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.