the choice of the particular open implementation.

Dispatcher functions are defined and installed in the evaluator prior to their usage in some user program. It might seem that there is an even stronger form of reflection whereby a user program can install dispatcher functions while it is running. In this set-up dispatcher functions are defined in the context of the user program. First and most obvious one can remark that this leads to reflective overlap regarding the environment. Dispatcher functions are evaluated in an (implicit) environment that will later be given as explicit argument to them.

Furthermore it can be argued that the fact whether extensions to the interpreter are made prior to rather than during the execution of a program, depends largely on the architecture of the open implementation. We claim that in the system given here, this is mostly the result of the direct mapping of the theoretical "meta-functional" view of reflection onto an implementation which retains the functional nature of parameterisation. Most (if not all) present day implementations of reflective system involve some kind of side effect in the installation of parameters into an evaluator (with procedural reflection for example this is the side effect that installs a reifier procedure into the current environment). This naturally yields a more dynamic behaviour and gives a more direct impression that the system/program reasons about itself or about its evaluator.

To illustrate our point, it is possible to imagine a practical implementation that has the capability to destructively alter the dispatcher after the instantiation of an evaluator. This yields something that is highly similar to the reifier approach, be it with one big difference, it is impossible to pass on reifiers as arguments. This in itself is not a problem, some people even claim that the ability to pass on reifiers is a flaw in the procedural reflection approach [Bawden88].

7 Comparing Static Reflection, Dynamic Reflection and Finite Towers

The above defined *meta** need not be used to spawn infinite reflective towers. It need not be used to express programs that can climb arbitrarily high in the tower. Consider the following example. Here again a climb syntax is defined, but in contrast with the above climb syntax it can only be used to climb a number of levels in the tower which is statically defined. The evaluators being defined are not the result of some fix-point equation.

```
0> (define climb-proc-0
     (lambda (evaluate e r k)
       (evaluate (car (cdr e)) r
```



```
         (lambda (how-many)
           (if (= 1 how-many)
             how-many
               'cannot-climb-further)))))
0:
0> (define climb-dispatcher-0 (list (cons 'climb climb-proc-0)))
0:
0> (define climb-evaluator-0 (meta* climb-dispatcher-0))
0:
0> (openloop climb-evaluator-0 "1> " "1: ")
1> (define climb-proc-1
     (lambda (evaluate e r k)
       (evaluate (car (cdr e)) r
         (lambda (how-many)
           (if (= 1 how-many)
             how-many
               (climb (- how-many 1)))))))
1:
1> (define climb-dispatcher-1 (list (cons 'climb climb-proc-1)))
1:
1> (openloop climb-evaluator-1 "2> " "2: ")
2> (climb 1)
1: 1
1> (climb 2)
0: cannot-climb-further
0>
```

Although the above sequence of climb-evaluators looks very similar to the climb-evaluators defined with the finite tower of section 3, they have noticeably different properties. The following equations show the construction of both. Again capitals denote functions and small letters denote pieces of source code. *CD, D, BE*, *M* and *CE* stand for *climb-dispatcher*, *default-dispatcher*, *basic evaluator*, *meta* and *climb-evaluator* respectively. The indexing convention is that we put an index to denote the level a function belongs to if this level is finite. For example BE_1 is an instance of the basic evaluator that is a level 1 procedure (taking one continuation argument). If the level is infinite the index is omitted. Thus *BE* denotes an instance of basic-eval that is an "infinite level" procedure. Indexes to source code don't signify a certain level because pieces of source code are never bound to a certain level and can be freely interchanged without compatibility problems. Thus cd_0, … , cd_n merely denote *n* different pieces of source code.

$$CE = (M\ ((\ldots((M\ ((M\ (BE\ cd_0))\ cd_1))\ \ldots))\ cd_n)) \qquad \text{with meta*}$$

$$CE_{n+1} = (M_{n+1}\ ((\ldots((M_2\ ((M_1\ (BE_0\ cd_0))\ cd_1))\ \ldots))\ cd_n))$$
$$BE_0 = Scheme$$
$$M_i = (BE_{i-1}\ m) \qquad \text{finite tower}$$
$$BE_i = (M_i\ D_i)$$
$$D_i = (BE_{i-1}\ d)$$

Most noticeable is a difference in performance. In the second case there is very large interpretation overhead. There are *n* levels of interpreters literally interpreting each other, where *n* is the statically predetermined maximum number of levels one can *climb* in the tower. The first is much more efficient because the interpretation overhead will only occur for interpretation of the specific code in the dispatcher implementing the *climb* construct.

The performance cost of extra numbers of evaluation levels is avoided due to the special properties of $meta_\infty$. The extra performance cost associated with finite towers is due to extra flexibility in the architecture. Whereas using $meta_\infty$ implies that the meta function is fixed, the meta-function of finite towers need not all be evaluated with the same evaluator. This brings us back to the issues raised in the introduction. For a finite tower it *is* possible to influence the 'core' of the interpreter. For example one could define a sequence of evaluators:

$$CE_1 = ((BE_0\ m)\ (BE_0\ cd_0))$$
$$CE_2 = ((CE_1\ m)\ (CE_1\ cd_1))$$
$$\ldots$$
$$CE_{n+1} = ((CE_n\ m)\ (CE_n\ cd_n))$$

Unlike the previously employed finite tower that explicitly used a basic evaluator for evaluating *m*, the evaluator that would result from continuing this sequence into infinity cannot be represented by a single fix-point equation. We might propose the following equation:

$$E = ((E\ m)\ (E\ d))$$

However, this equation merely corresponds to a meta-circular "definition" of *E*. It is a well known fact that this does not really define anything. There is no unique solution to this equation thus it does not specify an interpreter or a language at all.

There is also a difference pertaining to procedure compatibility. In the literal tower, procedures at different levels cannot be interchanged whereas in the "static reflection" tower they can be interchanged. This is due to the nature of *meta** which was specifically written with procedure compatibility in mind.

Before going on let's first introduce some terminology. Two languages are *related* if both are a customisation of the same open implementation. For example in the drawing of the finite tower in section 3.2 the evaluators *basic-eval0* and *eval0* implement related languages since both where created from the same *meta0*. Whereas *eval0* and *eval1* do not implement related languages because they where created from different *meta*s. It is important to note that we *do not* consider *meta0* and *meta1* to be the same open implementation. Although they share the same source code, they are still distinct because they are procedures belonging to different levels and thus have different procedure representations (different number of continuations!). Note that in the example in this section all evaluators implement related languages because they where created from the same open implementation namely $meta_\infty$.

Using this terminology we can categorise open implementations into two different categories. The first is the category of "plain" open implementations in which customisation parameters are expressed in a language not necessarily related to the language they engender. The second is the category "with reflective potential" in which customisation parameters are expressed in a language that is related to the engendered language.

We can also distinguish 2 categories of reflection: static-reflection and dynamic reflection. An evaluator constructed without fix-point operations will be categorised as static reflection. If the construction involves some kind of fix-point operation than we will categorise it as dynamic reflection. Note that this fix-point need not be so direct as in the *climb* example. More exotic things like mutually recursive equations are also possible.

It is not difficult to see that any evaluator constructed without using a fix-point will always be limited beforehand in the number of levels it needs to “reflect”. Hence the term static reflection, the number of possible levels of reflection is statically limited by the construction of the evaluator. In the case of dynamic reflection there is no guaranteed statically determined upper bound on the number of reflection levels a program might require.

The “limited climb” in this section is a sample of static reflection. The number of reflection levels required is statically limited beforehand. The climb construct only works up to a limited number of levels, beyond that upper-bound it will stop and return the message “cannot-climb-further”. In section 5.2 the “unlimited” climb is an example of dynamic reflection. The number of reflection levels actually required is dynamically dependant on the execution of the program: it depends on the argument passed to *climb*.

8 Reflective Programming Languages Based on an Open Implementation

First let us consider what a reflective language is. Traditionally one considers two important requirements a language must conform to in order to be reflective [Smith...]. First it needs "an account of itself embedded

within it". In other words some kind of representation for the language must be accessible from within itself. Secondly this "self-representation" must be causally connected to the system so that changes to it directly affect the system itself.

Under this definition, what we have built does not qualify as a reflective system. It is more like a low-level "do-it-yourself kit". A global variable called *meta** contains a reference to $meta_\infty$. It is up to the user to construct recursive definitions over evaluators for defining dynamic reflection or open read-eval-print loops for constructing static reflection. This can be somewhat involved sometimes, e.g. the construction of the climb dispatcher is more complicated than need be due to need for lazy evaluation. Furthermore, this reference can be used to generate different interpreters, that can each be used to evaluate different parts of a program (an often useful property). No true support is given to manage all this.

Although it is not a reflective system in the traditional sense—it has no self representation embedded within it—it can be used to create interpreters from a description expressed in that same language. It is even possible to actually provide access to this "self-representation" from within the language itself. The user will have to do some programming to accomplish this however.

When building a real reflective system based on an open-implementation, we would use the parameters to *meta* as a self representation. Of course it is not practical to burden the user with explicitly constructing fix-points to obtain dynamic reflection etc. Normally one would determine some practical, sufficiently flexible ways for accessing the self-representation and hard-code this into the system. A suggestion for a practical implementation, that has the capability to destructively alter the dispatcher, was already given in section 6. In this case access to the self-representation could (for example) take place by storing the dispatcher table in a special global variable which is made available to the user. Less destructive forms are imaginable. For example, a reflective variant of scheme's *let*, and *letrec*, can be provided to the user as standard mechanisms for writing statically or dynamically reflective code respectively.

9 Conclusion

The traditional model of reflection is not sufficiently detailed for expressing the fact that every interpreter has a fixed “untouchable” core that cannot be affected by reflective programming. Our open-implementation approach adds some detail to fix this, dividing an interpreter into a fixed and a parameterised part.

The open-implementation view gives a better and cleaner understanding of reflection. For one thing, it improves upon the ad-hoc and obscure way levels are linked to each other in the traditional approach. Another important aspect of the open-implementation model is that it clearly exhibits the notion of theory relativity. It actually takes the fact that some parts of the system will never be represented in its (self-)representation as a premise and puts these things separately into a *meta* function. This *meta* function establishes the meta-theory and the parameters to the functions constitute the representation of a language/representation. Thus the “meta-theory” and the notion of “representation” are clearly defined before we even start thinking about reflection. Reflection is then obtained by making a representation (parameters to *meta*) available from within the language.

It is our opinion that this alternative view on reflective systems will strongly influence the definition, implementation and theory of reflective systems. In fact it can now be argued that a large part of the literature on reflective systems is devoted to "variations on open implementations" for particular systems (e.g. alternative open implementations for Scheme). Which is an important topic, of course, but a topic that can be considered as a research topic that is more general than reflective systems.

The good news is that, given this alternative view on reflection, it is possible to start considering a generalised theory of reflective systems. As pointed out in [Mendhekar,Friedman93] when the view taken on reflective systems is: base system + reflective operators (the traditional view) then: "*we can never hope to have a generalised theory about reflective systems since the theory will have to take into account the operational behaviour of every base system*". This problem is entirely resolved when reflection is based on open implementations: the operational behaviour of the base system has already been taken care of in defining the open implementation, and is a prerequisite before turning a system into a reflective one.

10 References

[Bawden88] A. Bawden: **Reification without Evaluation**, Conference Record of the 1988 ACM Symposium on LISP and Functional Programming, 1988.

[desRivières&Smith84] J. des Rivières and B. C. Smith: **The Implementation of Procedurally Reflective Languages**, Conference Record of the 1984 ACM Symposium on LISP and Functional Programming, pp 331-347, Austin, Texas (August 1984)

[Friedman&Wand84] D.P. Friedman, and M. Wand: **Reification: Reflection without Metaphysics**, Conference Record of the 1984 ACM Symposium on LISP and Functional Programming, pp 348-355, Austin, Texas (August 1984)

[Jefferson&Friedman92] S. Jefferson, and D.P. Friedman: **A Simple Reflective Interpreter**, IMSA'92 International Workshop on Reflection and Meta-Level Architecture, Tokyo, November 4-7, 1992. (1992)

[Kickzales, des Rivières,Bobrow91] G. Kickzales, J. des Rivières, and D. G. Bobrow: **The Art of the Metaobject Protocol**, The MIT Press, Cambridge, Massachusetts, 1991.

[Maes87] P. Maes: **Computational Reflection**, VUB AI-Lab technical report 87-2. (1987)

[Maes88] P. Maes: **Issues in Computational Reflection**, Meta-Level Architectures and Reflection, P. Maes and D. Nardi (eds.) Elsevier Publishers B.V. (North-Holland). (1988)

[Mendhekar,Friedman93] A. Mendhekar, D.P. Friedman: **Towards a Theory of Reflective Programming Languages.** In informal proceedings of the OOPSLA'93 workshop on Object-Oriented Reflections and Meta-level Architectures, October 1993.

[Rao91] R. Rao: **Implementational Reflection in Silica**, Lecture Notes in Computer Science, P. America (ed.), ECOOP'91, European Conference on Object-Oriented Programming, Springer Verlag. (1991)

[Simmons&Friedman92] J.W. Simmons II, and D.P. Friedman: **A Reflective System is as Extensible as its Internal Representations: An Illustration**, Indiana University Computer Science Department Technical Report #366. (1992)

[Simmons,Jefferson&Friedman92] J.W. Simmons II, S. Jefferson, and D.P. Friedman: **Language Extensions via First-class Interpreters**, Indiana University Computer Science Department Technical Report #362. (1992)

[Smith84] B. C. Smith: **Reflection and Semantics in Lisp**, Conf. Rec 11th ACM Symp on Principles of Programming Languages (Salt Lake City, January 1984, pp23-35. (1984).

[Wand&Friedman88] M. Wand, and D. P. Friedman: **The Mystery of the Tower Revealed: A Non-Reflective Description of the Reflective Tower**, Meta-Level Architectures and Reflection, P. Maes and D. Nardi (eds.) Elsevier Publishers B.V. (North-Holland). (1988)

A Appendix: "plain" open-implementation for building finite towers

```
;-------------------------
; Open evaluator for ASEL
; possible to use meta-circularly
;-------------------------

(define meta
  (lambda (dispatch-table)

    (define evaluate
      (lambda (e r k)
        (if (pair? e)
           (find-pair (car e) dispatch-table
              (lambda (success-pair)
                ((cdr success-pair) evaluate e r k))
              (lambda ()
                (basic-evaluate e r k)))
           (basic-evaluate e r k))))

    (define basic-evaluate
      (lambda (e r k)
        ((if (constant? e)
           evaluate-constant
           (if (variable? e)
             evaluate-variable
             (if (if? e)
               evaluate-if
               (if (assignment? e)
                 evaluate-assignment
                 (if (definition? e)
                   evaluate-definition
                   (if (abstraction? e)
                     evaluate-abstraction
                     evaluate-combination))))))
         e r k)))

    (define evaluate-constant
      (lambda (e r k)
        (k (constant-part e))))

    (define evaluate-variable
      (lambda (e r k)
        (get-pair e r
           (lambda (success-pair)
             (k (cdr success-pair)))
           (lambda ()
             (wrong "symbol not bound: " e)))))

    (define wrong
      (lambda (message object)
        (display "Error:")
        (display message)
        (display object)
        (newline)))

    (define evaluate-if
      (lambda (e r k)
        (evaluate (test-part e) r
           (lambda (v)
             (if v
               (evaluate (then-part e) r k)
               (evaluate (else-part e) r k))))))

    (define evaluate-assignment
      (lambda (e r k)
        (evaluate (value-part e) r
```

```
          (lambda (v)
            (find-pair (id-part e) (car r)
               (lambda (success-pair)
                 (set-cdr! success-pair v)
                 (k (void)))
               (lambda ()
                 (set-car! global-env
                      (cons (cons (id-part e) v)
                            (car global-env)))
                 (k (void))))))))

    (define evaluate-definition
      (lambda (e r k)
        (evaluate (value-part e) r
          (lambda (v)
            (find-pair (id-part e) (car r)
               (lambda (success-pair)
                 (set-cdr! success-pair v)
                 (k (void)))
               (lambda ()
                 (set-car! r
                      (cons (cons (id-part e) v)
                            (car r)))
                 (k (void))))))))

    (define evaluate-abstraction
      (lambda (e r k)
        (k (make-compound
            (formals-part e) (body-part e) r))))

    (define evaluate-combination
      (lambda (e r k)
        ;(display "@: ")
        ;(write e)
        ;(newline)
        (evaluate (operator-part e) r
          (lambda (proc)
            (evaluate-operands (operands-part e) r
               (lambda (args)
                 (apply-procedure proc args k)))))))

    (define evaluate-operands
      (lambda (operands r k)
        (if (null? operands)
          (k '())
          (evaluate (car operands) r
            (lambda (v)
              (evaluate-operands (cdr operands) r
                 (lambda (w)
                   (k (cons v w)))))))))

    (define evaluate-sequence
      (lambda (body r k)
        (if (null? (cdr body))
          (evaluate (car body) r k)
          (evaluate (car body) r
            (lambda (v)
              (evaluate-sequence (cdr body) r k))))))

    (define make-compound
      (lambda (formals body r)
        (lambda (k . args)
          (evaluate-sequence body (extend r formals args) k))))

    evaluate))

(define apply-procedure
      (lambda (proc args k)
        (if (procedure? proc)
```



```
          (apply proc (cons k args))
          (wrong "operator is not a procedure" proc))))

(define extend
    (lambda (r ids vals)
      (cons (extend-frame '() ids vals) r)))

(define extend-frame
  (lambda (f ids vals)
    (if (null? ids)
      f
      (if (pair? ids)
        (extend-frame (cons (cons (car ids) (car vals)) f)
                      (cdr ids)
                        (cdr vals))
        (cons (cons ids vals) f)))))

(define get-pair
    (lambda (id r success failure)
      (if (null? r)
        (failure)
        (find-pair id (car r)
           success
           (lambda ()
             (get-pair id (cdr r) success failure))))))

(define find-pair
   (lambda (elt alist success failure)
     ((lambda (assq-result)
        (if assq-result
          (success assq-result)
          (failure)))
      (assq elt alist))))

(define empty-env '())

(define 1st (lambda (l) (car l)))
(define 2nd (lambda (l) (car (cdr l))))
(define 3rd (lambda (l) (car (cdr (cdr l)))))
(define 4th (lambda (l) (car (cdr (cdr (cdr l))))))
(define 5th (lambda (l) (car (cdr (cdr (cdr (cdr l)))))))

(define test-tag
  (lambda (tag)
    (lambda (e)
      (if (pair? e) (eq? (car e) tag) #f))))

(define make-primitive ;;only for "non-higher order" primitives
  (lambda (op)
    (lambda (k . args)
      (k (apply op args)))))

(define primitive-identifiers
   (lambda ()
    '(car cdr cons set-car! set-cdr! assq memq
      null? = eq? newline write display read
      + - * symbol? list pair? eof-object?
      close-input-port open-input-file void procedure?)))

(define primitive-procs
  (lambda ()
    (list car cdr cons set-car! set-cdr! assq memq
      null? = eq? newline write display read
      + - * symbol? list pair? eof-object?
      close-input-port open-input-file void procedure?)))

(define variable? symbol?)
(define if? (test-tag 'if))
(define assignment? (test-tag 'set!))
```

```
(define definition? (test-tag 'define))
(define abstraction? (test-tag 'lambda))
(define quote? (test-tag 'quote))

(define constant?
  (lambda (e)
    (if (pair? e) (quote? e)
      (if (symbol? e) #f #t))))

(define constant-part
    (lambda (e) (if (quote? e) (2nd e) e)))

(define test-part 2nd)
(define then-part 3rd)
(define else-part 4th)

(define id-part 2nd)
(define value-part 3rd)

(define formals-part 2nd)
(define body-part (lambda (e) (cdr (cdr e))))

(define operator-part 1st)
(define operands-part cdr)

(define void
    ((lambda (v) (lambda () v)) (cons '* '*)))

(define mapper
      (lambda (f l)
        (if (null? l)
          '()
          (cons (f (car l)) (mapper f (cdr l))))))

(define initialize-global-env
      (lambda ()
        (set! global-env
           (extend
             empty-env
             (cons 'apply (primitive-identifiers))
             (cons (lambda (k proc args)
                     (apply-procedure proc args k))
                   (mapper make-primitive (primitive-procs)))))))

(define default-dispatcher '())

(define basic-eval (meta default-dispatcher))

(define openloop
  (lambda (evaluate read-prompt write-prompt)
    (display read-prompt)
    (evaluate (read) global-env
      (lambda (v)
        (display write-prompt)
        (if (eq? v (void))
          "Nothing is displayed"
          (write v))
        (newline)
        (openloop evaluate read-prompt write-prompt)))))

(define loadfile
  (lambda (evaluate file)
    ((lambda (port)
       ((lambda (loop)
          (set! loop
            (lambda (v)
              (if (eof-object? v)
                (close-input-port port)
                (evaluate v global-env
```



```
                  (lambda (ignore)
                    (loop (read port)))))))
          (loop (read port)))
        '*))
     (open-input-file file))))

(define boot-1-level
  (lambda (evaluate input-prompt output-prompt)
    (initialize-global-env)
    (loadfile basic-eval this-file-name)
    (openloop evaluate input-prompt output-prompt)))

(define start
  (lambda (evaluate input-prompt output-prompt)
    (initialize-global-env)
    (openloop evaluate input-prompt output-prompt)))

(define global-env 'dummy);;just so that global-env exists and can be set! to

(define this-file-name "open-simple.scm")
```

B. Open Implementation “with reflective potential” for fix-point equations

```
;--------------------------
; Open evaluator for ASEL with reflective potential
; read this file into scheme and then evaluate `(start)'

;;---- Stack of meta continuations for simulating an infinite tower

(define make-default-stack
  (lambda (level)
    (list 'cstack level (- level 1))))

(define push
  (lambda (base-stack cont)
    (list 'cstack cont base-stack)))

(define pop
  (lambda (stack)
    (if (number? (3rd stack))
        (make-default-stack (3rd stack))
        (3rd stack))))

(define top
  (lambda (stack)
    (if (number? (2nd stack))
      (make-loop (2nd stack))
      (2nd stack))))

;; procedure for creating the revpl procedure for default continuations
(define make-loop
  (lambda (level)
    (define loop
      (lambda (m v)
        (display level)
        (display ": ")
        (if (eq? v (void))
          "nothing is displayed"
          (write v))
        (newline)
        (display level)
        (display "> ")
        (basic-eval m (read) global-env loop)))
    loop))

;;sometimes we need something that behaves like (lambda (v) v) as
; continuation
(define id-cont
```

```
  (push 'should-not-be-used (lambda (m v) v)))

;;---------
;This file is based on a copy of "open-simple.scm"
;It has been converted a bit to simulate an infinite tower.
;All lambdas have been replaced by similar lambdas with an extra first
;argument:
;a stack of meta-continuations
;all calls to such procedures similarly have been converted to pass on a
;stack of meta continuations.
;Note that continuations are also procedures and thus also have to receive
;a stack of meta-continuations as first argument.
;
;The following variable names are used throughout the file
;m : stack of meta continuations
;k : continuation
;e : expression
;r : environment
;;---------

(define meta
  (lambda (m dispatch-table)

    (define evaluate
      (lambda (m e r k)
        (if (pair? e)
           (find-pair (car e) dispatch-table
              (lambda (success-pair)
                ((cdr success-pair) m evaluate e r k))
              (lambda ()
                (basic-evaluate m e r k)))
           (basic-evaluate m e r k))))

    (define basic-evaluate
      (lambda (m e r k)
        ((if (constant? e)
           evaluate-constant
           (if (variable? e)
             evaluate-variable
             (if (if? e)
               evaluate-if
               (if (assignment? e)
                 evaluate-assignment
                 (if (definition? e)
                   evaluate-definition
                   (if (abstraction? e)
                     evaluate-abstraction
                     evaluate-combination))))))
         m e r k)))

    (define evaluate-constant
      (lambda (m e r k)
        (k m (constant-part e))))

    (define evaluate-variable
      (lambda (m e r k)
        (get-pair e r
           (lambda (success-pair)
             (k m (cdr success-pair)))
           (lambda ()
             (wrong m "symbol not bound: " e)))))

    (define evaluate-if
      (lambda (m e r k)
        (evaluate m (test-part e) r
           (lambda (m v)
             (if v
               (evaluate m (then-part e) r k)
               (evaluate m (else-part e) r k))))))
```



```
(define evaluate-assignment
  (lambda (m e r k)
    (evaluate m (value-part e) r
       (lambda (m v)
         (find-pair (id-part e) (car r)
            (lambda (success-pair)
              (set-cdr! success-pair v)
              (k m (void)))
            (lambda ()
              (set-car! global-env
                   (cons (cons (id-part e) v)
                         (car global-env)))
              (k m (void))))))))

(define evaluate-definition
  (lambda (m e r k)
    (evaluate m (value-part e) r
       (lambda (m v)
         (find-pair (id-part e) (car r)
            (lambda (success-pair)
              (set-cdr! success-pair v)
              (k m (void)))
            (lambda ()
              (set-car! r
                   (cons (cons (id-part e) v)
                         (car r)))
              (k m (void))))))))

(define evaluate-abstraction
  (lambda (m e r k)
    (k m (make-compound
          (formals-part e) (body-part e) r))))

(define evaluate-combination
  (lambda (m e r k)
    ;(display "@: ")
    ;(write e)
    ;(newline)
    (evaluate m (operator-part e) r
       (lambda (m proc)
         (evaluate-operands m (operands-part e) r
            (lambda (m args)
              (apply-procedure m proc args k)))))))

(define evaluate-operands
  (lambda (m operands r k)
    (if (null? operands)
      (k m '())
      (evaluate m (car operands) r
         (lambda (m v)
           (evaluate-operands m (cdr operands) r
              (lambda (m w)
                (k m (cons v w)))))))))

(define evaluate-sequence
  (lambda (m body r k)
    (if (null? (cdr body))
      (evaluate m (car body) r k)
      (evaluate m (car body) r
        (lambda (m v)
          (evaluate-sequence m (cdr body) r k))))))

(define make-compound
  (lambda (formals body r)
    (lambda (m . args)
      (evaluate-sequence (pop m) body (extend r formals args) (top m)))))

((top m) (pop m) evaluate)))
```

```
(define wrong
  (lambda (m message object)
    (display "Error:")
    (display message)
    (display object)
    (newline)
    ((top m) (pop m) 'error)))

(define apply-procedure
      (lambda (m proc args k)
        (if (procedure? proc)
          (apply proc (cons (push m k) args))
          (wrong m "operator is not a procedure" proc))))

(define extend
    (lambda (r ids vals)
      (cons (extend-frame '() ids vals) r)))

(define extend-frame
  (lambda (f ids vals)
    (if (null? ids)
      f
      (if (pair? ids)
        (extend-frame (cons (cons (car ids) (car vals)) f)
                      (cdr ids)
                        (cdr vals))
        (cons (cons ids vals) f)))))

(define get-pair
    (lambda (id r success failure)
      (if (null? r)
        (failure)
        (find-pair id (car r)
           success
           (lambda ()
             (get-pair id (cdr r) success failure))))))

(define find-pair
   (lambda (elt alist success failure)
     ((lambda (assq-result)
        (if assq-result
          (success assq-result)
          (failure)))
      (assq elt alist))))

(define empty-env '())

(define 1st (lambda (l) (car l)))
(define 2nd (lambda (l) (car (cdr l))))
(define 3rd (lambda (l) (car (cdr (cdr l)))))
(define 4th (lambda (l) (car (cdr (cdr (cdr l))))))
(define 5th (lambda (l) (car (cdr (cdr (cdr (cdr l)))))))

(define test-tag
  (lambda (tag)
    (lambda (e)
      (if (pair? e) (eq? (car e) tag) #f))))

(define make-primitive ;;use only for "non-higher order" primitives
  (lambda (op)
    (lambda (m . args)
      ((top m) (pop m) (apply op args)))))

(define primitive-identifiers
   (lambda ()
    '(car cdr cons set-car! set-cdr! assq memq
      null? = eq? newline write display read
      + - * symbol? list pair? eof-object?
```



```
      close-input-port open-input-file void procedure?)))

(define primitive-procs
  (lambda ()
    (list car cdr cons set-car! set-cdr! assq memq
      null? = eq? newline write display read
      + - * symbol? list pair? eof-object?
      close-input-port open-input-file void procedure?)))

(define variable? symbol?)
(define if? (test-tag 'if))
(define assignment? (test-tag 'set!))
(define definition? (test-tag 'define))
(define abstraction? (test-tag 'lambda))
(define quote? (test-tag 'quote))

(define constant?
  (lambda (e)
    (if (pair? e) (quote? e)
      (if (symbol? e) #f #t))))

(define constant-part
    (lambda (e) (if (quote? e) (2nd e) e)))

(define test-part 2nd)
(define then-part 3rd)
(define else-part 4th)

(define id-part 2nd)
(define value-part 3rd)

(define formals-part 2nd)
(define body-part (lambda (e) (cdr (cdr e))))

(define operator-part 1st)
(define operands-part cdr)

(define void
    ((lambda (v) (lambda () v)) (cons '* '*)))

(define mapper
      (lambda (f l)
        (if (null? l)
          '()
          (cons (f (car l)) (mapper f (cdr l))))))

(define openloop
  (lambda (m evaluate read-prompt write-prompt)
    (display read-prompt)
    (evaluate m (read) global-env
      (lambda (m v)
        (display write-prompt)
        (if (eq? v (void))
          "Nothing is displayed"
          (write v))
        (newline)
        (openloop m evaluate read-prompt write-prompt)))))

(define initialize-global-env
      (lambda ()
        (set! global-env
           (extend
             empty-env
             (cons 'apply (primitive-identifiers))
             (cons (lambda (m proc args)
                     (apply-procedure (pop m) proc args (top m)))
                   (mapper make-primitive (primitive-procs)))))
        (set! global-env
             (extend global-env
```

```
                      '(meta* default-dispatcher openloop)
                  (list meta  default-dispatcher openloop)))))

(define default-dispatcher '())

(define basic-eval (meta id-cont default-dispatcher))

(define start
  (lambda ()
    (initialize-global-env)
    (set-car! global-env (cons (cons 'global-env global-env) (car global-env)))
    (let ((s (make-default-stack 0)))
      ((top s) (pop s) 'begin))))

(define boot
  (lambda ()
    (initialize-global-env)
    (let ((s (make-default-stack 0)))
      ((top s) (pop s) 'begin))))

(define global-env 'dummy)
;;just so that global-env exists and can be set! to
```