LEABHARLANN CHOLÁISTE NA TRÍONÓIDE, BAILE ÁTHA CLIATH | TRINITY COLLEGE LIBRARY DUBLIN
Ollscoil Átha Cliath | The University of Dublin

Terms and Conditions of Use of Digitised Theses from Trinity College Library Dublin

Copyright statement

All material supplied by Trinity College Library is protected by copyright (under the Copyright and Related Rights Act, 2000 as amended) and other relevant Intellectual Property Rights. By accessing and using a Digitised Thesis from Trinity College Library you acknowledge that all Intellectual Property Rights in any Works supplied are the sole and exclusive property of the copyright and/or other IPR holder. Specific copyright holders may not be explicitly identified. Use of materials from other sources within a thesis should not be construed as a claim over them.

A non-exclusive, non-transferable licence is hereby granted to those using or reproducing, in whole or in part, the material for valid purposes, providing the copyright owners are acknowledged using the normal conventions. Where specific permission to use material is required, this is identified and such permission must be sought from the copyright holder or agency cited.

Liability statement

By using a Digitised Thesis, I accept that Trinity College Dublin bears no legal responsibility for the accuracy, legality or comprehensiveness of materials contained within the thesis, and that Trinity College Dublin accepts no liability for indirect, consequential, or incidental, damages or losses arising from use of the thesis for whatever reason. Information located in a thesis may be subject to specific use constraints, details of which may not be explicitly described. It is the responsibility of potential and actual users to be aware of such constraints and to abide by them. By making use of material from a digitised thesis, you accept these copyright and disclaimer provisions. Where it is brought to the attention of Trinity College Library that there may be a breach of copyright or other restraint, it is the policy to withdraw or take down access to a thesis while the issue is being resolved.

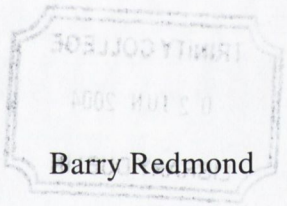
Access Agreement

By using a Digitised Thesis from Trinity College Library you are bound by the following Terms & Conditions. Please read them carefully.

I have read and I understand the following statement: All material supplied via a Digitised Thesis from Trinity College Library is protected by copyright and other intellectual property rights, and duplication or sale of all or part of any of a thesis is not permitted, except that material may be duplicated by you for your research use or for educational purposes in electronic or print form providing the copyright owners are acknowledged using the normal conventions. You must obtain permission for any other use. Electronic or print copies may not be offered, whether for sale or otherwise to anyone. This copy has been supplied on the understanding that it is copyright material and that no quotation from the thesis may be published without proper acknowledgement.

Supporting Unanticipated Dynamic Adaptation of Object-Oriented Software

A thesis submitted to the
University of Dublin, Trinity College,
in fulfillment of the requirements for the degree of
Doctor of Philosophy (Computer Science)



Barry Redmond

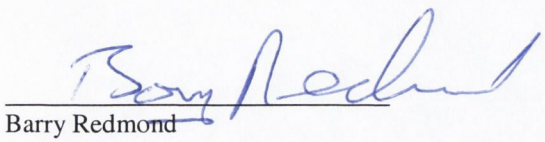
Department of Computer Science
Trinity College, Dublin

March 2004

TRINITY COLLEGE
0 2 JUN 2004
LIBRARY DUBLIN
THOBIS
~~7949~~
7488

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this or any other University, and that, unless otherwise stated, it is entirely my own work.

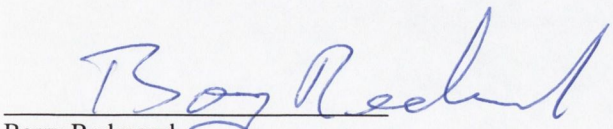


Barry Redmond

31st March 2004

Permission to Lend and / or Copy

I, the undersigned, agree that the Trinity College Library may lend and / or copy this thesis upon request.



Barry Redmond

31st March 2004

Summary

Dynamic adaptation of a running program allows the program's behaviour to be changed without stopping or restarting it. Examples of the need for this are in updating long-running systems that cannot be halted and in adapting mobile systems to transient changes in their environment. While it is possible to support dynamic adaptation when it is anticipated during the system design, it is difficult, if not impossible, to anticipate all the ways in which a system may need to be dynamically adapted. One way to address this problem is to use a software architecture that supports modifications to the running program without requiring any static preparation or anticipation of the modifications in the program design or implementation. Such architectures support unanticipated dynamic adaptation.

This thesis addresses unanticipated dynamic adaptation of object-oriented software. A small number of architectures that support unanticipated dynamic adaptation of object-oriented systems already exist, but they have some limitations. Most of them operate by associating components implementing the behavioural change with specified operations on selected classes and objects in a program at runtime. While they all support unanticipated dynamic adaptation they do so in a way that allows an unstructured association of behavioural change with classes and objects and therefore leads to a degradation of the inheritance and instance-of relationships in the program. For example, behavioural change associated with a class may not be associated with its subclasses, and behavioural change intended to be associated with all instances of a class may be missing for some instances. The undisciplined association of behavioural change with classes and objects in a program interferes with the conventional inheritance structure, leading to inconsistencies in the presence of dynamic adaptation.

This thesis proposes a new architectural model that supports unanticipated dynamic adaptation and addresses the limitations described above. The architecture uses the concept of a metatype to represent the behavioural changes introduced by unanticipated dynamic adaptation, and supports a simplified and ordered approach to manipulating dynamic changes as reusable independent entities. Metatypes extend the conventional notion of a type to include behavioural adaptation. They provide a structure and predictable scope for associating separately implemented behavioural change components with classes and objects in unanticipated combinations.

The proposed architecture supports solutions to a number of acknowledged challenge problems in the field of dynamic adaptation. These include the ability to ensure that behavioural change is inherited even when affected methods are overridden, a stronger separation between the program and the components implementing the behavioural change, and the ability to make the behavioural change sensitive to execution context. Strong separation supports reusability of both the program and the behavioural change components, and enables the ability to associate behavioural change with arbitrary classes and objects without anticipation.

The architecture brings control, constraint and a more principled approach to the use of reflection for unanticipated dynamic adaptation. The new architecture has been implemented for the Java language as Iguana/J and uses reflective techniques to alter the behaviour of a program. It has a significantly smaller and simpler class library than existing architectures. The implementation adds behavioural reflection to the existing structural reflection of Java, without extending the language.

This thesis reviews some existing architectures that support unanticipated dynamic adaptation for object-oriented systems. It describes the programmer's model and implementation of Iguana/J, and it evaluates the architecture by comparing it to the others, particularly by reference to acknowledged challenge problems in the field.

Acknowledgements

While only my name appears on this thesis many people helped and encouraged me along the way. First, I want to express my thanks to my supervisor for the last six years, Dr. Vinny Cahill. He provided just the right balance of guidance, prodding, and questioning that allowed me to find my own path. I will forever have a voice in my ear asking 'how?' and 'why?' and 'where?'. His interest in the work of each of his many students is illustrated by his ability to keep up with and remember the details of each one. I still don't understand how he does that.

I want to thank everyone in the Distributed Systems Group. Being a member of DSG is an education in itself. I particularly want to thank the others of the original reflection four – Jim Dowling, Peter Haraszti, and Tilman Schäfer. The various discussions we have had have contributed to many of the ideas in this thesis. The questions asked by John Keeney (my user community) and Donal Lafferty also prompted many useful research directions.

I also want to thank John Murray who provided long distance support, suggestions and insight.

Finally, and most of all, I want to thank my ever patient family. Bairbre, Niamh and Niall have put up with much, have covered for me when I evaded my duties or skipped my turn, have encouraged me, fed me, and have never complained. Thank you.

We must bear in mind, then, that there is nothing more difficult and dangerous, or more doubtful of success, than an attempt to introduce a new order of things in any state.

- Niccolò Machiavelli
The Prince, 1515.

Contents

Chapter 1: Introduction **1**

1.1 Unanticipated Dynamic Adaptation 2

1.1.1 Evolution, adaptation, and anticipation 2

1.1.2 Structural and behavioural adaptation 3

1.1.3 Behavioural adaptation by association 5

1.1.4 Runtime support 6

1.1.5 Other issues 6

1.2 Background 7

1.2.1 Separation of Concerns 7

1.2.2 Functional and non-functional concerns 8

1.2.3 Aspect Oriented Programming 9

1.2.4 Reflection 10

1.3 Challenge Problems 12

1.3.1 Reusability 12

1.3.2 Context sensitivity 13

1.3.3 Inheritance anomaly 14

1.4 Contribution 15

1.5 Roadmap 16

1.6 Summary 16

Chapter 2: Dynamic Adaptation Architectures **17**

2.1 Categorisation 17

2.2 Static Adaptation 19

2.3 Anticipated Dynamic Adaptation 21

2.4 Unanticipated Dynamic Adaptation 23

2.5 MetaXa 24

2.5.1 Programmer's model 25

2.5.2 Composition 26

2.5.3 Association 27

2.5.4 Inheritance and refinement of behavioural change 28

2.5.5 Anticipation, reusability, and separation of concerns 28

2.5.6 Implementation 29

2.5.7 Summary of MetaXa 30

2.6 Guaraná 30

2.6.1 Programmer's model 31

2.6.2 Composition model 33

2.6.3 Association 33

2.6.4 Inheritance and refinement of behavioural change 34

2.6.5 Anticipation, reusability, and separation of concerns 35
2.6.6 Implementation 35
2.6.7 Summary of Guaraná 36
2.7 PROSE 36
2.7.1 Programmer's model 36
2.7.2 Composition model 38
2.7.3 Association 38
2.7.4 Inheritance and refinement of behavioural change 39
2.7.5 Anticipation, reusability, and separation of concerns 39
2.7.6 Implementation 40
2.7.7 Summary of PROSE 40
2.8 Summary 41
Chapter 3: The Iguana/J Programmer's Model **42**
3.1 Introduction 42
3.2 Objectives 43
3.3 Overview 46
3.4 Iguana/J classes 48
3.5 Metaobject protocols 49
3.6 Metaobject protocol classes 51
3.7 Access to context information 53
3.8 Protocol derivation and metaobject composition 55
3.8.1 Protocol derivation 55
3.8.2 Metaobject composition 57
3.8.3 Truncation of the composition chain 58
3.8.4 Metaobject compatibility 59
3.8.5 Multiple instances of a metaobject class 59
3.8.6 Partially dynamic derivation 60
3.8.7 One metaobject per category 61
3.9 The null protocol 61
3.10 Association 63
3.10.1 Association with classes 63
3.10.2 Dynamic initiation of association 64
3.10.3 Static initiation of association 65
3.10.4 Wildcards 66
3.10.5 Association with objects 66
3.11 Shared metaobjects 67
3.12 Protocol parameters 69
3.13 Dynamic association 72
3.14 Metatypes 73
3.14.1 The meaning of metatype 73
3.14.2 Metatypes and composition 74
3.14.3 Dependence and independence 75

3.14.4 Metatype relationships 77
3.14.5 Metatype rules 79
3.15 Summary 80
Chapter 4: Implementation **82**
4.1 Introduction 82
4.2 Overview 83
4.3 Integrating with the JVM 85
4.4 Trapping class load 86
4.4.1 Extending class and object structure 86
4.4.2 Access to JVM data structures 87
4.4.3 Functions of the library 87
4.5 Association 88
4.5.1 Implementing association 88
4.5.2 Creating an association 88
4.5.3 Association with new objects 89
4.5.4 Interaction between object associations 89
4.5.5 Inheritance of association 90
4.5.6 Association with an existing class 91
4.5.7 Shared metaobjects 91
4.5.8 Changing association 92
4.5.9 Package-wide association 92
4.6 Composition 94
4.7 Interception 94
4.8 Transfer to meta-level 96
4.9 Recursive reflection 98
4.10 Protocol classes 99
4.11 Metatype rules 100
4.12 Null metaobject protocol 100
4.13 Class loaders 101
4.14 Exceptions 102
4.15 Summary 104
Chapter 5: Evaluation **105**
5.1 Evaluation approach 106
5.2 Reusability 107
5.2.1 The reusability challenge 107
5.2.2 The Iguana/J solution 108
5.2.3 Other solutions 109
5.2.4 Summary of reusability 111
5.3 Context sensitivity 111
5.3.1 The context sensitivity challenge 112
5.3.2 The Iguana/J solution 112
5.3.3 Other solutions 114

5.3.4 Summary of context sensitivity 114
5.4 Inheritance anomaly 114
5.4.1 The inheritance anomaly challenge 114
5.4.2 The Iguana/J solution 116
5.4.3 Other solutions 119
5.4.4 Summary of inheritance anomaly 120
5.5 Separation of concerns 120
5.5.1 The separation of concerns challenge 121
5.5.2 The Iguana/J solution 121
5.5.3 Other solutions 124
5.5.4 Summary of separation of concerns 124
5.6 Inheritance of change 125
5.6.1 The inheritance challenge 125
5.6.2 The Iguana/J solution 127
5.6.3 Other solutions 127
5.6.4 Summary of inheritance 129
5.7 Performance 129
5.7.1 Iguana/J performance 130
5.7.2 Relative performance 132
5.8 Summary 133
Chapter 6: Conclusion **135**
6.1 Achievements 135
6.1.1 Structure 136
6.1.2 Simplification 136
6.1.3 Constraint 136
6.1.4 Challenges 137
6.2 Future work 138
6.2.1 Dynamic derivation of protocols 138
6.2.2 Alternative composition strategies 138
6.2.3 Security implications 138
6.2.4 Alternative implementation techniques 139
6.2.5 More abstract support frameworks 139
6.2.6 Static type checking 139

List of Figures

Figure 1.1: Adaptation by associating a behavioural change component. 6
Figure 1.2: The three independent development roles 9
Figure 1.3: An outline of the structure of a reflective architecture 11
Figure 2.1: A MetaXa metaobject to trace calls to all methods of a given object. 26
Figure 2.2: A fragment of code to create and associate a MetaXa metaobject. 28
Figure 2.3: A Guaraná metaobject to trace all method calls and field accesses. 31
Figure 2.4: A fragment of code to create and associate a Guaraná metaobject. 34
Figure 2.5: A PROSE crosscut containing advice to announce method invocations 37
Figure 2.6: Placing the crosscut in a PROSE aspect 37
Figure 2.7: Inserting (and removing) the PROSE aspect 37
Figure 3.1: The two levels of the Iguana/J model. 43
Figure 3.2: Example of a metaobject class to handle method execution. 46
Figure 3.3: A protocol declaration that connects class `Trivial` to invocation 47
Figure 3.4: Associating a named base-level class with a named protocol. 47
Figure 3.5: UML diagram of the class structure for the example in figures 3.2, 3.3 and 3.4. 47
Figure 3.6: The eleven classes of the Iguana/J library 48
Figure 3.7: A protocol declaration with three reification categories. 50
Figure 3.8 The role of a meta-level interface object. 52
Figure 3.9: Deriving a metaobject protocol. 55
Figure 3.10: A metaobject protocol derivation tree. 56
Figure 3.11: Intentional truncation of the composition chain. 58
Figure 3.12: Multiple path protocol derivation 59
Figure 3.13: A protocol and two metaobject classes that expect parameter values. 70
Figure 3.14: A derived protocol `SyncProt`, and its metaobject, that passes parameter values to its parent. 71
Figure 3.15: Accumulation of metatypes by derived protocols. 74
Figure 3.16: A simple logging protocol and its metaobject class. 76
Figure 3.17: A dependent derived protocol and its metaobject class. 76
Figure 3.18: A non-dependent derived protocol, and its metaobject class. 76
Figure 3.19: Composing `AccessProt` and `SyncProt`. 76
Figure 3.20: A metatype structure in parallel with a type structure 80
Figure 4.1: Overview of the elements of the implementation. 84
Figure 4.2: The three types of event that activate Iguana/J. 87
Figure 4.3: The meta-level interface object refers to metaobjects reifying each operation category. 88
Figure 4.4: Conflict between package-wide and inherited association 93
Figure 4.5: A sample protocol P2 derived from P1 and reifying two categories. 99
Figure 4.6: A simplified fragment of the class created by compiling protocol P2. 99
Figure 4.7: Cases of exception sources and handling 102

Figure 5.1: One behavioural change component should be reusable with different classes. 108
Figure 5.2: An Iguana/J metaobject class to implement synchronisation. 109
Figure 5.3: An Iguana/J protocol using the synchronisation metaobject. 109
Figure 5.4: Code fragments to initiate association of `SyncProtocol` with `Stack` and `Queue`. 109
Figure 5.5: An outline of the `List` class, in which one public method uses another. 112
Figure 5.6: Event generation as a behavioural change component. 112
Figure 5.7: An Iguana/J metaobject class that generates an event depending on context 113
Figure 5.8: A protocol that includes the event generator metaobject. 113
Figure 5.9: A fragment of code to associate the protocol with the `List` class. 114
Figure 5.10: The basic `Buffer` class. 115
Figure 5.11: The subclass adds a `popLast()` method that uses shared fields. 115
Figure 5.12: Synchronisation should be inherited and extended for a subclass 116
Figure 5.13: Inheritance and extension of behavioural change using Iguana/J 116
Figure 5.14: A metaobject class to apply mutual and self exclusion to two methods. 117
Figure 5.15: A protocol that reifies invocation using the `SyncTwo` metaobject class. 117
Figure 5.16: A metaobject class that provides mutual exclusion between one method and each of two others. 118
Figure 5.17: A derived protocol that reifies invocation using `SyncThree` and `SyncTwo`. 118
Figure 5.18: For a strong separation of concerns the associator must be independent. 121
Figure 5.19: Access to instances of `Channel` must be redirected to instances of `NewChannel` 122
Figure 5.20: A metaobject class to redirect method invocations. 122
Figure 5.21: A protocol that uses the `Redirect` metaobject class. 122
Figure 5.22: Outline of a metaobject class to implement an association console. 123
Figure 5.23: Outline of a display class with login and data frames. 126
Figure 5.24: The simple login and data windows created by the `Display` class. 126
Figure 5.25: A metaobject class to implement an inactivity timeout 128
Figure 5.26: A protocol to use the `Timeout` metaobject class. 128
Figure 5.27: Applying a timeout of 30 seconds to all graphical components. 128

List of Tables

Table 2.1: Software adaptation architectures that support static mode adaptation 18
Table 2.2: Software adaptation architectures that support anticipated dynamic mode adaptation 18
Table 2.3: Software adaptation architectures that support unanticipated dynamic mode adaptation 18
Table 3.1: Classes and handler methods for the seven behavioural reification categories. 49
Table 3.2: Structural reification classes used by Iguana/J. 49
Table 4.1: Comparison of interception techniques. 95
Table 5.1: Normalised performance figures for Iguana/J. 131
Table 5.2: Some performance characteristics of architectures supporting unanticipated dynamic adaptation. 132
Table 5.3: Comparison of characteristics of Iguana/J and other architectures 134

Chapter 1

Introduction

Study without reflection is a waste of time, reflection without study is dangerous.

- Attributed to Confucius.

551 – 479 BC.

Dynamic adaptation of a running program allows the program's behaviour to be changed without stopping or restarting it. Examples of the need for this are in systems that must adapt to changes in their environment, and in long-running systems that must be updated but cannot be halted. While it is possible to support dynamic adaptation when it is anticipated during the system design, it is difficult, if not impossible, to anticipate all the ways in which a system may need to be dynamically adapted. One way to address this problem is to use a structure of elements (an architecture) at runtime that supports modifications to the running program without requiring any static preparation or anticipation of the modifications in the program's design or implementation. Such architectures support *unanticipated* dynamic adaptation.

This thesis presents a model for supporting unanticipated dynamic adaptation of object-oriented software systems. Existing architectures allow undisciplined adaptation of classes and objects, do not solve some important adaptation problems, and have large or complex class libraries. The model presented here supports disciplined and structured adaptation, addresses open issues in dynamic adaptation, and has a simple class library. The model is implemented as the Iguana/J architecture.

The remainder of this chapter explains the concepts underlying unanticipated dynamic adaptation and outlines some of the issues in supporting it. It describes the background to this research, sets out the scope of this thesis, and outlines its contribution to the body of knowledge. Chapter 2 presents a review of some adaptation architectures, chapter 3 discusses the programmer's model proposed by this thesis, chapter 4 explains the implementation of the model as Iguana/J, and chapter 5 evaluates the model. Chapter 6 presents the conclusions of the thesis.

1.1 Unanticipated Dynamic Adaptation

Many modern software systems are used in situations where behavioural changes must be applied dynamically, without stopping and restarting the system. There is a range of situations in which this requirement may arise, including updating mission-critical systems while they are running and supporting adaptation of mobile systems to temporary changes in their environment. Updating mission-critical systems dynamically is usually required because the interruption in service that would be caused by a restart cannot be tolerated. Examples of this type of system are large financial support systems, or communications systems [Hic01]. The changes to be applied dynamically to such systems are typically version upgrades due to the normal software maintenance process [Gup96]. Possible reasons for this type of change include corrections of errors, extensions of functionality, and updates to use new component libraries. For mobile software systems changes are more usually used to adapt to temporary variations in the system's environment. Typical reasons for change include coping with variations in network connectivity, or arrival or departure of distributed resources [Ama02] [Mal01]. To be most useful to the user changes to a mobile system should be carried out dynamically, without a system restart.

Whether a change is needed to update a large mission-critical system or to adapt a mobile system to variations in its environment the common requirement is that the software of the system must be adapted dynamically. Dynamic adaptation of software systems has been a subject of research for many years. Fabry, for example, proposed indirection as a mechanism to support module changes [Fab76]. Segal referred to dynamic updating as a means of maintaining large procedural software systems without restarting them [Seg89]. In more recent times research on dynamic adaptation has focused on object-oriented systems (e.g. [Kic91], [Kle96], [Kic97], [Tar00a]).

This research addresses unanticipated dynamic adaptation of object-oriented software systems. The following subsections set out the area of interest in more detail.

1.1.1 Evolution, adaptation, and anticipation

The terms *software adaptation* and *software evolution* are sometimes used interchangeably to mean any change to a software system (e.g. [Kni02]), but the term 'software evolution' has been frequently used with the more specific meaning of software upgrade or maintenance (e.g. [Gup96], [Leh96], [Hur96], [Ber97], and [Mez97]). 'Evolution' usually refers to alterations in response to persistent shifts in system requirements, and over the lifetime of a software system tends to be along a linear path as alterations accumulate. Evolution of software is part of its normal life cycle [Leh96]. Evolutionary changes may be made by modifying and recompiling source code [Ber97] [Men97] (static change) or by dynamically altering a running system [Gup96] (dynamic change).

'Adaptation' is widely used to mean both persistent evolutionary change and transient changes in response to temporary variations in a system's environment. Transient changes are encountered by mobile or context-aware systems as they adapt to fluctuations in requirements and available resources [Mal01] [Cap01]. Such changes may be applied and removed as the mobile system's environment changes and reverts, and do not tend to accumulate over the lifetime of the system. This research is concerned with applying dynamic

alterations to a software system for any reason, and therefore it is concerned with both dynamic evolutionary (persistent) change and dynamic transient change. The term 'dynamic adaptation' is used here to describe both of these types of change.

Anticipation of adaptation of a software system implies preparation of the system for the change in some way. Anticipation of adaptation does not necessarily mean anticipation of dynamic adaptation. All successful software systems undergo at least maintenance changes [Leh96], and for many systems these can be applied statically. Anticipation of static adaptation can mean designing the initial structure of the system so that subsequent changes to the design at the source code level will be easier to make and will have minimal unwanted consequences [Lop96] [Mez97]. On the other hand anticipation of dynamic adaptation can occur at a number of points in the life of the software system. These range from design time, where the use of special system architectures can support later dynamic modifications (e.g. [Dow00], [Hic01], [Bru01]), to load time, where modification of the binary code as classes are loaded can be used to insert hooks for a dynamic adaptation mechanism (e.g. [Bou99], [Oga00], [Paw01]).

An examination of various anticipatory approaches shows that not only do they require anticipation at different points in the life of the system but many approaches support different kinds of dynamic adaptation (for examples see chapter 2). By choosing one approach over another a system designer or user is limiting the set of dynamic changes supported by the selected approach. Unsupported kinds of change have not been anticipated. Thus it can be seen that anticipation can be divided into anticipation of a general future need for adaptation, and anticipation of the nature of the future adaptation. As it is difficult or even impossible to anticipate all the kinds of adaptation that may be required [Kni02], inevitably there will be some adaptations that cannot be carried out dynamically even if an attempt has been made to anticipate adaptation. This limitation can be overcome by supporting *unanticipated* dynamic adaptation. That is, by providing a mechanism that allows a program to be adapted at runtime in ways that were not anticipated in the design, implementation or deployment of the program. Such mechanisms can support adaptation of systems for which source code is unavailable. They not only avoid the need to anticipate the kind of adaptations that may be required, but they can avoid or minimise the need to anticipate adaptation in the program at all.

It has been pointed out, at an almost philosophical level, that even unanticipated adaptation must be anticipated at some time [Kni02]. The same discussion points out that the reverse is also true; even anticipated adaptation must have been unanticipated at some earlier time in the design or development of the program. In other words, anticipation and unanticipation of software adaptation is simply a matter of timing. There is no generally agreed definition of 'unanticipated software adaptation' or 'unanticipated software evolution', but it has been stated that the objective of research on the subject is to maximise the kinds of intentional non-invasive changes that can be carried out [Kni02]. Non-invasive changes are those that can be applied without prior alterations or preparations of the software to be changed [Cza00].

1.1.2 Structural and behavioural adaptation

This research divides dynamic software adaptation into two kinds: *structural* adaptation and *behavioural* adaptation. These terms originated in the reflection community, where 'structural reflection' and 'behavioural (or computational) reflection' have been in use for some time to differentiate between reflection

on the structure and the behaviour of a system [Fer89] [Gol97]. As reflection is frequently used as a supporting mechanism for dynamic adaptation (see section 1.2.4) the terms have taken on corresponding meanings in the software adaptation community.

Structural adaptation is the modification of system structure by adding or removing classes or by adding, removing or changing the class members or their attributes [Ber97] [God02]. For example, adding a method to a class, or changing the signature of an existing method is structural adaptation. At a higher level, structural adaptation can also include the reconfiguration of a system by manipulating connectors and relationships between components [Dow01]. However, structural adaptation can be complex to apply dynamically. If the adaptation alters the public interface of a class a corresponding change must be made in any clients of the class that use the altered or new part of the interface. Because it affects the links between program classes it can imply widespread consequential alterations to the program. Structural adaptation that adds or alters a private member requires direct changes to the internal code of the class to make use of the change, and the code changes must be compiled (or appear to have been compiled) in the context of the class.

Behavioural adaptation is the modification of the behaviour of a class or component without altering its public interface or private structure [Wel99]. No methods or fields are added to or removed from existing classes, and no field types or method signatures are changed in existing classes. Behavioural adaptation of a class is carried out by causing code that implements the new behaviour to be executed before, after or instead of appropriate operations of the class. Operations that might be manipulated in this way include instance creation, method execution, and field access. Because there are no changes to the public interface of the class being adapted there are no direct effects on the links between classes in the program, and the adaptation does not necessarily require adjustments to the relationships between program classes.

Behavioural adaptation can be viewed as a subset of structural adaptation. Structural adaptation necessarily causes behavioural change, but behavioural adaptation does not always cause structural change. Behavioural adaptation may require extra classes or components, but that is not regarded as structural change because the public and private structure of existing program classes does not change. Behavioural adaptation can have a more localised effect than structural adaptation because the changes made are internal to a single component and do not necesarily propagate outside the component. Of course it is possible that a behavioural change to a class could establish (or remove) a relationship with another existing class in the program, and therefore behavioural adaptation could be regarded as capable of making structural change. However, this would be a consequence of only some uses of behavioural change, and would not necessarily require changes in the other class, so this possibility is not considered to change the nature of behavioural adaptation.

Behavioural adaptation can mean either insertion of extra behaviour or replacement of existing behaviour. Insertion of extra behaviour can be used to support guard-type modifications such as argument validation, security checking, synchronisation, or sending notification messages. Replacement of behaviour can be used to cancel or redirect operations such as library calls, object creation, or communications with external devices. Behavioural adaptation architectures may support one or both of these techniques.

This thesis is concerned with behavioural adaptation rather than structural adaptation. Behavioural adaptation involves direct manipulation of a program's behaviour at precise points, without the consequential

changes that can be required by structural adaptation. However, behavioural adaptation does not support the larger scale reorganisation of a system that can be supported by structural adaptation. Behavioural adaptation operates at the implementation level of operations within classes and objects, whereas structural adaptation can operate at the architectural level of connections between classes. Restricting the scope of the research to behavioural adaptation effectively means constraining the kinds of changes that can be made, but with the benefit of precise targeting of those changes.

1.1.3 Behavioural adaptation by association

To change the behaviour of part of a program, whether by insertion or by replacement, it is necessary to cause code implementing the change to be executed at appropriate times. In other words, the code execution sequence in one or more classes must be changed. There are two general techniques that may be used to do this. The first is to replace the affected code block by replacing a class, a method, or part of a method. This approach has some limitations for dynamic adaptation, mainly because it is not sufficiently flexible. For example it affects all instances of the altered class equally so it does not easily support per-instance adaptation, and it also makes it difficult to apply multiple behavioural changes at the same point in the program. Unless the change is an elementary one it can be complex to carry out. In many runtime structures inserting or replacing code in an existing class requires additions to internal tables and adjustment of existing code offsets to account for the movement of code blocks in memory. It may also require the adaptation system to deal with existing execution threads in replaced code. This replacement technique may be more suitable for evolution (persistent accumulating changes) than for short term transient adaptation. Recent work by Sun on class replacement in Java tends to support this view [Dmi01]. It suggests that the complexity and other issues involved in dynamic class replacement make it best suited to evolution during system debug.

The second general technique is to leave the original code intact but to associate a separate component implementing the behavioural change with specific points in a component in the original code (see figure 1.1). At those points the thread of execution is intercepted and the operation is redirected to the associated component for processing in accordance with the required behavioural change. New behaviour may be carried out before, after or instead of the intercepted operation. This technique has a number of benefits that make it suitable for dynamic adaptation. It is a 'cleaner' technique because the connections between the existing program and the new behaviour are at points in the program instead of spread over blocks of replaced code, and therefore the adaptation requires much fewer consequential adjustments in the program classes. It maintains a separation between the program and the components implementing the behavioural change that helps to minimise dependence between them, and it can support association of multiple behavioural changes at one point in a program. Runtime object identity checks at the interception point can be used to support per-instance adaptation.

Most architectures supporting unanticipated dynamic adaptation use some form of the association technique (e.g. [Gol99], [Oli99], and [Pop01]).

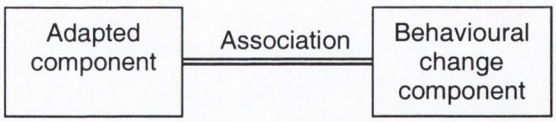


Figure 1.1: Adaptation by associating a behavioural change component.

1.1.4 Runtime support

Unanticipated dynamic adaptation requires significant runtime support. Two general supporting services are needed at runtime. First, it must be possible to identify and locate the entities in the program, such as classes, methods and fields, that are to be examined and adapted. This requires access at runtime to symbolic and structural information about the program. This information is patently program-specific so it must be supplied with each program. The second supporting service needed is a mechanism to adapt these entities (classes, methods etc) by replacing them or intercepting operations on them. This means that there must be executable code whose purpose is not to be part of the program itself but to provide a mechanism to alter the program. The mechanism must operate at the level of program instructions or blocks of instructions in order to replace or intercept them.

These requirements for runtime support mean that unanticipated dynamic adaptation is much easier to provide for programs executed by an interpreter or a virtual machine than it is for programs executed as native code. Programs compiled to native code do not usually contain sufficient symbolic and structural information about themselves, and it is very difficult to implement an adaptation mechanism for native code without inserting hooks into the program before it starts (i.e. without anticipating adaptation). On the other hand, interpreted or bytecode programs usually contain a large amount of symbolic and structural information about themselves, and an adaptation mechanism can be implemented by including it in the virtual machine. The virtual machine already operates directly on instructions at runtime so modifying or extending it can be used to support the low level manipulation mechanisms needed for dynamic adaptation. Implementing the adaptation mechanism in the virtual machine also avoids the need to insert it into the program before runtime. For these reasons architectures supporting unanticipated dynamic adaptation are usually based on languages such as Java or Smalltalk that are executed by a virtual machine rather than on languages that are compiled to native code.

1.1.5 Other issues

There are of course many considerations when carrying out dynamic adaptation of a software system, but this research is concerned principally with the architecture, model and mechanisms that provide the basic support. It is useful to consider the issues addressed and omitted by this research using the eight dimensions of dynamic adaptation proposed by Dowling [Dow00]:

- Interface (how is it triggered)
- Authorisation (who can trigger it)
- Admission (is it possible with present resources)
- Extension (where does the code come from)
- Safety (how is program safety ensured)
- Binding (how is it linked in)
- State transfer (how is state transferred from old to new)
- Dependency (what are the implications for unchanged code)

The architecture proposed by this thesis addresses three of these dimensions, interface, extension and binding, directly. The architecture effectively operates at a low level. It provides support structures and mechanisms for dynamic adaptation but does not directly address the dimensions of authorisation, admission, safety, state transfer or dependency. These are left to a higher level of software controlling the adaptation process. Some related work has been done on these issues. For example, Caromel and Vayssière have analysed the authorisation dimension in Java-based dynamic adaptation architectures and have suggested that it can be addressed using Java security permissions [Car01]. Difficulties in automatically determining admission (the timing of a dynamic change) have been identified by Gupta [Gup96] and by Hicks [Hic01]. State transfer has been examined by a number of researchers, including Gupta [Gup96], Dmitriev [Dmi01], and Hicks [Hic01], with various proposals for mapping pre-change state to post-change state.

1.2 Background

There are a number of important concepts that were part of the basis for this research. They are introduced in the following subsections.

1.2.1 Separation of Concerns

Parnas introduced the notion of separation by suggesting that criteria to measure software modularisation should include the extent to which a module supports independent comprehension and development [Par72]. Dijkstra proposed the term 'separation of concerns' to describe dealing with the obligations and constraints of a complex system one by one "... because our heads are so small that we cannot deal with them simultaneously without getting confused" [Dij76]. Separation of concerns was more precisely defined by Hursch as the separation between the basic functionality of the system and special purpose concerns such as synchronisation or location control [Hur95]. This was further refined by Aksit as a way to improve adaptability, flexibility and reusability of systems [Aks96]. In the context of dynamic adaptation separation of concerns refers to the separation at runtime between the program to be adapted and the components representing the behavioural change. Maintaining a separation between these two concerns helps to support independent development and component reuse, and simplifies application and removal of the adaptation.

Weak separation of concerns implies entanglement and interdependency, with static references between the program and the components implementing the behavioural change. Strong separation implies no entanglement and no static references between the entities. An architecture that supports a strong separation of concerns allows the program and the behavioural change to be developed, compiled, and tested independently of each other. In such an architecture dynamic association creates a link between the behavioural change and some part of the program at runtime, but up to that point they remain completely separate with no references between them.

Separation of concerns is closely related to unanticipation and to reusability. The lack of static references from the program to the behavioural change is part of the support for unanticipated adaptation. Filman [Fil00] refers to 'obliviousness' as a measure of the separation – the program should be oblivious to any behavioural change that may be made to it. Separation or obliviousness is a symmetric requirement in any potential association, because a lack of static references from the component implementing behavioural change to the program is part of the support for reusable behavioural change.

In a dynamic adaptation architecture with a strong separation of concerns association is performed completely at runtime, and the mechanism supports dynamic association of behavioural change with arbitrary parts of any program (although limited to association with the categories of operation supported by the mechanism). Of course the ability to make arbitrary associations does not guarantee that any particular association will be meaningful. That depends on the nature of the behavioural change and the operations it is associated with, and the effect of a particular association is something that can only be assessed by a system integrator or maintainer.

A strong separation supports the division of system design and maintenance into three independent roles [Tan01] (see figure 1.2):

- The role of application programmer, concerned only with the basic functionality of the program;
- The role of behavioural change programmer, concerned with the behavioural change but not necessarily with the specifics of the programs with which it may be used;
- The role of system integrator, concerned with associating specific behavioural changes with specific parts of programs.

1.2.2 Functional and non-functional concerns

One of the potential applications of dynamic adaptation is the incorporation into a program at runtime of additional behaviour that is not central to the principal functions of the program. For example, the additional behaviour might be concerned with issues such as synchronisation, persistence, security, or distribution. These are referred to as special or *non-functional* concerns, while the principal issues addressed by the program are called basic or *functional* concerns [Lis92] [Hur95] [Str96].

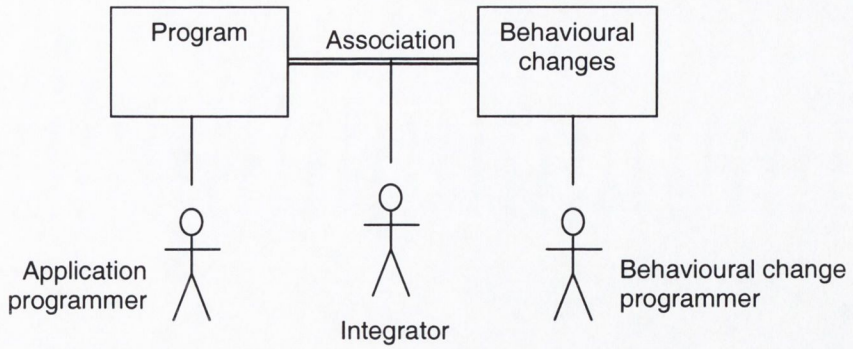


Figure 1.2: The three independent development roles.

Components addressing non-functional concerns are particularly suitable for incorporation (and removal) by dynamic adaptation because by their nature they are independent of the functional concerns of the program. Their independence of each other helps to avoid static dependencies between them, and allows the non-functional components to be developed independently and then associated with arbitrary, or almost arbitrary, parts of any program. A dynamic adaptation architecture that supports a strong separation of concerns along the association link maintains the independence of the components addressing functional and non-functional concerns. It can provide runtime support for systems that separate functional and non-functional concerns by design, and support for systems that require non-functional concerns to be manipulated independently at runtime.

This is in contrast to using dynamic adaptation to change the existing behaviour of a program at specific points by replacing or redirecting operations. Behavioural change that is intended to adjust existing functional operations is obviously concerned with functional rather than non-functional behaviour. A component implementing such functional behavioural change is likely to be less independent of the existing (functional) components than one implementing non-functional concerns. Although functional behavioural change may be less demanding of strong separation along the association link than non-functional behavioural additions, a dynamic adaptation architecture that provides strong separation is still useful to support dynamic application of functional corrections and adjustments.

1.2.3 Aspect Oriented Programming

Aspect oriented programming (AOP) [Kic97] is a technique for supporting separate treatment of issues that cross-cut the basic functions of a system. Cross-cutting issues (aspects) are those that can't be decomposed along the class inheritance lines of the primary functionality of the system. For example, applying access control to a program may require similar pieces of code to be scattered across otherwise unrelated classes [Hag02]. AOP supports independent development of such cross-cutting aspects, specification of the set of points (called *join points*) at which they are to take effect, and incorporation (called *weaving*) of the aspects at the join points. Because aspects represent cross-cutting concerns that are not part of the natural

decomposition of the system, they tend to be used to represent non-functional concerns (e.g. [Ald00], [Duc02]).

A large part of the work on AOP has focused on using it to capture static cross-cutting concerns, identified at design time and woven at compile time. Recent work on language-independent AOP supports weaving of compiled code [Laf03], but the weaving is still carried out before runtime. As pointed out by Brichau [Bri01] there is a requirement for AOP architectures that support dynamic aspects and runtime weaving (dynamic AOP) but very few existing AOP architectures provide it.

Dynamic AOP architectures (e.g. PROSE [Pop02]) and non-AOP dynamic adaptation architectures (e.g. MetaXa [Gol97], Guaraná [Oli99]) have many similarities, particularly when dynamic adaptation is used to incorporate additional behaviour that represents non-functional concerns. They both support alteration of system behaviour at runtime. Both strive for separation of concerns between the existing system and the additional behaviour [Aks96], and both strive for reusability of the behavioural change (e.g. [Han02], [Spe02]). As for dynamic adaptation architectures, the runtime support required for dynamic AOP means that it is not really practical to attempt it in a fully compiled language.

The difference between dynamic AOP and dynamic adaptation is essentially a difference in emphasis. AOP in general has as its primary goals the identification and representation of cross-cutting concerns, and the provision of techniques for weaving them into a system. Dynamic AOP attempts to provide this at runtime. It does not necessarily include the level of support for dynamic manipulation of program execution at the point of weaving that a dynamic adaptation architecture might support. On the other hand a dynamic adaptation architecture does not necessarily provide the support for identifying a set of points across a program (*join points*) and injecting behaviour in a coherent way. A dynamic AOP architecture is concerned with weaving extra functionality into a system, while a dynamic adaptation architecture is concerned with manipulation of behaviour at individual specified points. The join point specification mechanisms provided by AOP architectures allow them to operate at a higher level of abstraction than most dynamic adaptation architectures, and in fact it has been suggested by a number of researchers that the mechanisms provided by some adaptation architectures, particularly reflection (see below), could be used to implement weaving for AOP systems [Kic97] [Fil00] [Cza00] [Bak02] [Tan03].

1.2.4 Reflection

Some architectures that support dynamic adaptation are based on reflective techniques. Software reflection originally emerged from research into knowledge representation [Smi82], with its key chracteristic being support for a system to observe and modify itself. Maes proposed computational reflection to address issues in artificial intelligence such as self-optimisation and self-modification of behaviour [Mae87]. As used by Maes the term reflection referred to a process of observation, reasoning and modification. However, in recent years it has been used to describe techniques that support only a portion of the original concept. For example, the reflective features in the Java language [Sun98] support only examination of system structure, with no support for modifying classes, methods or fields in any way. Java reflection does not support monitoring or observation of execution, and it cannot be used to modify the behaviour of a class or method.

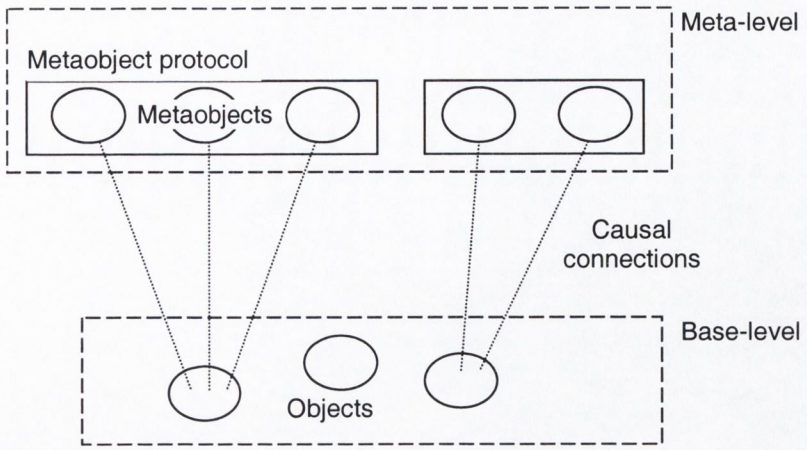


Figure 1.3: An outline of the structure of a reflective architecture.

Most modern reflective systems separate what is observed or modified from what does the observing or modifying (e.g. AL1/D [Oka92], Kava [Wel01], Javassist [Tat01], Reflex [Tan01], Iguana/C++ [Sch01], MetaXa [Gol99], Guaraná [Oli99]). The part that is observed or modified is concerned with the primary function of the overall system, and is called the base-level. The part that observes or modifies is concerned only with the structure or behaviour of the base-level, and is called the meta-level. Typically the meta-level program implements new or altered behaviour to be applied to certain operations in the base-level program. In dynamic adaptation terms the base-level is the principal program and the meta-level represents behavioural changes to be applied to the base-level program. Manipulating the meta-level program enables behavioural changes to be applied to or removed from specific parts of the base-level program. The meta-level program can also represent non-functional concerns to be incorporated into the base-level program.

Ferber considered reflection to consist of structural reflection and computational (or behavioural) reflection [Fer89]. Structural reflection supports the examination, and possibly the modification, of the structure of the base-level. Structure consists of the names and signatures of classes, interfaces, methods, and fields in the base-level. Behavioural reflection supports the observation, and possibly the modification, of the behaviour of the base-level without changing the publicly visible parts of its structure. This supports the concept of behavioural adaptation discussed in section 1.1.2.

When a base-level operation or structure is to be reflected upon it must first be *reified*. A reified operation or structure becomes concrete or manipulable, and it is usually represented by a *metaobject*, an instance of a metaobject class[1]. Behavioural change is implemented by designing metaobject classes. In use a metaobject is causally connected to a reified operation or structure, so that changes in the base-level cause a corresponding change in the appropriate metaobject, and if modification is supported changes in a metaobject cause corresponding changes in the base-level. The meta-level operates on the base-level indirectly, by manipulating the reified representation of the base-level. A reflective architecture may reify structures such as classes, interfaces, constructors, methods, and fields, and behaviour such as method execution, object

[1] Not to be confused with a *metaclass*, a term used in some languages, including Smalltalk, to refer to the class of a class.

creation and field access. Reflective architectures usually provide a means for coherent groups of metaobjects to be used together and to be manipulated as a group. The interface to a group is generally referred to as a *metaobject protocol* or MOP [Kic91]. Figure 1.3 illustrates these concepts in a general reflective structure.

The principal benefit that reflective mechanisms bring to dynamic adaptation is the separation that they support between the program (the base-level) and the components implementing behavioural change (the meta-level). The mechanisms allow the meta-level to reach into the base-level to examine or modify it, while maintaining a sharp distinction between the two levels. Systems that support this separation between the base-level and the meta-level components at runtime inevitably pay a performance cost because of the need to transfer control back and forth between the levels as reified operations are executed [Gol99]. The association, or binding, at runtime between the base-level and meta-level components increases flexibility but decreases performance [Str93]. Many reflective systems attempt to address the issue by limiting reification to the operations for which it is actively required (e.g. Kava [Wel01], MetaclassTalk [Bou99]) and by making the reification mechanism as efficient as possible (e.g. MetaXa [Kle96]).

Despite the performance cost reflective techniques can have significant advantages for implementing a dynamic adaptation architecture. Reflection can support non-invasive interception of operations in the base-level, redirection of those operations to the meta-level, flexible processing of redirected operations, and transparent manipulation of arguments and return values. Reflection is a very powerful mechanism. It has been pointed out, almost since reflection was first researched, that reflection needs discipline and constraints in order to become useful in a general programming environment [Mae87] [Men97]. Some of the issues related to what these constraints should be have been addressed by Forman and Danforth [For98]. They proposed a model (SOM) that imposes restrictions to ensure that substitutability (the basis for polymorphism) is not damaged under reflective manipulations.

1.3 Challenge Problems

A number of challenge problems for incorporating components defining behavioural change into a program have been identified and described by Tarr et al. [Tar00]. The problems include issues such as reusability, context sensitivity, and the inheritance anomaly. These challenge problems were part of the motivation for this research, and the architecture proposed by this thesis addresses them more successfully and with a simpler programmer's model than any existing dynamic adaptation architecture. The problems are also used as the basis of the evaluation of the proposed architecture.

The following subsections outline some of the most important of these challenges.

1.3.1 Reusability

Just as for components of any program it is desirable that the components implementing behavioural change be reusable. Reusability of components has a range of meanings, including designing components to make them reusable at a source code level in other programs [Joh88], the use of inheritance mechanisms to reuse

existing classes [Bie92] [Mch94], and reuse of compiled components in different programs or contexts (called here *binary reusability*). There are many measures of the reusability of a software entity [Cal91] [Bie92] [Pri97]. In referring to behavioural change in this thesis the term reusability is used to mean the ability to associate the component implementing behavioural change with arbitrary classes of a program. If it is possible to associate a component implementing behavioural change with different classes without editing and recompiling the behavioural change components then they can be said to have binary reusability. To achieve this the components must contain only an implementation of the behavioural change, without any static reference to the program class being adapted and without any static details of the association.

Reusability is an issue at both ends of a potential association. In addition to the reusability of the components representing the behavioural change the reusability of the program with respect to the new behaviour must also be considered. This is the extent to which a program can be reused (without recompiling or reloading it) with arbitrary behavioural change (or without any behavioural change). For maximum reusability of the program in the dynamic adaptation context the program should have no dependencies on any behavioural change that might be bound to it. Czarnecki refers to this as 'non-invasive adaptability' of the program [Cza00]. For example, an arrangement where one part of the program might invoke a method to associate a specified program object with a statically specified behavioural change would break the non-invasive adaptability of the program.

The reusability issue is illustrated in [Tar00] by a challenge problem in which it is required to add synchronisation to two separate classes, one representing a stack and the other a first-in-first-out (fifo) buffer. The synchronisation behaviour should be implemented as a single component that may be associated successfully with either class. Both classes have methods for putting and getting data, but there is no naming consistency between the methods. The principal difficulty arises in identifying to which methods of the class synchronisation operations should be applied when the method names are not known statically to the synchronisation component. An architecture that overcomes this challenge supports binary reusability of components implementing behavioural change.

1.3.2 Context sensitivity

Context sensitivity[2] means support for examination of the arguments, return values and execution context of operations so that an associated behavioural change component may make decisions based on this information. Examination of arguments and return values is supported by most dynamic adaptation architectures but examination of execution context is not (e.g. MetaXa [Gol99], Guaraná [Oli99], and PROSE [Pop01]). The issue is illustrated in [Tar00] by a challenge problem derived from one originally proposed in [Bri00] and referred to as the *jumping aspects* problem. It occurs when a behavioural change is to be applied to an operation depending on where the operation is invoked from.

The example used to illustrate the problem is based on an existing class with two public methods. The first method performs some simple operation on an instance of the class, and the second method performs a group of such operations by calling the first method repeatedly. It is required to dynamically apply a

[2] Not to be confused with *context-aware*, which is generally used with a wider more abstract meaning.

behavioural change that will generate a notification event whenever one or more of the operations are performed. The difficulty is that the simplest solution, applying event generation to invocations of the first method, will result in multiple events being generated when a user invokes the second method. A way to avoid this is to apply the event generation change to both methods, but to arrange that it will only become active in the first method when the invocation is not from the second method.

This approach requires that the dynamic adaptation architecture supports context examination. It must allow the behavioural change component to determine whether or not an intercepted invocation of a method came from some other specified method. A more general requirement is that the dynamic adaptation architecture must support examination of the current execution context to identify methods and classes in the call chain. This will allow the behavioural change to be context sensitive.

1.3.3 Inheritance anomaly

The inheritance anomaly was originally identified by Matsuoka as a problem encountered when implementing synchronisation in concurrent object-oriented systems [Mat91] [Mat93]. The problem is presented by Matsuoka as essentially one of conflict between the inheritance mechanism and the concurrent synchronisation requirements. It has been analysed in detail by a number of researchers (e.g. [Tho94], [Fer95], [Hol98]). McHale [Mch94] suggests that the inheritance anomaly is a wider issue than just synchronisation in concurrent languages, and that it is due to a fundamental conflict between inheritance and encapsulation. Under certain circumstances (the original anomaly of Matsuoka is one) the correct behaviour of a subclass requires that inherited code be restated, possibly with additions, in the subclass. This breaks the encapsulation of the superclass, because blocks of code from the superclass are repeated in the subclass with consequent potential for errors or omissions, particularly if the superclass is modified at a later time.

A version of the inheritance anomaly has been identified as an issue in systems that support the application of additional behaviour to a component while attempting to maintain a sharp separation between the implementation of the addition and the original component [Tar00]. The issue is illustrated by an extension of the example used to illustrate the reusability issue (see section 1.3.1). In the reusability example synchronisation constraints are implemented as a separate component applied as a behavioural change to a basic stack or buffer class. To illustrate the inheritance anomaly the example is extended by creating a subclass of the stack (or buffer) class in which further methods are added. The anomaly occurs in trying to ensure that synchronisation constraints are correctly applied to the subclass, including the added methods, without restating all the synchronisation constraints of the superclass.

If a dynamic adaptation architecture is used to associate the additional behaviour with methods of a class then the inheritance anomaly can be overcome if the architecture ensures that the association is inherited by subclasses [Tar00]. If this feature is supported by the architecture then the behavioural change will be associated automatically with overridden methods in subclasses. Tarr also proposes that it should be possible to adapt such inherited behavioural change for subclasses without affecting the association for the superclass, in the same way that methods may be adapted in a subclass without affecting methods in the superclass. Although these features were identified in [Tar00] as desireable there is still no existing dynamic adaptation architecture that supports them all.

1.4 Contribution

There are a number of existing architectures that support unanticipated dynamic adaptation to varying degrees. These include MetaXa, Guaraná, and PROSE. However they do not provide a coherent structure for applying behavioural changes. In particular they do not address any of the issues surrounding the scope, inheritance, and refinement of behavioural change.

- Indiscriminate scope of change: Dynamic adaptation using existing architectures can result in the inheritance relationship between two classes and the instance-of relationship between a class and an object appearing to be broken. Existing architectures allow indiscriminate association of unrelated behavioural changes with related classes and objects. This can lead to a class omitting behaviour that is associated with its superclass, and to objects having behaviour that is unrelated to the behaviour associated with their class.
- No inheritance of change: There is no support for inheriting change, and therefore propagation of changes throughout part of a class hierarchy must be done explicitly. It is difficult to apply behavioural changes reliably to a set of related classes or to all instances of a class.
- No refinement of change: Lack of support for refinement means that it is not easy to extend components implementing behavioural change. This makes it difficult to cope with wider adaptation requirements in subclasses of the base-level program.

The major contribution of this thesis is to address these problems by bringing structure and constraint to unanticipated dynamic adaptation. The architecture proposed by this thesis supports unanticipated dynamic adaptation, provides a type framework for constraining dynamically applied behavioural change, supports inheritance and refinement of behavioural change, and solves a number of acknowledged challenge problems not addressed by existing architectures for unanticipated dynamic adaptation. The proposed architecture uses the concept of *metatype* to impose structure and predictable scope on the dynamic association of behavioural changes with a program. Metatypes extend the conventional notion of a type to include behavioural adaptation, and they ensure that class behaviour is always inherited by subclasses and propagated to instances even in the presence of dynamically applied behavioural changes. Metatypes support derivation of behavioural change components in a structure similar to the derivation of conventional types. Metatypes were introduced by Schäfer as a level of abstraction to encapsulate the functionality of a reflective language extension [Sch01]; in this thesis the concept is extended to provide a consistent support structure for a dynamic adaptation architecture at runtime. Constraints on reflection were introduced by Danforth and Forman [Dan94]. The constraints proposed by this thesis are similar in some ways, but are applied to behavioural rather than structural adaptation, and support unanticipated rather than anticipated adaptation. The metatype concept and the constraints proposed here emphasise the separation between the components implementing behavioural changes and the components to which the changes are to be applied.

The dynamic adaptation architecture has been implemented for the Java language as *Iguana/J*. It uses reflective techniques to apply behavioural changes, and adds behavioural reflection to Java in order to do this. The implementation provides a consistent and simple programmer's model, with a smaller class library than other architectures supporting unanticipated dynamic adaptation. The implementation also addresses

other acknowledged issues for dynamic adaptation by supporting context examination and providing a stronger separation of concerns between the program and the behavioural change.

1.5 Roadmap

The rest of this thesis describes related work, the Iguana/J architecture and its implementation, and the evaluation of the architecture. Chapter 2 is an overview of other adaptation architectures, including a detailed review of the relevant features of three other architectures that support unanticipated dynamic adaptation. Chapter 3 describes the new architecture by introducing the Iguana/J programmer's model and the rationale for some of the critical design choices made for it. Chapter 4 outlines the implementation of Iguana/J. In chapter 5 Iguana/J is evaluated by comparing it to the three existing architectures described in chapter 2. The comparisons are carried out on the basis of acknowledged challenge problems for this field. The conclusion in chapter 6 summarises the thesis and indicates open issues and possible future research directions.

1.6 Summary

This chapter introduced the field of unanticipated dynamic adaptation, described some of the key choices and issues in using it, and set out the scope of this research. It provided the motivation for the research by describing important issues and challenge problems for dynamic adaptation. It discussed underlying concepts such as separation of concerns and non-functional concerns, and introduced related areas such as aspect oriented programming and reflection.

The chapter also outlined the contribution of this research to the body of knowledge. The research proposes an architecture for unanticipated dynamic adaptation that solves previously unsolved issues in the field, and that provides a more coherent and structured programmer's model with a smaller and simpler class library than any existing architecture.

Chapter 2

Dynamic Adaptation Architectures

It's just three sticks to them. But to the educated it's a great and glorious A.

- Eeyore.

A. A. Milne, The House at Pooh Corner, 1928.

This chapter contains reviews of a number of architectures that support adaptation of object-oriented software systems. The review is intended to be representative rather than exhaustive. The architectures listed are divided into three broad categories, according to whether they support static, anticipated dynamic or unanticipated dynamic adaptation. The chapter includes very brief descriptions of architectures in the first two categories. The reviews of the three architectures in the third category are more detailed, because they are closest to the architecture proposed by this thesis.

2.1 Categorisation

There is a range of possible approaches to software adaptation, from completely static to completely dynamic [Tar00], and that have varying trade-offs between functionality and cost. The range includes approaches that use source code preprocessors, aspect oriented programming and reflection in the implementation and application of behavioural change. For this review some of the architectures are initially categorised based on the concepts of *binding time* and *binding mode* defined by Czarnecki [Cza00] (see tables 2.1, 2.2, and 2.3). Binding time describes when the weaving or association of the program and the behavioural change occurs, and may be compile time, load time or runtime. Binding mode describes the permanency of the binding, and may be static or dynamic. Static binding mode means that the binding cannot be undone, whereas dynamic mode means that it may be undone and redone[3].

[3] The use of the terms *static* and *dynamic* to describe binding mode is different from their use in the context of adaptation in general, where static is taken to mean that the adaptation occurs before runtime and dynamic to mean that it occurs during runtime.

Name	Binding time	Binding mode	Anticipation
Hyper/J	Compile time	Static	Source code
Knit	Compile time	Static	Source code
Open C++	Compile time	Static	Source code
OpenJava	Compile time	Static	Source code
AspectJ	Compile time	Static	Compile time
AL-1/D	Compile time	Static	Compile time
Kava	Load time	Static	Load time
Javassist	Load time	Static	Load time
Aspectual Components	Runtime	Static	Load time
SOM	Runtime	Static	Source code

Table 2.1: Software adaptation architectures that support static mode adaptation

Name	Binding time	Binding mode	Anticipation
ABCL/R	Runtime	Dynamic	Source code
CodA	Runtime	Dynamic	Source code
Reflex	Runtime	Dynamic	Source code
LEAD++	Runtime	Dynamic	Source code
Caesar	Runtime	Dynamic	Source code
Iguana/C++	Runtime	Dynamic	Source code
MetaclassTalk	Runtime	Dynamic	Compile time
Java Aspect Components	Runtime	Dynamic	Load time

Table 2.2: Software adaptation architectures that support anticipated dynamic mode adaptation

Name	Binding time	Binding mode	Anticipation
MetaXa	Runtime	Dynamic	none
Guaraná	Runtime	Dynamic	none
PROSE	Runtime	Dynamic	none
Iguana/J	Runtime	Dynamic	none

Table 2.3: Software adaptation architectures that support unanticipated dynamic mode adaptation

For an architecture to support dynamic adaptation fully it must allow behavioural change to be applied to and removed from a program at runtime. An architecture that allows change to be applied but not removed would be restricted in the kinds of adaptation that it could be used for. Specifically, it would be difficult to use it for transient changes such as those required to adapt mobile systems to temporary variations in context. Architectures that support dynamic adaptation fully must carry out binding at runtime and must use dynamic binding mode. Such architectures can be further classified by whether they require anticipation of the binding in source code, at compile time, at load time, or not at all. Anticipation in source code implies that static references to the binding must be included. This might mean including calls to methods that carry out the binding, using replacement classes instead of original classes, or using special method calls instead of the `new` operator to create objects. Anticipation at compile time means the use of special preprocessors or compilation techniques that perform specific transformations to insert support for dynamic adaptation. Anticipation at load time implies that classes must undergo specific transformations as they are loaded, and that details of the transformations require knowledge of the points at which adaptation is to be performed at runtime. Architectures that do not require any anticipation support *unanticipated* dynamic adaptation.

In practice architectures that support binding only in source code, at compile time or at load time will have static binding mode (i.e. it cannot be undone) because compilation and loading are transient phases in the system's life cycle and therefore it is of little practical benefit to do and undo a binding during compilation or loading. On the other hand, most architectures that support binding at runtime will use dynamic binding mode (i.e. it can be undone) because most of the techniques used to support binding at runtime are reversible and dynamic binding mode provides greater flexibility in the manipulation of behavioural changes at runtime. An architecture that supports static binding mode at runtime can only be used to apply permanent changes, and therefore would be restricted to dynamic application of maintenance upgrades or other persistent evolutionary modifications.

Some reflective architectures that only support association of behavioural change components before runtime can still be used for making behavioural changes dynamically if a suitable behavioural change component is associated (before runtime). For example, a behavioural change component that redirects method invocation and allows the redirection target to be changed dynamically supports dynamic adaptation even though the behavioural change component is associated statically. This approach to dynamic adaptation is not explored further in this thesis because it cannot be used to support unanticipated adaptation. It requires anticipation of both the type of behavioural change component required and with which components of the program it should be associated.

Tables 2.1, 2.2, and 2.3 illustrate the range of possible approaches to adaptation of object-oriented software systems. They list binding time, binding mode and anticipation requirements for some existing AOP and reflective architectures. The completely static end of the range of approaches supports static binding at compile time, while the completely dynamic end supports unanticipated dynamic binding at runtime. It is acknowledged [Bri01] that there has been a lack of support at the dynamic end of the range. Tables 2.1, 2.2 and 2.3 divide the architectures into three broad categories: those supporting static adaptation, anticipated dynamic adaptation, and unanticipated dynamic adaptation. Those in the first two categories are described briefly in the following two sections, and those in the third category are discussed in more detail in the subsequent sections of this chapter. Iguana/J, an implementation of the dynamic adaptation architecture proposed by this thesis, is included in table 2.3 for reference, but is not discussed until chapter 3.

2.2 Static Adaptation

Static adaptation is not central to the subject of this research, but this short discussion of architectures supporting it is included to provide some perspective on dynamic adaptation. These architectures are categorised as supporting adaptation on the basis that they support modification of a program by applying changes defined by separate components. They are listed in table 2.1.

At the static end of the range of possible approaches to adaptation, AOP architectures such as AspectJ [Kic01], Hyper/J [Tar00a] and Knit [Eid01] provide support for encapsulation of cross-cutting concerns during system implementation, but they provide very limited support, if any, for dynamic adaptation. These three architectures focus on helping the programmer to modularize design concerns or system features that would otherwise cut across the conventional class structure of the program. The cross-cutting concerns or

features are statically woven or associated with the appropriate parts of the program at compile time. The aspects can be designed as reusable components, in a similar way to designing conventional classes as reusable components. There is little or no support for manipulation of aspects while the system is running. Reflective architectures such as Open C++ [Chi95] and OpenJava [Tat00] also perform static adaptation at compile time, without any explicit support for dynamic adaptation.

AL-1/D [Oka92] is a reflective architecture. It is intended for use in distributed concurrent systems to support a strong separation during development between the principal objectives of a component and secondary issues such as object location and migration. While it is not presented as an aspect oriented architecture its objectives are very similar to AOP. The base-level and meta-level components are bound at compile time, although there is some support for dynamic manipulation of the binding at runtime.

Two of the architectures in Table 2.1, Kava [Wel01] and Javassist [Tat01], (both based on Java) use load time modification of code to perform static adaptation. The benefit of load time rather than compile time modification is that program source code is not required, and therefore this approach may be used to apply static adaptation to third party and legacy programs.

Kava is a reflective architecture that performs static adaptation of programs by binding metaobjects to specified classes. The classes are modified at load time to create the binding. The classes and objects remain separate from the metaobjects at runtime, but there is no support for unbinding or rebinding them while the system is running. The basic difficulty is that interception hooks are inserted at load time only for classes and operations for which they are known, statically, to be required. Therefore Kava could not support dynamic rebinding except in cases where the new metaobject reified a subset of the operations reified by the metaobject bound at load time. One of the most interesting features of Kava is its use of a separate configuration file to declare static associations between classes and metaobject classes. This removes the need for the program or the metaobjects to contain embedded references to each other, and supports a strong separation of concerns between the program and the components implementing behavioural change.

Javassist is a reflective system for translating Java bytecode at load time. It is intended to be used to apply static adaptation to a program. It does not have explicit support for dynamic manipulation of behaviour after load time. Its reflective components are used only to translate the classes as they are loaded, and do not play any part in the execution of the program. However, the translation can be used to modify the program so that it binds to reflective components that are not part of Javassist itself. At least one reflective architecture (Java Aspect Components, see section 2.3) uses Javassist at load time to insert support for dynamic adaptation into a program.

Aspectual Components [Lie99] supports reusable aspects for Java by separating the weaving declarations from the aspects. It uses Binary Component Adaptation (BCA) [Kel98] to modify the bytecode of specified classes and methods before load time. The modification inserts interception mechanisms and adds support for runtime binding. Binding is initiated at runtime by instantiating connector classes that contain references to the aspects and to the classes to which they are to be applied. There is no support for unbinding or rebinding at runtime. Aspectual Components is slightly different from others in that while it performs binding at runtime, the modifications it makes to the classes of a program at load time are aspect specific.

SOM [For98] is a language-independent reflective programming model. Every class is an instance of a metaclass, and methods of the metaclass can be used to operate on the class at runtime. Classes (and metaclasses) can be created dynamically by creating a 'nascent' instance of an appropriate metaclass, then using metaclass methods to set the new class' s parent(s), and finally using metaclass methods to add or override methods of the new class. Metaclasses can be used to implement structural changes for a class by adding, removing or replacing methods or fields, and to implement behavioural changes for a class by overriding or redispatching methods. However, once a class has been created it is not possible to change its metaclass, so SOM cannot support removal or association of a behavioural change with a class after the class has been created. The technique that SOM supports for associating a metaclass and a class dynamically requires that the class be constructed from scratch at runtime; there is no support for dynamically associating a metaclass with an already compiled class. SOM is most notable for its introduction of metaclass constraints, which require that the metaclass of any class X be derived from the metaclasses of the parents of X. SOM also supports automatic dynamic derivation of a new metaclass to satisfy the constraints for a given class.

2.3 Anticipated Dynamic Adaptation

In the middle of the range of approaches to software adaptation there are a number of architectures that support dynamic adaptation but require preparation of the program in source code or at load time. These architectures are categorised here as supporting *anticipated* dynamic adaptation (see table 2.2). Five of the dynamic adaptation architectures listed require the adaptation to be anticipated in source code, one requires it to be anticipated at compile time, and one requires it at load time. This section is a brief discussion of each of them.

ABCL/R [Wat88] is an interpreted reflective language, intended to support concurrent programming. Every object in ABCL/R has a state and a set of scripts (methods). Every object also has an associated metaobject that contains as its data the structure (including scripts) of the base-level object. The metaobject supports modification of its data in response to messages received by it. As this data represents the structure of the base-level object, this allows the modification and addition of scripts in the base-level object, and thus the architecture supports dynamic changes to the object's behaviour. Although base-level object and metaobject are separate, they do not represent functional and non-functional behaviour. The metaobject represents all of the structure and behaviour of the base-level object, not just the non-functional behaviour. Therefore there is no concept of binding a component representing behavioural change to an existing base-level object, and as a result the categorisation of ABCL/R here is slightly awkward. The behaviour of a base-level object is changed by sending messages to the existing metaobject, not by binding a new metaobject to it. It is difficult to see how dynamic adaptation could be carried out without including explicit supporting code in the program itself, so this architecture has been categorised as requiring anticipation in the source code.

CodA [Mca95] is an interpreted reflective language described as "implemented in Smalltalk". Every CodA object has an associated set of meta-level components that define the object's behaviour. Every message received by the object is redirected to the object's meta-level, so replacing the meta-level

components modifies the object's behaviour. CodA does not use the meta-level to implement all the behaviour of the object; only the modified behaviour is implemented there. This supports a separation of concerns at runtime between the original program behaviour and the behavioural change. While CodA does support dynamic adaptation, the adaptation can only be applied to individual objects. It cannot be applied to all instances of a named class and therefore it cannot be used without anticipating it by design in the program source code where object references are available.

Reflex [Tan01] is a reflective architecture that focuses on providing an 'open' reflective extension of Java that can be specialised for different needs. It creates modified versions of program classes to introduce interception hooks, and the modified classes must be used in place of the original program classes. Instances of modified classes may be associated with metaobjects, and intercepted operations will be redirected to the metaobjects for handling. Specified program classes may be modified in source code or in bytecode before runtime, or in bytecode during runtime. If a class is not modified before runtime then the modifications can be carried out dynamically when a reflective instance of the class is required. To do this the programmer must have used a special Reflex method instead of the `new` operator to create the object. A specialised subclass is created dynamically to include the hooks required by the metaobjects being bound to it, and an instance of the subclass is returned. Although Reflex supports binding at runtime it must be anticipated. The programmer must replace `new` with special calls, or must perform code transformations before runtime.

LEAD++ [Ama99] is a reflective language that implements a dynamic adaptation model by extending the Java language. It supports dynamic adaptation of the structure of a program in response to specified changes in the state of the runtime environment. The architecture includes evaluation of state and selection of strategy. It is intended for systems that are required to adapt in specified ways by selecting among available strategies. The adaptation is part of the system design, and the LEAD++ architecture does not draw a distinction between original behaviour and behavioural change, or between functional and non-functional behaviour. Although it supports dynamic adaptation it requires that the adaptation be anticipated in the design of the program.

Caesar [Mez03] is a dynamic AOP model intended to support a strong separation between the implementation of an aspect and the binding of the aspect to a target class. Caesar uses three entities to do this: aspects, *Aspect Collaboration Interfaces* (ACIs), and *bindings*. An ACI is a declaration of provided and expected interfaces for the aspect. A binding is essentially a mapping of expected interfaces declared by the ACI onto target classes. Aspects and bindings are composed into *weavelets*, which are deployed in target classes. Caesar supports dynamic deployment, in which the keyword `deploy` is used to declare that a code block should have a weavelet applied to it, but the weavelet is not specified until runtime. This allows the weavelet (and therefore the aspect) to be dynamically selected, although the block with which it is to be deployed must be statically marked with the `deploy` keyword. Different threads of execution of the same code block can select different weavelets. Thus, Caesar supports application and removal of aspects at runtime, but the locations at which they are to be deployed must be anticipated by use of the `deploy` keyword.

Iguana/C++ [Sch01] is unique in that it is the only fully compiled reflective language supporting dynamic binding. All the other architectures supporting dynamic binding mode in tables 2.2 and 2.3 are interpreted by virtual machines, and therefore have the benefit of a substantial runtime platform. In Iguana/C++ a

preprocessor inserts the required runtime support into the program at appropriate places. Extensions to the C++ language support dynamic association of base-level classes and objects with metaobject protocols. This allows a programmer to modify the behaviour of objects dynamically, but the modifications must be anticipated in the design of the program. Only objects whose classes have been explicitly prepared by source code statements in the program may be modified at runtime by binding and rebinding them to metaobject protocols.

MetaclassTalk [Bou99] is a reflective architecture based on Smalltalk. It maintains runtime separation between the base-level program and the meta-level components by implementing the meta-level as metaclasses. Instances of the metaclasses are used as the classes of the program objects, and the behaviour of the program may be adapted at runtime by dynamically changing the class of an object. Metaclasses may be written to be independent of specific program object types, with configuration properties used to parameterise them for particular base-level classes. The architecture uses program transformation techniques at compile time to insert the hooks to support the runtime changes. However, binding between program objects and metaclass instances (representing behavioural changes) takes place at runtime. Binding may be done and redone so the binding mode is dynamic. To minimise runtime overheads only selected objects are modifed at load time, and therefore only objects for which adaptation has been anticipated can be dynamically adapted.

Java Aspect Components (JAC) [Paw99] is an AOP architecture that uses load time translation of binary code to introduce a weaving mechanism into selected classes. The load time translation is done using Javassist (described above), and the weaving mechanism is used at runtime to support limited dynamic adaptation. The benefit of this approach over compile time processing or source code modification is that source code is not required. By using a load time translation technique JAC allows the program classes and compiled aspect components to remain separate and reusable in their compiled forms. At load time the weaving mechanism is inserted only into selected classes of the program, and the architecture supports a range of dynamic operations to weave and un-weave aspects with those classes at runtime. If a class has not been modified at load time then it is not possible to weave aspects with it, and therefore dynamic adaptation must be anticipated at load time.

2.4 Unanticipated Dynamic Adaptation

Architectures at the dynamic end of the range provide a more sophisticated dynamic adaptation model, usually by tightly integrating a runtime support engine into a virtual machine. Three such architectures are MetaXa [Gol99], Guaraná [Oli99] and PROSE [Pop01], listed in table 2.3. All three are based on the Java language. MetaXa and Guaraná use specialised virtual machines, while PROSE uses the debugging features of the standard virtual machine to hook into operations. They all support binding at runtime of components representing behavioural change to specified classes of a program. MetaXa and Guaraná are reflective architectures, and PROSE is an AOP architecture.

These three architectures allow unbinding and rebinding of behavioural change at runtime, so they support dynamic binding mode. None of them require anticipation of adaptation before runtime. The

specialised virtual machines used by MetaXa and Guaraná allow them to intercept program operations without modifying source code or bytecode before runtime, and the debug mechanism used by PROSE supports dynamic insertion and removal of breakpoints without preparation of the program. All three architectures effectively suspend execution of the program when it reaches a point at which adaptation is required. Control passes to the component implementing the behavioural change, which runs in the same thread as the interrupted program. When the behavioural component completes execution the program continues.

The three architectures are described in the following three sections of this chapter. These are the closest to the architecture proposed by this thesis, Iguana/J, and therefore they are described in more detail than other architectures. The six headings under which the three architectures are described are:

- *Programmer's model* describes the architecture's general model of adaptation, including its class library, the kinds of operations that may be adapted, and how it addresses issues such as context sensitivity, level transfer, and examination and alteration of arguments and return values.
- *Composition* describes the architecture's support for associating more than one behavioural change with a single base-level class or object. In reflective architectures this refers to composition of separate metaobjects. In AOP architectures it refers to composition of separate aspects. Some researchers have used the term composition to refer to the general process of integrating behavioural change components and base-level components (e.g. [Hur95], [Aks96]). This thesis distinguishes between composition and association. In reflection terms composition occurs at the meta-level, between behavioural change components, while association occurs between the base-level and the meta-level.
- *Association* describes how association of components representing behavioural change with program components is initiated, whether association is supported for both classes and individual objects, and the scope of those associations.
- *Inheritance and refinement of behavioural change* describes the architecture's effect on the type relationships between related classes and objects in the presence of behavioural change. The three architectures reviewed here all lack support for maintaining consistency and structure between related classes and objects. This is a key issue for this research, and is addressed by the *metatype* concept in the architecture proposed in the following chapters of this thesis.
- *Anticipation, reusability, and separation of concerns* describes the architecture's support for maintaining a strong separation between the components representing behavioural change and the base-level classes and objects, as explained in section 1.2.1.
- *Implementation* gives an overview of the implementation techniques, and describes some of the performance implications of the implementation.

2.5 MetaXa

MetaXa [Gol97] [Gol99], originally called MetaJava, was the first Java-based architecture to support unanticipated dynamic adaptation. It was originally proposed in 1997 following research in the University of Erlangen by Michael Golm. At the time that MetaXa was designed the Java language did not include any

reflective capabilities, so MetaXa provided both behavioural and structural reflection. Its objectives are to support a strong separation between functional and non-functional concerns, and to provide an efficient general reflective model where reflective features can be used on arbitrary objects to examine and modify their structure. The architecture tries to minimise the performance cost of reflection on objects that do not require the reflective features.

2.5.1 Programmer's model

The reflective features of MetaXa allow individual metaobjects to be bound to arbitrary classes, objects and object references in a program. Only classes and objects whose behaviour is to be modified have metaobjects bound to them. There is no explicit support for metaobject protocols. In a typical MetaXa system a meta-level program, developed to work with a specific base-level program, will create and associate metaobjects with particular base-level classes and objects in order to modify their behaviour.

Meta-level programs are written and compiled as standard Java classes, using standard Java development tools. It is not necessary to include any meta-level references in base-level programs at compile time, or to prepare them before runtime. That is, base-level programs may be implemented independently of meta-level programs.

Meta-level programming is supported by a comprehensive, and arguably complex, class library. The principal class in the library is `MetaObject`, which is intended to be used as the superclass for metaobject classes implementing behavioural change. An instance of the metaobject class is bound to a base-level class or object, and is registered for appropriate base-level events in that class or object. When any of those events occur a synchronous notification is sent to the metaobject, and event handling methods can adapt the behaviour as required before the base-level program proceeds.

Class `MetaObject` provides a complex set of over 170 methods for binding, event handling and JVM manipulation. The event handling methods support registration for events such as method invocation, object creation and field access, and provide default notification methods for each event type. A typical meta-level class would extend class `MetaObject`, implementing behavioural change by overriding an appropriate set of event notification methods. The default notification methods in `MetaObject` do nothing. The metaobject initialisation can include code to attach itself to the target base-level object using one of the binding methods, and then register for the required base-level events using one of the registration methods. The act of registering for an event automatically reifies that event in the base-level object. When a reified event occurs processing of the event at the base-level is suspended and the event is sent to the event dispatcher in the metaobject. The dispatcher chooses an appropriate notification method in the attached metaobject, based on the type of the event, and the notification method decides how the reified operation is to be handled (see figure 2.1).

```
public class TraceMeta extends MetaObject {
   public TraceMeta(Object obj) {
      attachObject(obj);
      registerEventMethodCall(obj);
   }
   public void eventMethodEnterVoid(Object o,EventDescMethodCall e){
      System.out.println("Entering " + e.methodname);
      continueExecutionVoid(o, e);
   }
}
```

Figure 2.1: A MetaXa metaobject to trace calls to all methods of a given object.

The event notification methods support the insertion of additional behaviour before and after the reified operation, and they support cancellation or replacement of the operation. There are 10 notification methods for method invocation, differing only in the type of the return value (eight primitive types, `Object`, and `void`). Arguments to a reified operation are marshalled into an array of `Object`, with primitive types being wrapped automatically in instances of appropriate classes.

The MetaXa library includes classes reifying structural elements such as class, field, method, and opcode. These provide detailed access to the structure of base-level classes, supporting examination and modification. MetaXa does not use the standard reflection features of Java because it was designed before those features were introduced. The library also includes classes representing six behavioural reification categories. They are class loading, field access, method call, object creation, object destruction and object locking

Each notification method receives a reference to the target base-level object and an event descriptor object. The meta-level program may examine the event descriptor object to determine the source of the event. For method invocation events the descriptor includes the method name, signature and arguments. For field read events it includes the field name and signature, and for field write event it also includes the new value to be written.

Most of the methods provided by the class `MetaObject` are utility methods used to manipulate reified structure and operations. They form the Meta-level Interface, or MLI. The MLI allows a meta-level program to carry out low level operations including examination and manipulation of the runtime symbol table for a class (the Java 'constant pool') and examination and manipulation of method bytecode. However, MetaXa does not support examination of the Java execution context when a reified operation occurs so a metaobject cannot be made context sensitive.

2.5.2 Composition

Composition of metaobjects in MetaXa is dynamic and uses a simple linear chain structure. The composition occurs automatically and incrementally at runtime as the metaobjects are registered one by one for events in the base-level object. When an event occurs it is notified to each metaobject in turn, starting with the most recently registered for that event.

This composition model suffers from a number of problems. The main one is that it does not deal with compatibility issues between composed metaobjects, so it is left to the meta-level programmer to ensure that a particular composition will produce a meaningful result. In addition, the orderly passing of the event along the chain depends on the inclusion of appropriate statements in the handler methods of each metaobject. These issues arise in any architecture that uses a simple composition chain. It is a more serious problem in architectures such as MetaXa that use implicit composition because the meta-level programmer does not necessarily know what metaobjects are already associated with a particular base-level object.

2.5.3 Association

MetaXa supports association of metaobjects with classes, individual objects and individual object references. Where associations have overlapping effects they must be applied in order from the most general to the most specific (i.e. class, then object, then reference) as otherwise the overall effect may not be a valid composition of the metaobjects concerned [Gol97]. To ensure that this association order is preserved the designers of the architecture recommend that association with a class be only carried out immediately after it is loaded, and that association with an object or a reference be only carried out immediately after they are created. This could be a serious limitation if, for example, it is required to apply behavioural change to a class that has been loaded for some time.

Association is initiated by creating an instance of the required metaobject class, and then invoking one of three methods inherited by the metaobject. The methods take a reference to a base-level class or object and create the binding between the metaobject and the specified class, object or reference. This must be followed by invoking appropriate inherited methods of the metaobject to register it for the event types that it reifies. In a typical metaobject the association and the registration are carried out in the metaobject's constructor using base-level class or object references passed as constructor arguments (as in the listing in figure 2.1), and therefore creation and association of a metaobject can be carried out in a single step (see figure 2.2). Associating a metaobject with an individual base-level object affects only that object. Associating a metaobject with a single object reference affects only accesses to the object via that reference. Explicit calls from the object to superclass methods (using `super`) are also subject to any metaobjects associated with the object or reference, although instances of those superclasses are not affected.

Associating a metaobject with a base-level class affects all existing and future instances of the class, and operations on both declared and inherited members of the class can be reified. It is not clear from the published material how the architecture deals with the difficulty of reifying invocation of an inherited method without also affecting invocation of that method via an instance of the superclass. Reifying any operation in a class does not cause it to be reified in a subclass, regardless of whether the subclass overrides the operation or not. In other words association of a metaobject is not inherited at the base-level.

MetaXa supports reification of operations on a subset of members of a class, allowing adaptation to be applied more precisely. For example, it is possible to reify invocation of just a few named methods rather than all methods of a class. The method names may be given as part of the association operation rather than embedded in the metaobjects, allowing the metaobjects to be reusable with different base-level classes.

While MetaXa supports reification of invocation of both public and private methods, there is an important limitation on its reification of field access. Because the interception mechanism relies on modification of bytecode, reification of access to public fields would require the identification and modification of every relevant field access instruction in any class of the program. The MetaXa designers decided that this was not practical, so only access to private and protected fields may be reified.

```
//...
Object obj = new Object();
Metaobject mo = new TraceMeta(obj);
//...
```

Figure 2.2: A fragment of code to create and associate a MetaXa metaobject.

2.5.4 Inheritance and refinement of behavioural change

MetaXa ensures that behavioural change applied to a class will be applied to all instances of the class even when some instances also have other behavioural changes applied to them as individuals. This means that it is not possible for an object to omit or avoid behavioural change intended to apply on a class-wide basis (although this depends on adherence to the association ordering restrictions discussed in section 2.5.3). However, the architecture has no support for controlling the relationships between metaobjects associated with related base-level classes. Metaobjects may be associated with a class without reference to any metaobjects that may be associated with the class's superclasses or subclasses. While this allows flexibility in the application of behavioural changes it also means that a programmer cannot depend on an inherited method behaving in the same way that it does for the superclass. For example, if a class has a metaobject that adds access control to method invocations its subclasses will only have access control for the inherited methods if the same metaobject has been explicitly associated with each of them.

Closely related to this issue is the one of inheritance of metaobject association (see section 1.3.3). In MetaXa associations are not inherited at the base-level, so any behavioural change that is required to propagate down the base-level class inheritance tree must be explicitly associated with each subclass. For example, a metaobject associated with a class to apply synchronisation to method invocation will not affect even inherited methods in subclasses. Although this can be dealt with by explictly associating the metaobject with each subclass doing so is very prone to omission, particularly of subclasses added to the design later.

Behavioural change may be refined by subclassing metaobject classes in the normal way. For example, a metaobject implementing synchronisation constraints for a base-level class could be subclassed to add further states or constraints so that it could be used with subclasses of the original base-level class.

2.5.5 Anticipation, reusability, and separation of concerns

MetaXa supports a strong separation of concerns between the base-level and the meta-level. The base-level program does not need any embedded references to the meta-level, and does not require any preparation

before load time. The association of a meta-level program with the base-level program does not have to be anticipated in any way, and MetaXa can be used to associate behavioural change with arbitrary classes of a program at runtime.

Metaobjects, implementing behavioural change, can be reusable in their binary form. Constructor arguments can be used to specialise a metaobject when it is instantiated, and details of an association with a class do not have to be embedded in the code of a metaobject.

However, in practice the separation of concerns is degraded slightly by a difficulty in making the initial association between the base-level program and the meta-level program. Association can only be initiated by, at a minimum, creating instances of metaobjects and passing references to classes or objects as constructor arguments. This means that either the original program must be modified to include those statements or a special small 'stub' program must be used to start the original program and make the initial associations.

2.5.6 Implementation

MetaXa is implemented as a special virtual machine and a library of classes. The virtual machine is intended to implement the JVM specification (1st edition) [Lin96], and to contain the extra functionality needed to support MetaXa. There are no changes or extensions to the Java language.

Association of a single object with a metaobject consists of two steps. First the base-level object is bound to the metaobject, and then the metaobject is registered for specific events in the base-level object. The first stage, binding, is implemented by changing the class pointer in the object's handle to point to a new dynamically created copy of the original class. The new subclass, called a *shadow class*, is a shallow copy with some of its pointers referring back to the data structures in the original class. The object now appears to be an instance of the shadow class but its behaviour is initially identical to that of instances of the original class. Changing the class of the object allows the methods and fields used by the object to be modified without affecting other instances of the original class. To complete the binding the *meta pointer* in the new shadow class is set to point to the metaobject. Every class running in a MetaXa virtual machine has a hidden meta pointer which is null if there is no metaobject bound to the class.

To ensure that an object's apparent type is not affected by changing its class pointer MetaXa separates the concepts of an object's *class* and its *type*. Every object handle contains both a class pointer and a type pointer. In an unreflective object they both point to the object's class. The class pointer can be changed to a subclass of its original class, but the type pointer is never changed from its original class. The MetaXa virtual machine uses the type pointer when carrying out type comparisons, casting, or other type-dependent operations, so an object whose class pointer has been changed will still appear to be of its original type.

The second stage of association of an object with a metaobject is registration for events. Up to this point the metaobject and the base-level object have been associated with each other, but no base-level events have been intercepted or redirected to the metaobject (i.e. no operations of the base-level object have been reified yet). Registering for specific events in the base-level object causes the interception mechanisms to be

inserted into the appropriate bytecode locations for those events. The low level interception mechanism is alteration of bytecode in the class of the object. The altered bytecode invokes an event dispatcher in the associated metaobject instead of carrying out the original operation. The altered bytecode includes marshalling of any arguments to the operation. Because of MetaXa's shadow classes the bytecode changes affect only the intended base-level object. Other instances of the same original class are unaffected and do not suffer any execution time slowdown.

Association of a metaobject with an object reference is implemented in a similar way to association with an object, but using a cloned object handle containing a pointer to the shadow class instead of the original class.

Association of a metaobject with a class is simpler than association with a single object or reference. As the association is intended to affect all instances of the class it is not necessary to create shadow classes, and the class pointers and type pointers of all instances can remain unchanged. The meta pointer of the original class is set to point to the metaobject, and reification of events can be carried out by inserting the interception mechanism into the bytecode of the methods in the original class.

One of the principal goals of MetaXa is to minimise the inevitable execution time penalty incurred by introducing interception mechanisms. It achieves this by introducing the mechanisms only for classes and objects that have associated metaobjects. The dynamic insertion of required interception mechanisms ensures that there is no performance effect on unreified operations in a class or object, and the shadow class technique ensures that operations can be reified for one instance of a class without slowing execution of the same operation on other instances of the class.

2.5.7 Summary of MetaXa

MetaXa achieves its principal objectives by providing a general reflective architecture to support dynamic association with a strong separation of concerns. It eliminates any impact of the interception mechanism on unreified operations, thus avoiding execution time penalties when reflection is not used. Its principal weaknesses are in its lack of control of the relationships between associations with classes, its need for a wrapper program to bootstrap the meta-level, and its lack of support for explicit composition of metaobjects. The restrictions on the order of overlapping uses of the three association techniques (class, object, and reference) could be a serious limitation on the ability to provide dynamic adaptation in some circumstances.

2.6 Guaraná

Guaraná [Oli99] uses a specialised JVM to provides reflective facilities, focusing on metaobject composition and security. Guaraná was designed in the University of Campinas, Brazil, in 1997/98 by Alexandre Oliva.

The Guaraná architecture is intended to support dynamic adaptation of programs by associating multiple metaobjects with base-level objects. It supports a strong separation of concerns between base and meta-levels, and does not require preparation or anticipation in the base-level program It is notable for its flexible

support for composition of metaobjects, and for its attention to the security problems inherent in allowing the behaviour of program classes to be changed.

2.6.1 Programmer's model

Guaraná supports the association of metaobjects with classes and with individual objects. Metaobjects are designed to add behaviour to operations on the class or object with which they are associated. All classes and objects begin life with no associated metaobjects, and may have one or more metaobjects dynamically associated with them at any time.

Association is carried out by invoking static methods of class `Guarana`, giving a reference to the base-level class or object and to the metaobject. Metaobjects must be subclasses of Guaraná class `MetaObject`. Associating a metaobject causes all method invocation and field access operations on the class or object to be reified. When a reified operation occurs an instance of class `Operation` is created to represent the operation and its arguments and is passed to the metaobject for handling.

```
public class TraceMO extends MetaObject {
   public Result handle(final Operation op, final Object o) {
      System.out.println("Before: " + op);
      return Result.modifyResult;    // We want to handle 'after'.
   }
   public Result handle(final Result res, final Object o) {
      System.out.println("After: " + res.getOperation());
      return null;
   }
}
```

Figure 2.3: A Guaraná metaobject to trace all method calls and field accesses.

Although a base-level class or object can be associated with a number of metaobjects the class or object has a direct link to just one metaobject, known as its *primary* metaobject. The primary metaobject may contain references to further metaobjects, and may delegate reified operations to them as appropriate. Metaobjects that delegate are called *composers*, and must be subclasses of `Composer` (which is itself a subclass of `MetaObject`).

Delegation establishes not just a hierarchy of operation handling but a hierarchy of authority. Requests for a metaobject to be added to or removed from the meta-level configuration of a class or object have to be approved by the existing configuration, starting with the primary metaobject. The request may be delegated down the hierarchy in any appropriate way, and any composer may decide to approve or disapprove the request without delegating.

A meta-level programmer implements behavioural change by overriding the `handle()` methods of class `MetaObject` or `Composer` (see figure 2.3). There are two handler methods. One is called before the reified operation is carried out, and the other is called (optionally) after the operation. They are effectively 'before' and 'after' handlers. The 'before' handler receives the `Operation` instance, and a reference to the

target class or object. It may examine (and possibly alter) the target operation and its arguments. The 'before' handler may also indicate whether or not it wishes the 'after' handler of its metaobject to be called, or it may supply a result value and cause the reified operation to be skipped. The 'after' handler may examine and modify the result of the operation. Typical composition behaviour would be to send the `Operation` instance to the 'before' handlers of all metaobjects in a composition in turn, execute the reified operation, and then send the result to interested 'after' handlers in turn. Handlers may be designed independently of particular base-level operations, and may be specialised at runtime by using arguments to metaobject constructors.

Operations reified by Guaraná include method invocation, field read and write, and monitor entry and exit. Operations on both public and private members are reified. Creation of an object cannot be reified at the class of the new object, although it can be reified in the class that causes the object creation (see below); this can be thought of as reifying outgoing creation but not incoming creation. Guaraná does not include structural reification categories, as it uses the Java reflection classes to represent methods, fields and classes. It does not support examination of execution context so metaobjects cannot be context sensitive.

As part of Guaraná's attempt to provide some security for meta-level programs it does not provide any means for any metaobject to acquire a reference to any other metaobject (other than composer metaobjects holding references to their own delegate metaobjects). To support communication between metaobjects associated with different base-level objects Guaraná provides an indirect message transmission mechanism (called *broadcast*). This allows `Message` objects to be sent to the primary metaobject of a specified base-level object without knowing anything about the destination metaobject. The primary metaobject may distribute the message to other metaobjects in its configuration. Metaobjects associated with different base-level objects may communicate, but the indirect nature of the communication prevents a rogue metaobject from making unauthorised modifications. The architecture also enforces a security barrier between the base-level and the meta-level by not having any means for a base-level object to acquire references to its own metaobjects. Although a reference to the primary metaobject is stored in the base-level class or object it is hidden from Java programs.

One of the most notable aspects of Guaraná is the way that it handles creation of a meta-level configuration for new objects. In most other dynamic adaptation architectures the meta-level configuration of the new object is a copy of that of the object's class. This is the case for MetaXa, PROSE and Iguana/J. In Guaraná the meta-level configuration of the new object is created by the meta-level configuration of the class or object that created the new object. After this is done the meta-level configuration of the class of the new base-level object is given an opportunity to modify the new meta-level configuration. This means that it is possible, even probable, that instances of one base-level class could have completely different meta-level configurations. Meta-level configurations tend to propagate from creating object to created object, regardless of their classes. The benefit of this approach is that a group of related objects can be adapted for a particular use, independently of the other instances of the same classes. It implies that Guaraná is intended for adapting related objects to particular usage circumstances in a part of a program rather than for adapting a class and its instances across a whole program.

2.6.2 Composition model

Guaraná's supports delegation from composer metaobjects to other metaobjects. A composer is simply a specialised metaobject that holds and uses references to other metaobjects. By designing appropriate composers a programmer may implement almost any composition model. Security is supported by requiring existing metaobjects to approve or disapprove of any changes to the meta-level configuration of which they are a part, and by preventing metaobjects from acquiring references to each other. Anonymous communications between metaobjects is supported by the Guaraná API.

Composition is not automatic. There is no composition mechanism built in to Guaraná. Composer metaobject classes must be created by the meta-level programmer (although some basic composer classes are included in the Guaraná class library). Multiple metaobjects can only be associated with a class or object if at least the primary metaobject is a composer and agrees to accept additional metaobjects into the composition. This helps to guard against invalid compositions by giving metaobjects within the composition a veto on extending the composition.

How reified operations are processed by the composition of metaobjects depends on the design of the composer, and on how it delegates the operation to the other metaobjects. The separated 'before' and 'after' form of the operation handling, and the iterative structure of the delegation makes the composition more robust. Passing of an operation between composed metaobjects is not dependent on any individual metaobject (other than the composer metaobject). It is not as easy for a badly designed metaobject to disrupt the flow of the operation through a Guaraná composition as it is in architectures that use a simple linear composition chain.

While it is possible to design composer classes to suit a particular requirement the Guaraná class library contains some that provide common composition models. For example, the `SequentialComposer` class delegates to metaobjects in a way that gives similar behaviour to that of the linear chain of metaobjects used by MetaXa and by Iguana/J.

2.6.3 Association

Guaraná supports association of metaobjects with classes and with individual objects. Association is initiated by using a static method `reconfigure()` of class `Guarana`, providing it with a reference to the base-level class or object and to the metaobject (see figure 2.4).

Every association with a class or object is independent. Association of a metaobject with a class affects operations on static members of that class, but it does not affect (non-static) operations on instances of the class and it does not affect superclasses or subclasses. Guaraná does provide a mechanism that allows the meta-level configuration of a class to influence the meta-level configuration of its instances as they are created, but not the meta-level of previously created instances. There is no support for changing the behaviour of all existing instances of a class.

Association of a metaobject with a single object affects only that object. If the object already has metaobjects associated with it (perhaps put there by the object's class when it was created) then the new one is additional, not a replacement. Metaobjects associated with a base-level object affect operations on declared and inherited members.

Guaraná does not distinguish between categories of base-level operation when associating a metaobject with a base-level class or object. Associating a metaobject reifies all operations, on public and private members, even if the metaobject is interested in just one kind of operation. A metaobject can limit itself to specified operations by performing runtime checking of names, so narrowing of association scope must be part of the metaobject rather than part of the association.

```
//...
obj = new Object();
mo = new TraceMO();
Guarana.reconfigure(obj,null,mo);
//...
```

Figure 2.4: A fragment of code to create and associate a Guaraná metaobject.

2.6.4 Inheritance and refinement of behavioural change

Guaraná focuses on adaptation of the behaviour of individual objects. It does not have a concept of adapted behaviour forming part of the base-level inheritance or instance-of relationships. Behavioural change is not inherited at the base-level, and is not propagated from a class to its instances. This makes it difficult to use Guaraná to apply behavioural change to all existing instances of a class, or to ensure that a behavioural change applied to a class is also applied to all its subclasses.

If specific behavioural change is to be applied to all instances of class `X` and to all instances of its subclasses then the metaobjects representing the change must be explicitly associated with `X` and with each subclass in turn. To ensure that all instances of the classes are associated with the behavioural change a meta-level configuration must be associated with all the affected classes before any instances are created, so that the classes' meta-level configurations can influence the meta-levels of each instance as they are created. Applying a behavioural change to all instances of a class can also be done by finding each instance individually and associating the meta-level with it, but this may be impractical in many programs.

There is no support for refinement of behavioural change, other than simple subclassing of individual metaobject classes. It is not possible to refine or extend the complete meta-level configuration associated with one base-level class or object for use with another base-level class or object.

2.6.5 Anticipation, reusability, and separation of concerns

Guaraná does not depend on hooks inserted into the program, and does not require the program to contain references to the meta-level program, so no anticipation of adaptation is required. It can be used to adapt programs for which no source code is available.

Metaobjects do not need embedded references to the program with which they will be associated. Arguments to metaobject constructors can be used to specialise metaobjects as they are created, so they can be designed to be reusable in binary form. Guaraná's sophisticated composition support allows metaobjects to be composed dynamically and safely in combinations that were not anticipated when they were designed.

These features provide Guaraná with a strong separation of concerns. Behavioural change, implemented as metaobject classes, can be dynamically associated with arbitrary classes and objects in the program. However, initiation of the association requires references to objects in both the base-level and the meta-level, and therefore it suffers from the same difficulty of making the initial association as MetaXa does. Either the original (base-level) program must be altered to create at least one dynamic association at startup, or a special small 'wrapper' program must be used to start the original program and to make an initial dynamic association. Either approach weakens the separation between the base-level and meta-level components.

2.6.6 Implementation

Guaraná is implemented as a modified version of the Kaffe open source Java virtual machine [Kaf03] and a library of classes. The principal class in the library is `Guarana`. It provides static methods for manipulating associations between base-level objects and metaobjects.

The link from a base-level object to its primary metaobject is a hidden non-static field added to class `Object` and inherited by all other classes. As a class is itself an object it too has this hidden field. The field contains a pointer to the associated primary metaobject, or null if there is no meta-level for that class or object. The interception mechanism is implemented by changing the interpretation of invocation and field access bytecodes so that they check the meta pointer of the target object (for operations on non-static members) or class (for operations on static members).

This implementation approach means that reification depends only on the presence or absence of a value in the meta pointer. One of the main benefits of this is that it allows Guaraná to intercept field accesses to any given class or object without the need to search all classes in advance to find all code that performs such access. It also allows method invocations to be intercepted without modifying bytecode at the start of the method (as is done by MetaXa). The disadvantage of the approach is that it adds an overhead to the execution of all invocation and field access bytecodes, even for unreified operations. In addition, as reification is indicated by the presence of a single meta pointer in a class or object all reifiable operations on that class or object are redirected to the meta-level, even those that are not required. This means that the full interception and redirection overhead is incurred for all reifiable operations on any base-level class or object that has an associated meta-level, and that the handler of each metaobject must explicitly filter redirected operations to detect the required ones.

2.6.7 Summary of Guaraná

Guaraná supports unanticipated dynamic adaptation with a strong separation of concerns, and provides a sophisticated composition model. It addresses some of the security issues that arise in dynamic adaptation by requiring existing metaobjects to approve any changes to their configuration, and by minimising access to metaobject references.

The interception technique used, modification of interpretation of interesting bytecodes, simplifies reification of operations but at the cost of a slight overhead in all executions of those bytecodes.

The main limitations of Guaraná are that it intercepts all operation categories leaving the metaobject to filter out the ones of interest, it does not support inheritance and refinement of behavioural change, and separation of concerns between base and meta-levels is weakened by the need to make at least one dynamic association to start the meta-level program.

2.7 PROSE

PROSE [Pop01] is a dynamic AOP architecture for Java. It comes from research on dynamic weaving at the Swiss Federal Institute of Technology, Zurich, by Andrei Popovici. PROSE attempts to extend AOP from static compile time and load time weaving to dynamic runtime weaving. It differs from the other unanticipated dynamic adaptation architectures discussed here because, like most AOP architectures, it is intended to support the insertion of additional non-functional behaviour rather than the modification of existing behaviour.

It supports cross-cutting mechanisms at runtime that are similar to the cross-cutting mechanisms supported by static AOP weaving, and this makes it different from the class and object based association mechanisms of reflective architectures such as MetaXa, Guaraná and Iguana/J. One of the stated objectives of PROSE is to avoid the specialised aspect languages of approaches such as AspectJ [Kic01] and Hyper/J [Tar00a], and to allow aspects and crosscuts to be expressed in the language of the program being modified.

PROSE is presented as a platform for rapid prototyping and testing rather than as a production platform. This is mainly because its implementation is based on the JVM Debug Interface (JVMDI) [Sun99], which has a significant effect on the execution time of a program.

2.7.1 Programmer's model

PROSE is not intended to support the association of additional behaviour with particular classes or objects. Rather it is intended to *weave* additional behaviour into a specified set of points across a whole program. The weaving is class-based, and the architecture does not support weaving with individual objects. Only behavioural weaving is supported. That is, it is possible to insert additional behaviour, but it is not possible to change class structure by, for example, adding or changing methods or fields. The code inserted by weaving is referred to as *advice*, a term that originates from AspectJ.

A meta-level programmer must create *crosscuts* to implement the advice (the additional behaviour) and to define the points in the program at which the advice should be woven. A crosscut must be a subclass of `Crosscut`. A PROSE *aspect* contains one or more crosscuts, and must be a subclass of `Aspect`. To cause the weaving to take place the aspect must be inserted into the PROSE system at runtime. This causes PROSE to instantiate the aspect and the crosscuts that it contains, and it causes the advice to be associated with the specified join points in the program.

The example in figure 2.5 shows a crosscut class that implements advice to display a message on entering any method of any subclass of `SomeClass`. The crosscut class overrides method `ANYMETHOD()` to implement the advice. By default `ANYMETHOD()` is for advice that is to be woven at all method invocations in all classes, so `setSpecializer()` is used to restrict the weaving to classes of interest. The single argument to `setSpecializer()` is a specialiser object, an instance of `Specializer` or one of its subclasses. The specialiser object narrows the set of weave points by specifying additional conditions. In the example the specialiser object restricts weaving to subclasses of `SomeClass`.

```
class TraceCrosscut extends MethodCut {
   // Instance initialisation:
   { setSpecializer(DeclarationS.inSubclass(SomeClass.class)); }

   public void ANYMETHOD(ANY a, REST r) {
      System.out.println("Starting a method in a subclass of
           SomeClass");
   }
}
```

Figure 2.5: A PROSE crosscut containing advice to announce method invocations.

```
class TraceAspect extends AbstractExtension {
   // Instance initialisation:
   Crosscut cc1 = new TraceCrosscut();
}
```

Figure 2.6: Placing the crosscut in a PROSE aspect..

```
//...
TraceAspect asp = new TraceAspect();
ExtensionSystem.insert(asp);
//...
ExtensionSystem.withdraw(asp);
```

Figure 2.7: Inserting (and removing) the PROSE aspect.

Figure 2.6 shows an aspect class that includes just one crosscut. It can include references to as many crosscuts as required (each including advice and specialisers). Figure 2.7 shows the code fragment needed to insert the advice. Insertion causes the PROSE runtime system to attach interception mechanisms to the specified points and to register the given advice for those points. When program execution reaches any join

point control is passed to the associated advice. When the advice returns the program resumes at the join point.

If the set of join points for the advice is required to be selected by a combination of conditions then the specialiser object passed to `setSpecialiser()` must be created by a chain of invocations of static methods in PROSE library classes. Such a chain can become quite complex even for simple combinations of conditions. For example, to narrow the specialiser in figure 2.5 to include only methods whose name begins with "get" in subclasses of `SomeClass` we would have to replace the argument to `setSpecializer()` to be:

```
MethodS.named("get*").AND(DeclarationS.inSubclass(SomeClass.class))
```

Aspects may be designed to be independent of particular method names and then specialised at runtime by using parameters to aspect and crosscut constructors.

PROSE supports the examination of arguments to an operation intercepted at a join point, but it does not support alteration of the arguments nor replacement or cancellation of an intercepted operation. This is consistent with the architecture's goal of support for weaving of additional behaviour into a program, rather than changing existing behaviour. PROSE also lacks support for examination of execution context by advice woven at a join point, and therefore cannot support context sensitive aspects.

2.7.2 Composition model

Composition in PROSE is dynamic and implicit; that is, aspects may be composed and uncomposed at runtime, but the programmer has no explicit control over the composition. Composition occurs automatically within the PROSE runtime system if multiple aspects affect one point in the program. In other words, if sets of join points specified by different crosscuts intersect then the join points in the intersection will have more than one piece of advice associated with them. Each piece of advice is registered separately, as part of the insertion processes for the aspects that include them. There is no explicit composition of the separate pieces of advice; composition is performed by the PROSE system by registering each of them for the same join point event.

When the program reaches a join point for which more than one advice is registered then each piece of advice will be invoked in turn. There is no *before* and *after* (all advice is *before* the join point at which it is woven), so the operation at the join point is simply dispatched sequentially to each advice registered for the join point. There is no support for explicit interaction between advice woven at one join point, or for ensuring that separate pieces of advice do not conflict.

2.7.3 Association

Weaving is initiated by invoking a static method of class `ExtensionSystem` (see figure 2.7). It takes an instance of the required aspect class as its sole argument, and the system weaves the aspect according to the specialiser conditions specified in the crosscuts included in the aspect.

Weaving is always on a class basis, affecting all existing and future instances of a class equally. The crosscut specialiser may specify join points across a set of classes or in a single class, but there is no support for weaving advice with only a single instance of a class. PROSE supports weaving at join points on both static and non-static members of a class.

Weaving expressions in crosscuts start from the broadest possible scope, and must be narrowed to the required set of join points. Without specialisers most crosscuts will match a very large set of join points across all classes in the program. Specialisers define filters that reduce the initial set of join points to the required set. Most other architectures take the opposite approach, where the set of points at which to associate the additional behaviour is initially empty and the programmer must add to the set.

Weaving advice with a single class does not normally affect subclasses. If it is required that advice should affect subclasses then the specialiser `DeclarationS.inSubclass()` must be included in the crosscut. If the subclass condition is used then the associated advice will always be woven with all subclasses of the named class, even if other advice is also woven at the same points.

Advice can only be woven with externally visible points of a class. It cannot be woven with operations on private members of a program class, because it is not possible to refer to private members from the crosscut and specialiser classes.

2.7.4 Inheritance and refinement of behavioural change

PROSE is intended to weave additional behaviour (advice) across a specified set of join points in a program. The architecture supports inheritance of advice between program classes, but only if the subclass condition is explicitly specified as part of the join point expression within the crosscut.

Aspects may be refined by subclassing aspect and crosscut classes, but PROSE does not control the relationship between advice woven with related classes. If a particular piece of advice is woven with a class `X` then subclasses of `X` may have no advice or even completely different advice woven with them.

2.7.5 Anticipation, reusability, and separation of concerns

PROSE supports a strong separation of concerns between the program and the aspect components. No anticipation of weaving (adaptation) is required in the program. The interception mechanisms are inserted at runtime using the JVMDI without any preparation of the program source code or bytecode, and the program does not need to contain any references to the aspect components.

The aspects and crosscuts, implementing the behavioural change, may be designed to be reusable. Although a crosscut class contains a specialiser to specify where its advice should be woven, the specialiser can be provided as a constructor argument to the crosscut class. This allows the advice and where it should be woven to be specified separately, and thus the crosscut can be designed to be independent of any particular class and method names.

The JVMDI used to implement PROSE includes a remote interface for manipulation of breakpoints and watchpoints from outside the running JVM, and therefore provides a means for PROSE to support remote insertion of aspects without anticipating it in the program or the aspect components. This allows PROSE to avoid the need for a special program wrapper to provide a start point for the meta-level program, and thus to maintain the strong separation of concerns between program and aspect components. While the program wrapper approach can be used for simplicity, remote aspect insertion via JVMDI allows the program to be started as normal without any consideration for dynamic aspect weaving.

2.7.6 Implementation

PROSE is implemented using the JVMDI [Sun99]. It does not need a special JVM and it does not need preprocessing of program source or bytecode. The JVMDI supports setting of code breakpoints and field watchpoints. PROSE builds on top of these mechanisms to support interception at points such as method entry and exit, and field read and write.

When a crosscut is to be inserted into the runtime system the PROSE manager examines it to identify the set of join points at which it is to be woven. It uses the JVMDI to create a breakpoint or watchpoint at each of the join points in the set, and it registers the advice and specialisers of the crosscut in an internal registration list. When the program reaches a breakpoint or watchpoint the PROSE manager is given control, and the program is suspended. The PROSE manager uses its registration list to decide which, if any, crosscut advice should be called.

Using the JVMDI means that it is possible to intercept almost any operation in the program, and to execute additional code at those points. However using the JVMDI has a number of implications for the architecture. It does not support skipping or redirection of operations at a breakpoint so it is not possible to alter existing program behaviour, only to insert extra behaviour. It has a significant effect on the execution speed of a program. All method invocations and field accesses in all classes and objects are 2 to 5 times slower [Pop01] when the JVMDI is enabled, even when no breakpoints or watchpoints are set. In addition, using the JVMDI means that the JIT compiler features of the JVM must be disabled so any execution time improvements that might have been achieved are lost.

Because PROSE works from outside the virtual machine it is restricted in how it accesses the program. The main difficulty is that it cannot refer to private members of program classes, and therefore it cannot insert breakpoints or watchpoints on private members. This in turn means that join point expressions cannot refer to private members of program classes. This could be overcome by extensive use of the Java reflection API, but the designers of PROSE considered that the cost was not worth the benefits [Pop02].

2.7.7 Summary of PROSE

PROSE supports class based application of behavioural change to a program. It supports the expression of advice (behavioural change) and join points in the same language as the program, and supports a very strong separation of concerns between the program and the aspects.

It is intended to be an AOP architecture, and therefore its objectives are more focused than those of a general dynamic adaptation architecture. It supports the insertion of additional behaviour, but not modification of existing behaviour. Its join point expressions provide support for program-wide crosscuts, but are awkward to use for a single class and its subclasses. It does not support weaving of aspects with operations on private members of classes, or weaving with individual objects. It performs automatic composition of aspect advice where more than one piece of advice must be woven at the same join point.

In attempting to support sophisticated join point expressions without using a specialised language PROSE relies on a complex library of over 60 Java classes and over 30 exceptions and interfaces. It could be argued that it would be easier to learn a simple specialised language. PROSE allows advice to be woven across the program without regard for the inheritance relationship between classes, and therefore it must be used with care if advice is to be applied to methods even when they are inherited. The initial implemenation of PROSE, based on the JVMDI, suffers from significant execution time slow down even when no aspects are woven.

2.8 Summary

There is a growing number of architectures that support adaptation in various ways, but most of them support static binding mode or require static anticipation of the adaptation. Only a small number of architectures address unanticipated dynamic adaptation. We have reviewed three such architectures here, Guaraná, MetaXa and PROSE. All three are based on the Java language, and take advantage of various features of the Java runtime system to implement dynamic interception mechanisms.

MetaXa is a reflective architecture and supports addition and modification of behaviour, with a focus on minimising the performance impact of the architecture on the program. Execution of unadapted operations on classes or objects is unaffected by the presence of MetaXa, and the inevitable performance cost is only incurred for adapted operations.

Guaraná also supports adaptation by using reflection to add or modify behaviour, but it focuses on security and dynamic composition of metaobjects. It is the only architecture for unanticipated dynamic adaptation that addresses security and composition structure.

PROSE is an AOP architecture to support dynamic weaving of additional behaviour (aspects) across a set of join points. It does not support modification of existing behaviour. Its principal goal is to support dynamic weaving with a strong separation of concerns but without the use of a specialised language to express crosscuts. This has been achieved at the cost of a complex class library. Unlike Guaraná and MetaXa, PROSE does not support adaptation of individual objects.

None of these three architectures address the challenge problems of context sensitivity (section 1.3.2) and the inheritance anomaly (section 1.3.3), and none provide any support for maintaining structure and consistency between adaptations applied to related classes or objects. These issues are addressed by Iguana/J, the architecture proposed by this thesis, which is described in the following chapter.

Chapter 3

The Iguana/J Programmer's Model

It is the simple thing that is so hard to achieve.

- Bertholdt Brecht.
Die Mutter, 1932.

This chapter describes some of the most important features of the Iguana/J programmer's model, and the motivation behind important design decisions affecting the model. This is not intended to be a programming manual for Iguana/J, but rather to give the reader an overall understanding of a programmer's view of it, to explain some of the model's concepts in more detail, and to discuss the research issues supporting the design decisions taken for the model.

3.1 Introduction

The Iguana/J architecture uses reflective techniques to support unanticipated dynamic adaptation of Java programs. It reifies, or makes concrete and manipulable, specified operations in a program, such as method invocation, object creation, and field read and write, and supports the application of behavioural change to those operations as the program executes. Replacement implementations are provided by metaobject classes that are developed and deployed separately from the original program, and may be inserted and removed at runtime. The original program forms the base-level of the reflective architecture, and the metaobjects (instances of metaobject classes) are part of the separate meta-level program (see figure 3.1). Reification of an operation at a point in a base-level program allows the program's behaviour at that point to be reimplemented without explicitly changing the code of the original program. The operations that may be reified include object creation, method invocation, and field access.

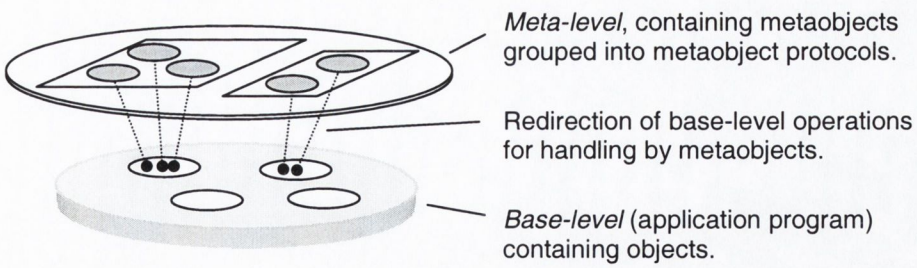


Figure 3.1: The two levels of the Iguana/J model.

At a more mechanical level Iguana/J can be understood as intercepting selected operations in the base-level and redirecting them to handlers (metaobjects) in the meta-level program. Handling is executed in the same thread as the intercepted operation. When the metaobject returns control to the base-level program execution continues just after the intercepted operation, so that the actions of the metaobject replace the intercepted operation. The metaobject can examine arguments or execution context, and can carry out any required sequence of actions, including proceeding with the intercepted operation. This means that a metaobject can carry out any combination of 'before' actions, replacement operations, and 'after' actions.

Replacement of the implementation of specified operations in the base-level program alters the behaviour of the program at those points, and is the mechanism that supports adaptation of the base-level program. Iguana/J supports changes to the internal behaviour of classes and objects; it does not support modifications to their external interfaces and it does not support structural modification such as the additon or removal of methods or fields. Iguana/J makes use of the existing Java reflection classes to represent and operate on structural elements such as classes, methods and fields, and provides a set of classes to represent and operate on behavioural elements such as invocation and object creation.

Metaobjects, representing replacement behaviour for specified base-level operations, are instances of Java classes. Metaobjects are grouped together to form a metaobject protocol (MOP) representing some coherent behavioural change. At runtime a metaobject protocol may be associated (and unassociated) with a base-level class or object. This association causes specified operations in the base-level class or object to be reified by the appropriate metaobjects in the metaobject protocol.

3.2 Objectives

Iguana/J is based on previous research into reflective languages by Gowing (Iguana) [Gow97] and more recently by Schäfer (Iguana/C++) [Sch01]. Iguana/J itself is not a reflective language and does not cause Java to become a reflective language. Iguana/J treats reification as a change to a program rather than as a change to a language; it provides features that allow a system rather than a language to become reflective. This approach is evident in the fact that Iguana/J does not extend or change the Java language and does not require any modifications to the source code of the program that is to be adapted.

Starting with the basic concepts of reification categories and metaobject protocols from the previous research by Gowing and by Schäfer, this research extends the scope and refines the semantics of that model of reflection to support unanticipated dynamic adaptation. New features such as dynamic association of behavioural change with classes, association without preparation of the base-level program, protocol parameters and greatly extended concepts of metatype and association scope, make the Iguana/J architectural model more dynamic, more structured and more consistent than the earlier Iguana models.

The principal goal of Iguana/J is to support the application of behavioural changes to a running Java program, without requiring preparation of any kind in the program and with a set of features that allows a number of important issues in the field of dynamic adaptation to be addressed. Iguana/J does not attempt to support structural changes such as addition or removal of methods or fields, or any change that alters the visible interface of a class or object. Iguana/J is intended to provide a structure for manipulating behavioural change at runtime that ensures changes are applied in a consistent manner to related classes and objects.

Existing architectures supporting unanticipated dynamic adaptation allow change to be applied in an undisciplined way that can leave subclasses without behavioural changes applied to their superclass, and can leave instances without changes applied to their class. A number of researchers have referred to the need for structure to help manage the use of dynamic adaptation. In the context of reflection and adaptation of real-time systems Mitchell referred to "... a requirement for some form of type system for meta-objects" [Mit97]. Mens et al. commented on the difficulties caused by the uncontrolled power of reflection, and on the need for constraint and structure at the meta-level [Men97]. Even earlier, Maes commented on the need for "discipline in the use of reflection" and suggested that "... safer (weaker) versions of reflective facilities will have to be designed" [Mae87].

Despite these observations there have been very few proposals to address the issue. The Guaraná architecture [Oli99] imposes some constraints in the use of reflection, but they are concened with security and authorisation within the metaobject composition process rather than with the general structure of behavioural change. SOM [For98] uses *metaclass constraints* to ensure that a metaclass cannot damage the substitutability of instances of a class for instances of any of its superclasses. The motivation for SOM's metaclass constraints is to maintain structural (i.e. methods and fields) rather than behavioural integrity within the class hierarchy. Iguana/C++ introduces the concept of a metatype to represent reflective alterations to the semantics of a language, but it is limited in scope and is not applied to dynamic adaptation of programs. Iguana/J addresses the issue of structure and constraint at the meta-level by adapting and extending the concept of a metatype to represent dynamically applied behavioural change. This supports a more predictable association between the program and the behavioural change, in which the programmer can have more confidence that the changes do not disrupt the inheritance or instance-of relationships within the program.

Separation of concerns, between the program to be adapted and the components implementing behavioural change, is required in order to avoid anticipation of change. Separation supports reuse [Hur95] [Aks96] and later binding (association) between the program and the behavioural change components [Cza00]. An architecture supporting unanticipated dynamic adaptation should provide a separation in which the compiled behavioural change components can be associated with arbitrary parts of a program at runtime (within the scope of the types of change supported by the architecture). If a compiled component is limited at

runtime to association with a set of statically prepared points in the program then it cannot be said to support unanticipated change. A principal objective of Iguana/J is to support adaptation that is unanticipated in any way by preparation of the program.

Some existing architectures support unanticipated association of change with arbitrary classes or objects in the program, but still require changes to the program in order to insert the statements that initiate the association. In other words, while they support unanticipated change they require invasive changes to initiate it. This is the case with Guaraná and with MetaXa [Gol97]. Iguana/J aims to provide a strengthened separation of concerns, in which no invasive changes are required to initate the association. PROSE [Pop01] provides such a separation (see section 2.7.5) although its support for dynamic adaptation is limited in other ways[4].

An important objective is to make the architecture as simple as possible for a programmer to use. The technique used to express behavioural change is a major part of a programmer's involvement with an architecture, and therefore has an important influence on this objective. Czarnecki [Cza00] proposes three possible techniques for expressing behavioural change. They are to use a conventional class library, a separate special language, or a language extension. Special or extended languages can support a more domain-specific expression of the behavioural change, capturing the intentions of the programmer more directly. On the other hand, the programmer must learn and use new language constructs, and preprocessors, special compilers, or modified interpreters are required to handle them. There are many examples of the use of the class library technique, including MetXa, Guaraná, PROSE, JAC [Paw99], Kava [Wel01], Reflex [Tan01], and Weave.NET [Laf03]. The separate language technique is used by AspectJ [Kic01], and the language extension technique is used by Iguana/C++. Iguana/J uses a combination of the class library technique (for metaobject classes) and a special language (for metaobject protocol declarations) as a compromise between ease of use and expressiveness. Simplicity is maintained by minimising the size of the class library and by limiting the syntax of metaobject protocol declarations. Simplicity is also maintained by integrating the added reflective features with the existing Java reflection classes, and avoiding extensions to the Java language.

In addition to structure, constraint, reuse, and separation, other objectives of the Iguana/J architecture are intended to improve support for dynamic adaptation in other ways. For example, it provides enhanced support for examining execution context, and thereby allows behavioural change to take effect based on semantic as well as syntactic information. A secondary objective for the architecture is to minimise the performance cost of using it, as a dynamic adaptation architecture incurs an inevitable runtime overhead in transferring program operations to and from the separate behavioural change components [Cza00]. Iguana/J also provides many of the features found in various combinations in other architectures that support unanticipated dynamic adaptation. These include application of change to a range of program operations such as method invocation, object creation, and field read and write, support for composing behavioural change components, absence of a need for anticipation or invasive alterations in the program to be adapted, absence of a need for access to the source code of the program to be adapted, and support for both adjusting the behaviour of a program operation and for inserting additional behaviour around an operation.

[4] For example, PROSE does not support redirection of operations, or changes to operations on private members of a class.

3.3 Overview

The Iguana/J model supports unanticipated dynamic adaptation by providing a reflective architecture at runtime. The three basic concepts in the Iguana/J model of reflection are metaobjects, metaobject protocols, and association. The simplest way to understand these concepts in the model is to examine the three implementation steps that must be taken by a programmer to adapt a Java program dynamically. The three steps are the creation of the metaobject classes to implement the new behaviour, the creation of the metaobject protocol, and the association of the metaobject protocol with the program to be adapted. The steps are discussed separately below.

Step 1: Implement the new behaviour.

When a base-level operation is reified its normal implementation is replaced by an implementation defined by a metaobject. Metaobjects are instances of metaobject classes, which must be created by subclassing appropriate classes from the set of seven Iguana/J metaobject classes (see table 3.1). Each of the seven classes deals with one kind of reifiable operation (one reification category). To implement new behaviour the programmer must subclass the appropriate one of the seven metaobject classes and override its implementation or handler method. For example, to implement new behaviour for invocation of a method the programmer should subclass `MExecute` and override `execute()`. The example in figure 3.2 shows a metaobject class called `Trivial` that prints a message just before and just after a reified method invocation. The `execute()` method may contain any valid Java code, may use Java and Iguana/J reflection methods to operate on arguments and results, and may use the inherited `proceed()` method to cause the original operation to be carried out.

```
public class Trivial extends MExecute {
     public Object execute(Object target, Object[] args, Method m) {
        System.out.println("Entering " + m.getName());
        Object result = proceed(target, args, m);
        System.out.println("Finished " + m.getName());
        return(result);
     }
}
```

Figure 3.2: Example of a metaobject class to handle method execution.

Step 2: Create the metaobject protocol.

A metaobject class must be declared in a metaobject protocol declaration before it can be used. The primary purpose of a metaobject protocol declaration is to connect named metaobject classes to named reification categories (kinds of operation) in a base-level class or object (not yet specified). A protocol allows metaobject classes that each provide new behaviour for one reification category to be grouped together under a single name. Metaobject protocols must be statically declared using a special syntax. The example in figure 3.3 declares a protocol called `SimpleProtocol` that uses an instance of `Trivial` to reify method invocation. Protocol declarations must be compiled by the Iguana/J protocol compiler. The result of the compilation is a Java class that is used to represent the protocol in the running system.

```
protocol SimpleProtocol {
   reify Invocation: Trivial;
}
```

Figure 3.3: A protocol declaration that connects class `Trivial` to invocation.

Step 3: Associate the metaobject protocol with the base-level program.

The results of steps 1 and 2 above are one or more metaobject classes, and a class representing the metaobject protocol declaration. Nothing in the procedure up to this point has involved the base-level program whose behaviour is to be modified. Association is a dynamic process that connects a named metaobject protocol with a named base-level class or an identified single object. Association of a protocol with a particular class can be initiated by using `Meta.associate()`, as in the example in figure 3.4. The method expects two arguments giving the name of the base-level class and the name of the protocol. Association causes operations corresponding to the categories declared in the protocol to be reified in the base-level class, and therefore to have their implementations replaced by the implementations defined in the metaobject classes of the metaobject protocol.

```
Meta.associate("testPackage.TestBaseClass", "SimpleProtocol");
```

Figure 3.4: Associating a named base-level class with a named protocol.

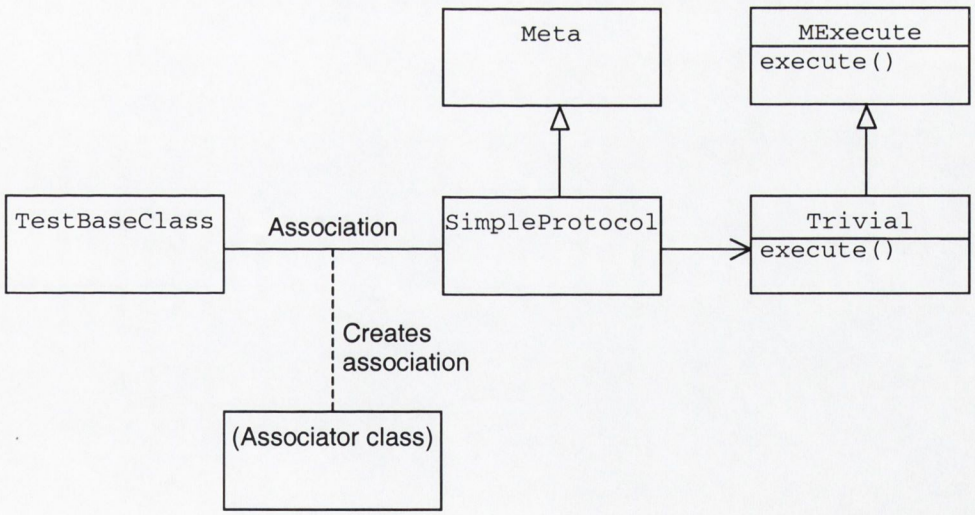


Figure 3.5: UML diagram of the class structure for the example in figures 3.2, 3.3 and 3.4.

Figure 3.5 illustrates the result of these three steps for the code examples given in figures 3.2, 3.3 and 3.4. The class `Trivial` is the metaobject class created in step 1, the class `SimpleProtocol` is the protocol class created from the protocol declaration in step 2, and the association between the base class and the

protocol is created in step 3. The 'associator class' in figure 3.5 is the class that causes the association by executing `Meta.associate()`.

The rest of this chapter deals with some details of the programmer's model, expands on the concepts of metaobject, metaobject protocol, and association, and discusses the use of the metatype concept.

3.4 Iguana/J classes

The Iguana/J class library contains a total of eleven classes (see figure 3.6) to support behavioural reification categories and metaobject protocols. This is less than the number of classes in the libraries for MetaXa, Guaraná, or PROSE, and is part of Iguana/J's support for a simplified programmer's model. Seven of Iguana/J's classes are metaobject classes to represent the seven reification categories (see table 3.1). To implement new behaviour a programmer must subclass appropriate classes from this set and override their handler methods. The seven classes are all subclasses of `ie.tcd.iguana.Metaobject`.

In common with most other architectures supporting unanticipated dynamic adaptation Iguana/J does not support structural adaptation such as adding methods or fields to a class. Despite this, some structural elements must be reified so that they may be examined and used as part of behavioural reification. For example, classes, methods and fields that can be the target of a reified operation must themselves be reified so that they may be examined and used by the object that reifies the operation. The reification of structural elements required by the Iguana/J model may be thought of as 'read-only' reification, supporting examination and use of the reified structures; it does not need to support making changes to the reified structures. The classes of the standard Java package `java.lang.reflect` implement just such 'read-only' reification for structural elements, so Iguana/J uses those classes to provide most of the structural reification that is required.

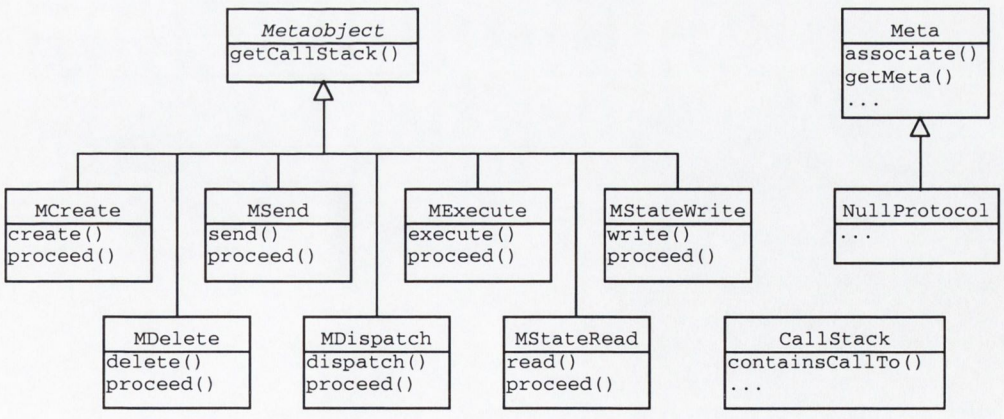


Figure 3.6: The eleven classes of the Iguana/J library.

Reified operation	Metaobject class	Handler method
object creation	`ie.tcd.iguana.MCreate`	`create()`
object deletion	`ie.tcd.iguana.MDelete`	`delete()`
method call (send)	`ie.tcd.iguana.MSend`	`send()`
method dispatch	`ie.tcd.iguana.MDispatch`	`dispatch()`
method execution	`ie.tcd.iguana.MExecute`	`execute()`
field read	`ie.tcd.iguana.MStateRead`	`read()`
field write	`ie.tcd.iguana.MStateWrite`	`write()`

Table 3.1: Classes and handler methods for the seven behavioural reification categories.

Structural element	Reification class
class	`java.lang.Class`
constructor	`java.lang.reflect.Constructor`
method	`java.lang.reflect.Method`
field	`java.lang.reflect.Field`
array	`java.lang.reflect.Array`
stack	`ie.tcd.iguana.CallStack`

Table 3.2: Structural reification classes used by Iguana/J.

The only structural element for which reification is required but which is not provided by the standard Java class library is the Java call stack, so Iguana/J provides a single structural reification class called `CallStack`. Like the structural reification classes provided by standard Java it provides 'read-only' functions and is final. All the structural reification classes used by Iguana/J are listed in table 3.2.

The Iguana/J class `Meta`, shown in figure 3.6, fills two roles in the architecture; it is the superclass of all classes representing metaobject protocols and it holds a number of static utility methods (such as `associate()`) to operate on the association between a class or object and a protocol. Metaobject protocol classes are discussed in more detail in section 3.6. `NullProtocol` is a special case of a metaobject protocol class, and is discussed in section 3.9.

3.5 Metaobject protocols

A metaobject protocol brings a set of metaobject classes together under a single name, and represents some coherent behavioural change that may be applied to a base-level class or object. Most architectures supporting dynamic adaptation (both anticipated and unanticipated) based on reflective techniques do not use the concept of metaobject protocol. They use direct association of individual metaobjects with base-level entities using one of two techniques; the metaobjects may represent individual behavioural changes and be associated as required (e.g. CodA[Mca95], Reflex [Tan01], MetaXa [Gol97], Guaraná [Oli99]), or a single metaobject may represent all the behaviour, both the original and the changes (e.g. ABCL/R [Wat88], LEAD++ [Ama99]). In the first technique each individual behavioural change is applied by associating another metaobject with the base-level entity, without reference to how it will interact with metaobjects

already associated with that base-level entity[5]. In the second technique messages are sent to the already associated metaobject to adjust its behaviour, and therefore that of the base-level entity, but there is no support for ensuring that an accumulated set of adjustments form a coherent overall behavioural change. MetaclassTalk [Bou99] also uses a single metaobject to represent all the behaviour of the base-level entity, but behavioural change is applied by replacing the metaobject rather than sending messages to it.

Iguana/C++ requires that groups of metaobjects be formed explicitly, as metaobject protocols, before they can be applied to a base-level class or object. This is more restricting on the meta-level programmer as it prevents random use of individual metaobjects at runtime. However, requiring metaobject protocols to be formed compels the meta-level programmer to become aware of the interactions between metaobjects used together. It reduces the possibility that metaobjects affecting a single part of the program will interact in unforeseen ways, and supports the construction of more complex behavioural changes from individual metaobject classes.

Iguana/J uses metaobject protocols in order to impose order on the association of metaobjects with base-level entities. Metaobject protocols are also used as the foundation for the metatyping structure that further extends the control over how behavioural change may be applied (see section 3.14). Protocol declarations in Iguana/J support features that are supported by Iguana/C++, such as reification categories, protocol derivation, and local and shared metaobject scope. Iguana/J extends metaobject protocols with additional features such as independent declaration, global scope, package assignment and parameters. The general objective in making these changes was to improve the separation between base-level and meta-level concerns, and to make the meta-level program more modular and more reusable. Reusability is improved by protocol parameters (discussed separately in section 3.12), while modularity is improved by independent protocol declaration (discussed below).

The basic form of a protocol declaration in Iguana/J is similar to the form used in Iguana/C++, although in contrast to Iguana/C++ the declarations are created in separate text files rather than embedded in the base-level source code. The separation of the protocol declaration from the base-level source code is an important change. Removing protocol declarations from the source code allows Iguana/J to avoid the need for a base-level source code preprocessor and greatly improves the separation between base-level and meta-level concerns. Giving protocol declarations an independent existence, separate from any base-level source code, also means that the scope of an Iguana/J protocol declaration is global. Access to the protocol is no longer restricted to the base-level source code file that contains the protocol declaration.

```
protocol SecurityProtocol {
   reify Invocation: CheckInvocation;
   reify StateRead: CheckRead;
   reify Creation: CheckCreation;
}
```

Figure 3.7: A protocol declaration with three reification categories.

[5] Although Guaraná allows already associated metaobjects to reject association of a new metaobject.

Protocol declarations use a simple syntax. Each declaration is introduced by the `protocol` keyword and contains zero or more statements assigning reification of named categories of operation to named metaobject classes. Each statement starts with the keyword `reify`, followed by a category name and the metaobject class name to be used for that category. There are seven possible reification names, corresponding to the seven reification categories (see table 3.1). The example in figure 3.7 declares that each of three categories should be reified by instances of the three metaobject classes `CheckInvocation`, `CheckRead` and `CheckCreation`. The three metaobject classes must be created by subclassing Iguana/J metaobject classes `MExecute`, `MStateRead`, and `MCreate` respectively. Associating this metaobject protocol with a base-level class will cause the appropriate operations in the base-level class to be handled by instances of each of the metaobject classes named in the protocol declaration.

The syntax for protocol declarations has been deliberately kept simple. Other architectures in which behavioural change components may be constructed from smaller components tend to have complex new syntax (e.g. AspectJ [Kic01]) or complex class libraries (e.g. PROSE [Pop01]).

Each protocol declaration must be compiled by the Iguana/J protocol compiler before runtime. The protocol compiler converts the declaration into the source code of a customised Java class, and then invokes the Java compiler to create a standard Java class file. The class generated by this process is automatically a direct subclass of `ie.tcd.iguana.Meta` and has the same name as the protocol declaration from which it was created. It embodies all the information about the protocol and must be made available to any system that uses the protocol. Protocol declarations are not normally dependent on any particular base-level program, so they may be compiled and distributed independently of base-level classes. For convenience the Iguana/J protocol compiler allows more than one protocol declaration in a single text file.

Because the scope of a metaobject protocol is global, and each protocol is represented at runtime by a customised Java class with the same name as the protocol, protocols can be assigned to a Java package. This ensures that protocols can be managed more easily, and avoids possible name clashes. A protocol is assigned to a package by including a package statement at the top of the protocol declaration file, using the same syntax as a Java package statement in a standard Java source file.

3.6 Metaobject protocol classes

In dynamic adaptation architectures the connection between a base-level class or object and a component implementing behavioural change is usually to an object representing the behavioural change. Architectures that support application of multiple behavioural changes to a single base-level entity may use one of two techniques. The first is by connection from the base-level entity directly to a primary metaobject (supplied by the programmer), either the first metaobject in a chain (e.g. MetaXa) or a 'composer' metaobject (e.g. Guaraná). A connection to a single metaobject can also be used with recursive reflection to support multiple behavioural changes at a single point by constructing a reflective tower [Smi82]. Examples of this approach include FRIENDS [Fab98] and MetaJ [Dou01]. The second technique for supporting application of multiple behavioural changes is by connection from the base-level entity to a fixed dispatching object (part of the architecture's kernel) that distributes the operation to behavioural change objects (e.g. PROSE).

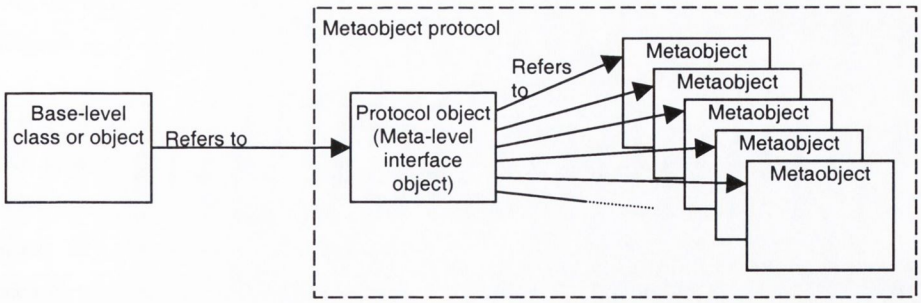


Figure 3.8 The role of a meta-level interface object.

Iguana/J uses a version of the first technique, in which the connection from a base-level entity to its behavioural change is a reference to an instance of a programmer-supplied metaobject protocol class. A metaobject protocol class is the class created by using the Iguana/J protocol compiler to compile a protocol declaration. The protocol class instance provides a point of reference for all the metaobjects in one protocol and simplifies the integration and manipulation of the metaobjects making up the protocol.

The instance of the metaobject protocol class fills three roles:

- It provides the primary link between the base-level class or object and its associated protocol.
- It holds references to the metaobjects making up the protocol.
- It supports some operations on the association between the protocol and the base-level class or object.

The instance of the protocol class collects the metaobjects together so that they may be manipulated as a unit. It acts as a meta-level interface object, providing a point of access to the set of metaobjects associated with a base-level class or object. The meta-level interface object is created automatically as an instance of the appropriate metaobject protocol class when the association between the protocol and a base-level class or object is created. Each base-level class or object may be associated with only one protocol at any time, so each has only one meta-level interface object.

The resulting general runtime structure is represented in figure 3.8. The metaobject protocol class can be considered as reifying the protocol itself, in order to support a limited set of operations on the protocol at runtime. These operations include construction of the metaobjects required by the protocol, dynamic association and disassociation of the protocol, and dynamic type checking. References at runtime to a particular metaobject protocol are actually references to the metaobject protocol class that represents it. If the declaration of the protocol included a package assignment then a reference to the protocol by name must be fully qualified by the package name.

While it might have been possible to create a metaobject protocol structure that did not use an explicit meta-level interface object it would still have been necessary to have a runtime representation of each protocol to contain information about the protocol declaration (such as which metaobject reifies which type of operation). In fact this is the approach that was used by Iguana/C++, mainly for reasons of

implementation efficiency. For Iguana/J it was decided that using the protocol object as the meta-level interface object was more consistent with the overall structure of the architecture, and that it provided a more flexible arrangement for possible future enhancements to the architecture.

As a basic safety measure the reference from a base-level class or object to its associated meta-level interface object is hidden, and is not directly accessible from Java code. This ensures that an association cannot be created or altered by simply changing the reference and thereby bypassing the formal association procedure. A copy of the reference can be acquired by using the static method `Meta.getMeta()`, with a reference to the base-level class or object as the single argument. This supports the following construct for acquiring the reference:

```
TestObject obj = new TestObject();
Meta m = Meta.getMeta(obj);
```

An alternative would have been to provide an accessor method in every class, to support the use of a construct such as:

```
TestObject obj = new TestObject();
Meta m = obj.getMeta();
```

The static method approach was chosen simply as an implementation convenience, as it avoids the need to insert a method into `java.lang.Object` for inheritance into all base-level classes.

3.7 Access to context information

When a reified operation occurs in the base-level program it is handled by a metaobject of the protocol associated with the object or class containing the reified operation. As one of the objectives of Iguana/J is to support behavioural change based on semantic as well as syntactic information a metaobject must be able to examine the arguments, return value, and execution context of an operation. In order to effect behavioural change with the widest scope the metaobject should also be able to alter the values of arguments or return values. For example, if the reified operation is a method invocation then the metaobject must have access to the arguments of the invocation to examine them, alter them or simply to proceed with the reified operation. If the reified operation is a field write then the metaobject must be able to examine and manipulate the value to be written.

One of the design choices for Iguana/J was how a metaobject should access the arguments of the reified operation that it was handling. Other architectures supporting dynamic adaptation provide access in various ways. For example, Iguana/C++ reifies the stack as an instance of `MStack`, Kava [Wel01] provides access via a `Context` object, PROSE uses an object implementing `java.util.List`, and MetaXa and Guaraná both marshall arguments into an array of type `Object`. The best technique for an architecture depends on what kinds of manipulations of the arguments are to be supported for the behavioural change components. Issues to be considered include whether alteration of arguments is to be supported, how well the technique

supports passing of the arguments between elements of the meta-level, and any implementation overhead implied by the technique.

For Iguana/J two possible approaches were considered. The first was to reify the Java stack and to support examination and manipulation of the arguments directly in the stack, and the second was to adopt the technique used by the existing Java reflection classes in the package `java.lang.reflect`. The Java reflection classes use an array of objects to hold arguments, with primitive types wrapped in instances of their equivalent classes. This is the technique that was selected for providing access to arguments to reified operations in Iguana/J. It supports examination and alteration of arguments, and supports the use of the Java reflection classes to represent structural elements such as methods and fields. For example, a method represented by an instance of `java.lang.reflect.Method` can be invoked using `invoke()`, passing the invocation arguments as an array of `Object`. By using the same marshalling technique for Iguana/J operation arguments can be passed between Iguana/J and the Java reflection classes in a common format.

The principal disadvantage of the array-of-objects technique is that it involves the creation of a number of objects, including the array object and wrapper objects for primitive argument types, each time a reified operation occurs. This slows down execution of reified operations. On the other hand, any alternative to this format for arguments would require either conversions back and forth in order to use the Java reflection classes (and therefore incur the slowdown anyway) or it would require a set of new structural reflection classes that support the alternative format. Some elementary implementation exercises were carried out to investigate the likely impact of an argument handling technique based on specialised structural reflection classes. The conclusion drawn was that such an approach would probably be a little faster to execute than the array-of-objects technique of Java reflection. However, the objective of simplicity of design and ease of integration with reflection classes already familiar to programmers was judged to outweigh the small improvements in runtime efficiency that might be gained by using the stack reification approach.

Access to return values is provided as type `Object` returned by the `proceed()` method (with primitives wrapped in equivalent class instances), and the value may be altered if required. In addition to access to arguments and return values, access to the execution context is also important [Tar00]. Such access allows a metaobject to take decisions based on the context in which the reified base-level operation was being executed. Some dynamic adaptation architectures do not support this (e.g. MetaXa, Guaraná, PROSE), but it is supported by others (e.g. Kava, Iguana/C++, JAC [Paw99]). In Iguana/J a stack reification class, `CallStack`, is provided to support context examination, despite the decision not to use stack reification for access to arguments. `CallStack` is a structural reification class that supports examination by a metaobject of the context in which the reified operation has occurred. An instance of `CallStack` is available to each metaobject and represents the chain of active method invocations up to the point where the reified operation occurred. The metaobject can use the instance of `CallStack` to discover whether a named method is in the chain, and it can make context sensitive decisions based on that information. `CallStack` does not support modification of information in the stack, or access of any kind to arguments in the stack.

3.8 Protocol derivation and metaobject composition

Derivation of protocols from existing protocols and composition of metaobjects are very closely linked to each other because the principal effect of derivation is to cause composition. This section discusses how they operate, and examines some implications of their operation.

3.8.1 Protocol derivation

A metaobject protocol may be derived from one or more other metaobject protocols using a simple syntax (see examples in figure 3.9). Protocol derivation supports the application of multiple independent behavioural changes to a single base-level class or object. A derived protocol represents the aggregated behavioural changes of its parent protocols, and associating a derived protocol with a base-level entity applies all those changes to it.

Protocol derivation is not inheritance. It is delegation from each child protocol to its parents, and is intended to support composition and reuse of behavioural changes. It is not intended to support reuse of protocols by extension at design time, in the way that inheritance supports reuse of classes [Joh88] [Bie95] [Con00]. Each protocol represents some coherent and usually independent behavioural change (implemented by a set of metaobject classes). Delegation supports composition of those behavioural changes without requiring an inheritance relationship between the delegator and delegatee [Vie98], supports more flexible composition of behaviour from different sources than can be supported by inheritance [Lie86], and improves encapsulation and reusability of the composed behaviours [Gam95].

```
protocol SecurityProtocol {
   reify Invocation: CheckInvocation;
   reify StateRead: CheckRead;
   reify Creation: CheckCreation;
}


protocol LicenceProtocol {
   reify Invocation: LicenceCheck;
}


protocol DerivedProtocol : LicenceProtocol, SecurityProtocol {
   reify Invocation: Verbose;
   reify StateWrite: PreventWrite;
}
```

Figure 3.9: Deriving a metaobject protocol.

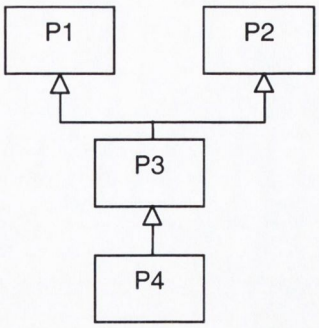


Figure 3.10: A metaobject protocol derivation tree.

When a reified operation occurs it is passed first to the protocol directly associated with the entity in which it occurs[6]; that protocol carries out the behaviour implemented by its appropriate metaobjects and then delegates the operation to its parent protocol(s). In the protocol derivation tree[7] illustrated in figure 3.10 P4 is derived from P3 which is in turn derived from P1 and P2. If P4 is associated with a base-level entity, then reified operations in that entity will be passed first to P4, then delegated to P3, then to P1 and finally to P2.

Where there are multiple parent protocols a left-right preorder depth-first strategy is used, in which the operation is delegated to parent protocols (from left to right) before it is delegated to sibling protocols. This strategy ensures that each derived protocol is treated as a unit with its parent protocols. The alternative, a breadth-first strategy in which siblings are visited before parents, would delegate operations to protocols in different derivation branches before completing delegation along any branch. A depth-first strategy means that the behavioural changes represented by a derived protocol are executed in a predictable fixed order even when that derived protocol is itself a parent protocol for a further derived protocol. While protocols are intended to represent independent behavioural changes the order in which they are executed can be important. A left-right depth-first delegation strategy allows the meta-level programmer to determine execution order absolutely by suitable arrangement of the protocol derivation order.

Each protocol contains metaobjects implementing behavioural changes for one or more reification categories. When an operation is delegated to a protocol it is handled by the metaobject for the appropriate category. The delegation of an operation from protocol to parent protocol is in fact implemented as a delegation from metaobject to metaobject, and is the means by which metaobjects are composed with each other to effect their behavioural changes on a single operation. At runtime a derived protocol potentially represents a set of metaobjects for each category of operation, where the metaobjects explicitly declared by the derived protocol are augmented by metaobjects contributed by its parent protocols. The number of metaobjects for each category depends on the number of parent protocols that reify that category and contain metaobjects for it. A derived protocol can be considered to represent an aggregation of the behavioural

[6] Each class or object can have a maximum of one metaobject protocol associated with it at a time (see section 3.10).
[7] The protocol derivation tree is an inversion of the conventional tree structure. Parent protocols would be called *children* in a conventional tree. They are called parents here to describe the derivation relationship.

changes represented by its parent protocols. Protocol derivation is discussed in more detail in the context of metatypes (see section 3.14).

3.8.2 Metaobject composition

Architectures that support application of multiple behavioural changes to a single operation in a program must compose the components representing the changes so that the operation can be passed from one to the other in a structured way. Composition of behavioural change components can be categorised as implicit or explicit. Implicit composition occurs when the behavioural change components are individually associated with a point in a program, and each one is automatically composed with any that are already associated with that point (e.g. MetaXa, Guaraná, PROSE, AspectJ, JAC). Explicit composition [Mul95] occurs when behavioural change components intended to be applied at a single point in a program must be intentionally composed by the meta-level programmer (e.g. Iguana/C++, MetaclassTalk [Bou99]). Implicit composition can be simpler for a programmer to use, because it occurs automatically during association. However, implicit composition also increases the risk that conflicting behavioural changes may be used together on a single operation in a program. A programmer adding a new behavioural change component to a system that uses implicit composition is not necessarily aware of other behavioural change components affecting the same point, and therefore does not have to address the issue of potential conflicts between them.

Iguana/J uses explicit composition of metaobjects implementing behavioural changes. The principal result of protocol derivation is to compose the metaobjects of the protocols into sets that handle each of the categories of operation that may be reified. The protocol derivation structure, and thus the metaobject composition structure, must be explicitly created by the meta-level programmer.

Composition of metaobjects is on a per-category basis. For example, all metaobjects declared to reify invocation in each protocol are composed to form a delegation sequence that handles invocation. The example declarations in figure 3.9 show a protocol called `DerivedProtocol` derived from the two protocols `LicenceProtocol` and `SecurityProtocol`, in which metaobject classes declared in all three protocols contribute to `DerivedProtocol`. Where there is more than one metaobject class for a category of operation they are automatically composed with each other. This happens for the invocation category, so in `DerivedProtocol` method invocation will be handled in turn by each of the three metaobject classes `Verbose`, `LicenceCheck` and `CheckInvocation`. State read will be handled by `CheckRead` alone, creation will be handled by `CheckCreation` alone, and state write by `PreventWrite` alone. The other three reification categories (deletion, send and dispatch) are not reified by any of the three protocols, and therefore the protocols do not contain any metaobjects to handle them.

Delegation from metaobject to metaobject results in a composition model structured as a simple linear chain, where each of the metaobjects for a category is given the opportunity, sequentially, to handle the operation. The chain is created automatically by the delegation mechanism of protocol derivation. The depth-first delegation passes a reified operation in turn to the appropriate metaobject (if there is one) for that category of operation in each protocol. The result is that each operation is presented first to any appropriate metaobject declared explicitly in the derived protocol and then it is delegated up the protocol derivation hierarchy to metaobjects for that category in any parent protocol. Normally each metaobject will carry out

appropriate *before* actions, then delegate to the next metaobject (using an inherited `proceed()` method), and when that call returns carry out any appropriate *after* actions. Giving the metaobject control over the delegation supports behavioural changes structured as wrappers, where the operation must be surrounded by related elements of the change. This is proposed by Tarr et al. [Tar00] as a requirement for behavioural change components. For example, it supports application of synchronisation to an operation by simply enclosing the delegation statement in a Java `synchronized` block. A similar approach, with explicit passing of the operation, is used by MetaXa and by Iguana/C++.

Alternative approaches involving separate 'before' and 'after' handlers, with no responsibility for carrying out the adapted operation, are used by PROSE and Guaraná. In both of these the adapted operation is carried out by an internal function after all the 'before' handlers return. This makes the use of changes that are naturally structured as wrappers difficult to apply. In Guaraná each 'before' handler is responsible for passing the operation to the next metaobject. This provides flexibility at runtime in how composed behavioural changes interact. In PROSE passing of the operation to the next aspect is carried out by an internal dispatching function. While this approach avoids any possibility that passing of the operation to the next behavioural change component will be inadvertantly omitted by a meta-level programmer it removes any ability for the programmer to adjust the composition behaviour at runtime.

3.8.3 Truncation of the composition chain

The composition chain in Iguana/J is quite fragile, because each metaobject is responsible for passing the operation to the next one in the chain. Typically the overall result of composition is that the 'before' actions of each metaobject are carried out in turn, moving up the chain, then the operation is carried out, and then the 'after' actions of each metaobject are carried out, moving back down the chain. However, a metaobject can alter this behaviour by choosing not to pass the operation any further along the composition chain. Figure 3.11 illustrates this using a metaobject class that decides whether a method invocation on a base-level class is allowed to proceed or not. If it may proceed the metaobject uses `proceed()` to pass the operation to the next metaobject in the composition chain. If the operation is disallowed the metaobject returns immediately, thereby preventing the operation being passed further along the chain. By giving each metaobject the responsibility for passing the operation to the next one in the chain the architecture supports the use of metaobjects that may want to avoid or redirect the original operation. However it also means that a malicious or badly designed metaobject could disrupt a composition chain. The responsibility for ensuring that this does not happen unintentionally rests with the meta-level programmer.

```
public class CheckInvocation extends MExecute {
     public Object execute(Object target, Object[] args, Method m) {
        if(allowed(m.getName(), args)) {
          return(proceed(target, args, m));
        }
        return(null);
     }
     private boolean allowed(String s, Object[] a) { . . . }
}
```

Figure 3.11: Intentional truncation of the composition chain.

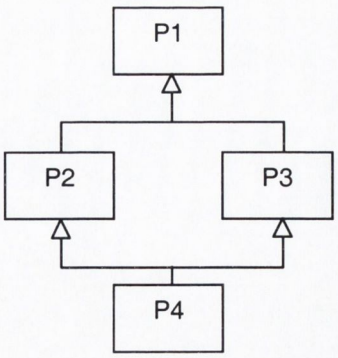


Figure 3.12: Multiple path protocol derivation.

Disruption of the composition chain is an issue in any architecture that supports composition of behavioural change components and places control of passing an operation to the next component in the composition structure with each component (e.g. MetaXa). It occurs even in some architectures that use separate 'before' and 'after' handlers, if each 'before' handler is responsible for passing the operation to the next 'before' handler (e.g. Guaraná). Only architectures that provide no programmer control over passing the operation between components in the composition structure avoid this issue completely (e.g. PROSE).

3.8.4 Metaobject compatibility

There is a general issue of how to ensure that composed metaobjects, provided by different protocol declarations, do not implement conflicting behaviour. For example, if a protocol that redirects a read operation to a remote process is used with a protocol that redirects to a persistant local data store then obviously only one of them can operate correctly. This compatibility problem can occur in any reflective architecture that supports automatic composition of metaobjects, and has been widely discussed in the literature (e.g. [Mul95] [Bou98] [Paw99] [Bou99] [Tar00] [Ber03]).

As discussed in section 3.8.2, metaobject compatibility tends to be more of a problem in architectures that use implicit composition. Implicit composition allows metaobjects to be used together without requiring the meta-level programmer to be aware of the combinations in use at different points in a program. Iguana/J supports only explicit composition of metaobjects (by protocol derivation). It does not directly address the potential problem of metaobject compatibility, but relies on the meta-level programmer to detect conflicts during the design of protocol derivation sequences.

3.8.5 Multiple instances of a metaobject class

Because Iguana/J supports derivation from multiple metaobject protocols it is possible for a metaobject protocol to be included in a derived protocol along more than one path. In the example illustrated in figure 3.12 metaobject protocol `P4` is derived from both `P2` and `P3`, and each of `P2` and `P3` is derived from `P1`.

Thus P1 is included in P4 along two separate paths, and the model must address the issue of how the metaobjects of P1 are included in the metaobject composition of P4. The solution adopted is to ignore duplicate derivation paths, thus ensuring that the metaobjects of any protocol are only ever included once in a derived protocol. In the example of figure 3.12 this means that the metaobjects declared by protocol P1 will be included just once in the metaobject composition chains of P4. Taking this together with the depth-first traversal of the protocol derivation tree for ordering the metaobject composition chain, it is apparent that the order of metaobjects on the composition chains of P4 will contain (for each reification category) metaobjects declared in P4, then P2, then P1, and then P3.

The decision to include the metaobjects of a protocol only once when the protocol is reachable by multiple derivation paths was taken based on the assumption that the behavioural change represented by a protocol is required only once in any derived protocol. A consideration of various possible uses of metaobject protocols led to the belief that situations where the behavioural change represented by a protocol reachable by multiple paths would be required multiple times in the derived protocol are likely to be rare (see section 3.14.3 for further discussion of this point).

While Iguana/J ensures that the metaobject classes of one protocol will only be included once in a derived protocol, no matter how many paths the protocol may be reachable by, it does not impose any restriction on the inclusion of multiple instances of the same metaobject class when it is declared in different protocols. One of the ways in which this situation can arise is when a derived protocol and its parent protocol declare the same metaobject class to reify a category. In this case the composition for that category in the derived protocol will contain two instances of that metaobject class. The decision to allow duplicate metaobjects in a composition chain under these circumstances was based on the observation that although the same metaobject class is used it is in two different protocols and therefore the two uses of the class are intended to be part of two different behavioural changes. The alternative approach, not to allow duplicate metaobjects from different protocols, would be likely to damage the coherent behaviour of the set of metaobjects within one or both of the protocols.

Of the dynamic adaptation architectures reviewed in chapter 2 of this thesis only Iguana/C++ addresses the issue of how to treat multiple instances of the same behavioural change component in a composition structure. Iguana/C++ applies a general rule that includes any metaobject class a maximum of once in a metaobject composition structure no matter how many times it occurs in the protocols in the derivation tree. It does not differentiate between metaobjects contributed by a single protocol reachable by multiple derivation paths, and multiple uses of the same metaobject class within different protocols.

3.8.6 Partially dynamic derivation

Derivation of metaobject protocols, and therefore composition of metaobjects, is partially static and partially dynamic. While a protocol contains static references to its parent protocols, the examination of the parent protocols and the creation of resulting metaobject composition chains takes place at runtime. The static references to parent protocols (if any) include only their names and the number and types of parameters expected by them (protocol parameters are discussed in section 3.12). This means that a derived protocol has a compiled-in dependency on the signatures of its parent protocols, but not on any other details of its parents.

At runtime, when a derived protocol is first used, its parent protocol structure is traversed to collect information such as declared reification categories and metaobject classes, so that the metaobject composition chains can be created for the derived protocol. Because derived protocols have no compiled-in dependency on the details of their parent protocols, a parent protocol can be altered in any way (other than changing its signature) without recompiling the derived protocol. For example, in the protocol derivation structure represented in figure 3.12 the declarations for protocols P1, P2 and P3 may be altered without recompiling P4. This includes adding reification categories, changing metaobject classes useds, and changing parent protocols of P1, P2 or P3. The metaobjects for each reification category for P4 are composed dynamically, when P4 is first used, by visiting P3, P2 and P1 to examine their declarations.

This partially dynamic derivation and composition technique is a compromise. It allows flexibility in the design and modification of related protocols, while still taking advantage of optimisations in the structure and code of protocol classes that static knowledge of parents' names allows. Fully dynamic protocol derivation, supporting construction of protocol derivation trees at runtime, was not explored as it was beyond the immediate scope of this research.

3.8.7 One metaobject per category

Within the declaration of a single metaobject protocol each reification category can have only one metaobject class assigned to it. This means that it is not possible to create a composition of multiple metaobjects for one category within a single protocol declaration, and therefore the only way to compose metaobjects is by protocol derivation. The justification for this is that each protocol is intended to represent a simple coherent behavioural change. The protocol structure allows metaobject classes implementing change for different categories of operation to be combined under a single name, and therefore it supports independent implementation and use of behavioural changes affecting different kinds of operation. The protocol structure does not allow more than one metaobject class to be used for any one category of operation, and therefore it prevents combining metaobject classes for one category within one protocol. This is intended to encourage the use of protocols to represent single simple behavioural changes, and the use of protocol derivation to combine them.

3.9 The null protocol

Iguana/J introduces the concept of a *null protocol* to make the protocol derivation and protocol association structures more consistent. The null protocol is an empty protocol that reifies nothing and is not derived from any other protocol, and therefore uses no metaobjects. The Iguana/J class library includes a class called `NullProtocol` to represent the null protocol. The protocol's declaration can be written as:

```
protocol NullProtocol {
}
```

Metaobject protocols that are not explicitly derived from another protocol are implicitly derived from the null protocol, and therefore the null protocol is at the top of the protocol derivation tree. All protocols are

ultimately derived from the null protocol. The derivation relationship between the null protocol and other protocols is structurally similar to the inheritance relationship between the Java class `java.lang.Object` and other classes.

The null protocol represents zero behavioural change. It is useful to consider it as containing metaobjects that implement the original unchanged behaviour for each of the seven categories of operation that may be reified. For example, for the invocation category it invokes the target method, passing the arguments and returning the result. As the null protocol is the ultimate parent of all protocols all reified operations are ultimately delegated to the null protocol (unless one of the metaobjects truncates the composition chain), and therefore all operations are ultimately carried out. The arguments or target of the operation may be altered as it passes through each of the metaobjects of the protocols, so each metaobject has the opportunity to affect the operation before it is delegated to the null protocol. Similarly, the operations's return value is passed back down the delegation chain from the null protocol, giving each metaobject the opportunity to examine or alter it if required.

Creating a null protocol that implements the default behaviour gives consistency to the metaobject delegation process. Because every metaobject composition chain has (conceptually at least) a metaobject at the end that implements the default behaviour there is no need for metaobjects to behave differently depending on whether or not there is another metaobject to delegate to. In MetaXa, for example, a metaobject must choose between `doExecute()` to delegate or `continueExecution()` to carry out the operation. In Iguana/J there is always another metaobject to delegate to, because the null protocol is always at the end of the delegation chain. All metaobjects use the `proceed()` method, regardless of their position in the composition chain.

Conceptually, all base-level classes and objects not associated with some other protocol are associated with the null protocol by default. As the null protocol represents zero behavioural change and simply implements the default behaviour for each category of operation, association of a base-level class or object with it has no effect. This provides further consistency to the model because all classes and objects are associated with a protocol, either null or non-null.

The null protocol also provides support for the metatype structure discussed in more detail in section 3.14. The metatype structure imposes rules on the relationships between protocols associated with classes and subclasses. A null protocol that is the ultimate parent of all protocols and is associated with all classes and objects by default provides a consistent base for the metatype rules.

Because the null protocol always has no metaobjects and no state, a single instance of `NullProtocol` can be shared across the whole system and used as the meta-level interface object for any class or object associated with the null protocol. A request (via `Meta.getMeta()`) for a reference to the meta-level interface object for any of these classes or objects will return a reference to this single instance of `NullProtocol`. Sharing a single instance like this saves the creation of many instances because the common case is likely to be that many of the classes and objects in a system are associated with the null protocol,

3.10 Association

Association of a metaobject protocol with a base-level class or object is the means by which the new behaviour represented by the protocol is applied to a class or object. Association of a protocol with a class affects all instances of the class, while association with a single object affects only that object. Association causes the appropriate operations in the class or object to be reified and redirected to the metaobjects for handling.

A base-level class or object may be associated with just one metaobject protocol at any one time. This restriction forces the meta-level programmer to carry out composition explicitly (by deriving protocols), and therefore makes it more likely that metaobject compatibility issues will be detected at design time. Supporting association of multiple protocols with individual base-level classes and objects would mean that arbitrary protocols would have to be composed automatically at runtime, during association. This would leave the system more open to inadvertant use of conflicting protocols together.

Association of metaobject protocols with classes and with individual objects are discussed separately in the following two subsections. The metatype structure imposes certain restrictions on which protocols may be associated with related classes and objects, but these are not discussed here. They are dealt with in detail in section 3.14.

3.10.1 Association with classes

A dynamic adaptatation architecture can initiate association of behavioural change components with a class or object in a program using one of three general techniques. The first is dynamic initiation, caused by execution of an appropriate statement. For example MetaXa and Guaraná both provide methods that can be used to associate a specified class or object with a specified metaobject. This technique can be used to apply arbitrary behavioural change at any time while the program is running, but the requirement that a statement be executed usually means that an invasive change must be made to the program to insert it. The second technique is static initiation, using external static declarations of the associations between behavioural change components and program classes. The declared associations take effect when the system starts up, or as each program class is about to be used. For example, Kava [Wel01] uses a separate configuration file to declare associations. This technique allows association to be initiated without invasive changes to the program, because no specific executable statement is required. However it can only be used for behavioural changes that are known at system start up. The third technique is remote initiation. A service embedded in the runtime support system accepts association commands over some communications link, and dynamically initiates them. PROSE supports this technique, based on remote communication with the JVM Debug Interface (JVMDI). The technique combines some characteristics of the other two techniques by avoiding invasive changes to the program while still supporting arbitrary associations during runtime.

Iguana/J supports dynamic and static initiation of association between a base-level class and a metaobject protocol. Iguana/J does not support remote initiation of association, although it could be implemented as a metaobject protocol. This is discussed further as part of the evaluation, in section 5.5.2. Whichever technique is used to initiate an association in Iguana/J, the association itself is dynamic. That is, the

association takes place while the system is running and may be undone or replaced by an association with a different metaobject protocol. It is only the initiation of the association that may be either dynamic or static.

3.10.2 Dynamic initiation of association

Dynamic initiation supports association of a class and a protocol when the system is already running. It is carried out by using the static method `Meta.associate()`, which takes the fully qualified names of both the base-level class and the protocol as strings. For example, the following Java statement causes the base-level class `com.foo.SomeClass` to be associated with the metaobject protocol `com.prot.SimpleProtocol`:

```
Meta.associate("com.foo.SomeClass", "com.prot.SimpleProtocol");
```

The `associate()` method may be called from any code that knows the names of the base-level class and the protocol.

Two alternatives to this technique for initiating associations dynamically were considered. The first was to modify the `Object` class to contain a public static `associate()` method that would take a protocol name as its single argument, and would be inherited into every Java class. The second was similar but at the other end of the association link; that is, to provide a public static `associate()` method in every protocol class, taking a class name as its only argument. Both of these approaches were rejected for the same reason. They both require a compile time reference to one of the elements to be associated. In the first case the reference is to the base-level class, and in the second case it's to the protocol class. Both of these approaches would mean that a reference to one of the elements to be associated is required (although a class object can be acquired dynamically using `Class.forName()`), and therefore both would increase the dependence between the associating code and either the base-level code or the meta-level code. By providing a method that only requires the names of the base-level class and the protocol as strings, the association operation is made more independent and more flexible. It allows an association to be made where both the base-level class and protocol names are specified as strings at runtime, possibly via some dynamic configuration mechanism.

In previous Iguana architectures [Sch01] association of a base-level class with a metaobject protocol meant only that instances of that base-level class should have the appropriate operations reified and handled by the metaobjects of the protocol. These architectures did not support reification of static operations. Iguana/J addresses this limitation by extending the semantics of association of a class with a protocol to include reification of operations not just on all instances of the class but also on the class itself. In other words, when a class is associated with a protocol it has two separate but related meanings: that instances of the class should have that protocol as their initial protocol, and that the protocol should be used to reify static operations on the class. In practice this means that the class, just like each of its instances, has its own set of metaobjects to handle reified operations. Each instance of the class will be associated with the same protocol (initially at least), and therefore similar reified operations on the class and on an instance of the class will be handled in the same way. For example, if the protocol reifies invocation then the same metaobject class will be used to handle both static and non-static method invocations on the class and all its instances. Some

reified operations are not effective for both meanings of association of a protocol with a class. For example, reifying creation in a protocol has an effect on the class (as each new instance is created), but then the new instance acquires its own copy of the same protocol in which the metaobject reifying creation has no effect.

Association of a base-level class with a protocol affects all subclasses of the named class. If the named base-level class is not yet loaded then the dynamically initiated associate operation will cause it to be loaded. The associate operation does not cause subclasses to be loaded. Subclasses that are already loaded are affected immediately, and subclasses that are loaded subsequently are affected as they are loaded.

3.10.3 Static initiation of association

The static technique for initiating an association between a named class and a named metaobject protocol is by specifying the association before the system starts. The association is specified independently of the behavioural change component and of the progarm to be adapted. This independent specification of association is the 'ternary' representation identified by Brichau et al. [Bri01]. Examples of this approach include Kava and Weave.NET [Laf03] which both use an XML file to contain declarations of associations. For both, the file allows association (or weaving) details to be specified, while still avoiding invasive changes and maintaining a strong separation of concerns. By maintaining the separation and providing an independent entity (the XML file) to specify association they reenforce the concept of three separate development roles – base-level programmer, meta-level programmer, and system integrator. There have been other proposals for independent specification of association. For example, Robbens proposes *non-functional policies* to specify associations separately from base-level and meta-level [Rob99], but they are implemented as objects at runtime and therefore invasive changes are required to create them. David proposes a structure of adaptation policies and system policies to specify weaving separately from the aspects and program to be woven [Dav02].

Iguana/J supports independent specification of association using a special file, called the Iguana/J configuration file, that is read by the Iguana/J core library at startup. Each line in the configuration file declares an association between a named base-level class and a named metaobject protocol. These static declarations of association do not cause the base-level class or any of the classes involved in the protocol to be loaded when the declaration is read at system initialisation time. The association with the protocol is only created when the specified base-level class (or any of its subclasses) is loaded in the normal course of system execution. The association established as a result of a static declaration in the Iguana/J configuration file is semantically the same as an association established as a result of a call to `Meta.associate()`.

For example, the following declaration in the configuration file will cause the base-level class `com.foo.SomeClass` to be associated with the metaobject protocol `com.prot.SimpleProtocol`:

```
com.foo.SomeClass ==> com.prot.SimpleProtocol;
```

Static declarations of association support a strong separation of concerns between the progam and the metaobject protocols. They avoid a need to modify the program to insert `Meta.associate()` calls, and therefore they allow the association to be initiated without invasive changes to the program. Static initiation

of association supports the application of behavioural change to third party programs for which no source code is available.

3.10.4 Wildcards

When specifying which locations in a program behavioural change components should be associated with it is sometimes useful to be able to identify a set of locations with a single expression. For example, it might be necessary to associate the new behaviour with all the classes in a package. Wildcard characters in name specifiers can support this. Wildcard characters in association or weaving expressions are supported by, for example, Aspectual Components [Lie99], Kava [Wel01], Hyper/J [Tar00a], AspectJ [Kic01] and Weave.NET [Laf03]. Of the three architectures supporting unanticipated dynamic adaptation that are reviewed in sections 2.5, 2.6 and 2.7 of this thesis, only PROSE supports a wildcard concept. The other two, MetaXa and Guaraná, require that association expressions fully identify individual classes or objects in the program.

Iguana/J supports a limited form of wildcard in the base-level class name used in initiating an association. By using '*' as the simple class name the association is applied to all members of the specified base-level package. For example, the following Java statement causes all the classes in the base-level package `com.foo` to be associated with the metaobject protocol `com.prot.SimpleProtocol`:

```
Meta.associate("com.foo.*", "com.prot.SimpleProtocol");
```

This wildcard feature may be used for both association by a call to `Meta.associate()` and association by a declaration in the Iguana/J configuration file. It supports package-wide association, and is an attempt to simplify the work of the meta-level programmer in associating a related set of base-level classes with a common protocol. As package-wide association is intended to be a shorthand for individual associations with every class of the package it has a similar effect as individual associations with every class of the package. The only difference is that package-wide association does not cause any base-level classes to be loaded, whereas an individual association with a class causes that class to be loaded.

3.10.5 Association with objects

Dynamic adaptation architectures may support application of behavioural change to classes or to individual objects, or both. For example, JAC [Paw99], Reflex [Tan01] and PROSE support changes to classes only, CodA [Mca95] supports changes to individual objects only, and Iguana/C++, MetaXa, and Guaraná support changes to both classes and individual objects. Support for adapting individual objects provides a wider scope for applying behavioural change. An object can be adapted for a particular context without affecting other instances of the same class. On the other hand class adaptation affects all instances equally, and therefore is simpler to use when the same change is to be applied to each instance of a class.

Iguana/J supports association of behavioural change with both classes and individual objects. One of the effects of associating a class with a metaobject protocol is to set that protocol as the initial one for new

instances of the class, so each new instance automatically begins life with the set of behavioural changes represented by its class's protocol. There are two ways that the protocol associated with an existing object can be changed. The first is by changing the protocol associated with the object's class. This not only changes the class's protocol but also that of all existing instances of the class. The change also propagates down to all existing instances of all subclasses.

The second way to change the protocol associated with an object is to initiate it dynamically, using an overloaded version of the static method `Meta.associate()`. It is not possible to support static declarations of associations for objects because an object has no statically known unique identifier in the way that a class has a statically known unique name. The following example associates a single identified instance of the class `TestClass` with the protocol `com.prot.SimpleProtocol`:

```
TestClass obj = new TestClass();
Meta.associate(obj, "com.prot.SimpleProtocol");
```

Association of an object affects only that object, without affecting the object's class or other instances of the class. The metaobjects of the specified metaobject protocol will be used to handle reified operations on the associated base-level object, but because the protocol is not associated with the object's class they will not be used to handle static operations or object creation. Reifying object creation in a protocol that will only be associated with individual objects is meaningless. The protocol will not be used to reify creation of that object because the object is already created when the protocol becomes associated with it. Reifying creation is only meaningful in a protocol that will be associated with a class.

A side effect of overloading `Meta.associate()` so that it accepts either a reference to a base-level object or the name of a base-level class is that it is not possible to associate a single instance of `java.lang.String` with a protocol. An attempt to do so will result in a misinterpretation of the instance of `String` as the name of a class. An alternative that would avoid this limitation is to change the `associate()` method from accepting the name of a class to accepting a reference to a `Class` object. `Class.forName()` can be used to get a reference to any named `Class` object at runtime, so that could be used to convert a name to a reference before calling `associate()`. The drawbacks of using this alternative are that it puts the onus on the programmer to get a reference to the `Class` object, and, more importantly, it removes the ability to do package-wide association by passing a name argument containing a wildcard character. It was considered that the convenience of package-wide association and of passing class names rather than references was likely to be of more practical use than support for applying behavioural change to the `String` class. In any case, it would not be possible to associate the `String` class with a protocol due to a limitation in the prototype implementation of Iguana/J that affects a small number of core system classes (see section 4.4).

3.11 Shared metaobjects

The concept of supporting a mixture of meta-level components with different scopes was first proposed by Matsuoka in ABCL/R2 [Mat91]. The architecture supports 'individual-based' metaobjects and 'group-wide'

metaobjects. The group-wide metaobjects are intended to support a global view of computation and coordination among objects. Metaobjects with varying scope were later included in Iguana [Gow97], and more recently local and shared scope was supported in Iguana/C++ [Sch01]. Iguana/J supports both *local* and *shared* scope for metaobjects, in order to support both object-specific and coordinated behavioural change. In this thesis the treatment of shared metaobjects is explored and clarified, particularly in composition with other metaobjects.

An Iguana/J protocol declaration can specify the scope of each metaobject used to reify a category of operation. Each metaobject can be made local or shared by placing the keywords `local:` or `shared:` at appropriate points between the reify statements. A local metaobject is one that uses one metaobject instance for each base-level class or object with which the declaring protocol is associated. Consider the following example protocol declaration:

```
protocol LicenceProtocol {
  local:
  reify Invocation: LicenceCheck;
}
```

When a scope keyword is encountered in a declaration all metaobjects declared from that point on are of that scope until another scope keyword is encountered. `local:` is redundant in this example declaration, because local is the default scope, but it is included for clarity. It means that the scope of the `LicenceCheck` metaobject class is local, and therefore that there will be one instance of it for each base-level class or object with which the protocol is associated. If the protocol is associated with a class that has 30 instances then there will be 31 instances of `LicenceCheck`, each one handling method invocation in a single object or in the class itself.

A shared metaobject is one that uses a single metaobject instance shared between each base-level class and its instances. Consider the following protocol declaration:

```
protocol LicenceProtocol {
  shared:
  reify Invocation: LicenceCheck;
}
```

In this example the scope of the `LicenceCheck` metaobject has been changed to shared, and therefore a single instance of it will be shared by a base-level class and its instances within a single virtual machine. If this protocol is associated with a base-level class that has 30 instances then there will be only one instance of `LicenceCheck`, used to handle all static and non-static method invocations on the base-level class or any of its instances. The scope of the shared metaobject is just that base-level class and its instances. If the same protocol is associated with a second base-level class then a second instance of `LicenceCheck` will be created and shared between the second class and all its instances.

Whether a metaobject should be local or shared depends on whether or not it contains state relating to a single base-level object. If it does then the metaobject should be declared as local. If it contains no state, or if its state relates to all instances of a base-level class, then the metaobject should be declared as shared.

When one protocol is derived from another the scope of each inherited metaobject is retained. One of the results of this is that a metaobject composition chain, created by deriving protocols from each other, can contain a mixture of shared and local metaobjects. A local metaobject appears on just one composition chain; a shared metaobject appears on multiple composition chains, all reifying the same category of operation but for different instances of the same base-level class.

A meta-level interface object (an instance of the appropriate protocol class, see figure 3.8) is never shared, except for the null protocol. Every class or object associated with a non-null protocol has its own meta-level interface object, containing references to the metaobjects making up the protocol. If any of those metaobjects are shared then there will be multiple references to them from the separate meta-level interface objects of the base-level class and base-level instances associated with the protocol.

3.12 Protocol parameters

To support the design of reusable behavioural change components some technique is required for specialising the components when they are instantiated for association with a class or object [Tar00] [Spe02]. Specialisation allows a generalised behavioural change component to be given information, such as names of classes or class members, to make its operation specific for the class or object with which it is associated. The most common way to support specialisation is via arguments to metaobject (or aspect) class constructors. This technique is used, for example, by MetaXa, Guaraná, PROSE, Kava, MetaclassTalk, and Reflex. In general it can be used by any architecture in which the behavioural change components are instantiated from a class. Some element of metaobject specialisation is supported by Gowing's Iguana [Gow97] via metaobject constructor arguments, but it is not integrated into protocol declarations. For example, it is not possible to pass specialisation values to the metaobjects as part of associating the protocols with a class or object.

In Iguana/J, metaobject protocol declarations are extended to include parameterisation. Iguana/J supports specialisation of metaobjects via protocol parameters and metaobject constructor arguments. Parameter values are supplied when an association is initiated and are passed to the protocol's metaobject constructors to be used to adapt the metaobjects to the particular association. For example, the names of methods or fields in the base-level class might be passed via protocol parameters so that the metaobject can handle operations on them in an appropriate way. This allows the metaobjects of a protocol (and the protocol itself) to be designed to be more generic and reusable, with parameter values providing specialisation information at the time of association with a class or object.

Protocol parameters must be included in the protocol declaration, and the declaration must distribute the parameters to the appropriate metaobject classes. The parameters distributed to the metaobject classes must match the signatures of their constructors. If they do not then an error will be reported by the protocol compiler.

```
protocol AccessProt(String mName, String fName) {
   reify Invocation: CheckInvoke(mName);
   reify StateWrite: CheckWrite(fName);
}

class CheckInvoke extends MExecute {
   private String s;
   public CheckInvoke(String s) {
      this.s = s;
   }
   public Object execute(Object t, Object[] args, Method m) {
      if(m.getName().equals(s)) {
         // ... Check security permissions.
      }
      return(proceed(t, args, m));
   }
}

class CheckWrite extends MStateWrite {
   private String s;
   public CheckWrite(String s) {
      this.s = s;
   }
   public void write(Object t, Object arg, Field f) {
      if(f.getName().equals(s)) {
         // ... Check security permissions.
      }
      proceed(t, arg, f);
   }
}
```

Figure 3.13: A protocol and two metaobject classes that expect parameter values.

Consider the example in figure 3.13. The protocol accepts two parameters and passes them to the constructors of the two metaobject classes. Values for these parameters are supplied when an association of the protocol with a class or object is initiated. The metaobject classes are designed to be independent of the names of the method and field that they will operate on, and therefore to be reusable in different contexts.

If the association is initiated by a static declaration in the Iguana/J configuration file then the values are supplied by including them in the declaration. For example:

```
com.foo.SomeClass ==> AccessProt("getDetails", "total");
```

If the protocol association is initiated dynamically then the parameter values must be marshalled in an array of objects for passing to the `associate()` method. The following example is the equivalent of the static declaration of association above:

```
Object params[] = new Object[2];
params[0] = "getDetails";
params[1] = "total";
Meta.associate("com.foo.SomeClass", "AccessProt", params);
```

In both dynamically and statically initiated association the types of the supplied parameter values must match the types declared by the protocol. In both cases it is not possible to perform static type checking. For static

declarations in the configuration file it is not possible because the file is not compiled or processed before runtime. For dynamically initiated associations it is not possible to perform compile time checks, because the protocol is referred to only as a name string. Type checking must be performed at runtime when an attempt is made to create the association. Any alternative technique that allows the Java compiler to check the types of parameters supplied to the protocol would have to include an embedded reference to the protocol class, and that would weaken the separation between the meta-level components and the base-level program. The lack of static type checking is an inevitable price to be paid for a strong separation of concerns in an architecture supporting dynamic adaptation [Tar00]. In MetaXa and Guaraná, arguments passed to metaobject constructors are type checked by the Java compiler in the normal way, but those architectures do not support creation and association of metaobjects without invasive insertion (and compilation) of association statements. In PROSE, arguments to constructors of crosscut classes are type checked if weaving is initiated locally, by statements embedded in the program. However, if the ability to initiate weaving remotely is used then the issue of type checking does not arise because the current version of PROSE does not support passing of arguments over the remote aspect insertion interface. Even if remote weave-time arguments were supported they could not be statically type checked, almost by definition.

If Iguana/J encounters a parameter type mismatch at runtime an exception is thrown in the case of dynamic initiation, and an error in the case of static initiation. Type mismatch in a static declaration of association is dealt with by throwing an error because it is a static configuration problem that could result in incorrect system startup.

There is one important difference between using parameters in a static declaration of association and using them with a call to `Meta.associate()`. A static declaration of association is not Java code and is not executed; it is read and parsed by the Iguana/J runtime system, and therefore any parameter values given must be interpreted by Iguana/J and converted to appropriate Java runtime entities. The current implementation supports interpretation of literal parameter values such as numbers, quoted strings, or the keywords 'true' or 'false'. Although metaobjects that require types other than primitive or `java.lang.String` as arguments to their constructors can be created, a protocol that uses such a metaobject cannot be used in a static declaration of association.

```
protocol SyncProt(String m1,String m2,String f) : AccessProt(m2, f) {
  reify Invocation: Sync(m1);
}

class Sync extends MExecute {
  private String s;
  public Sync(String s) {
    this.s = s;
  }
  public Object execute(Object t, Object[] args, Method m) {
    // ... Uses this.s
  }
}
```

Figure 3.14: A derived protocol `SyncProt`, and its metaobject, that passes parameter values to its parent.

Protocol parameters can be passed up the protocol derivation tree to the metaobjects declared in parent protocols, as illustrated in figure 3.14. In this example, two of the three parameters expected by `SyncProt` are passed to its parent protocol, `AccessProt`.

3.13 Dynamic association

In any architecutre that supports dynamic association of behavioural change components with a program an issue arises as to how such an association should affect any change component already associated with the same part of the program. The two principal possibilities are that the new behavioural change should be integrated, or composed, with the existing behavioural change, or that it should replace the existing behavioural change. MetaXa, Guaraná, and PROSE all compose the new behavioural change with any existing one at an affected point. In MetaXa additional metaobjects are simply added to the linear metaobject composition chain. In Guaraná, the existing metaobjects are requested to incorporate the new metaobject into the compostion structure. In PROSE, the new aspect is added to the list of aspects to be notified when the join point operation is executed.

In Iguana/J a protocol dynamically associated with a base-level class or object completely replaces any protocol already associated with it. This choice is based on the concept of a metaobject protocol as an already integrated set of behavioural changes. By replacing any existing protocol Iguana/J avoids implicit composition of metaobjects, and forces explicit composition, as discussed in section 3.8.2. The reification categories and metaobjects of the new protocol take the place of the existing ones, and new behavioural changes take effect. However, a number of questions arise about how metaobject composition, shared metaobjects and protocol parameters interact with the protocol replacement operation.

When a protocol is replaced it is possible that the new protocol shares one or more parent protocols with the previously associated protocol. The new protocol may even be derived from the old one (or vice versa). In any of these cases the effect is that there are one or more contributing protocols in common between the old configuration and the new one. If any of the metaobjects of a protocol in common hold state values then an association operation that just discards all the old metaobjects and installs new ones will cause a loss of state information. Even though a metaobject implementing the same behavioural change is present before the association and after it, that metaobject will be a new one without the state values that were held by the old one. The solution adopted for this is that the metaobjects of any protocols in common between the old and the new configuration are retained when a dynamic association operation occurs. This ensures that if, for example, additional new behaviour is to be applied to a class by replacing its existing protocol with one derived from its existing one then any state information held by an existing metaobject will survive the new association operation.

One of the effects of this retention of common contributing protocols is that under some circumstances protocol parameter values supplied with the new association operation may be ignored. Metaobjects that are retained from the old configuration are not created as part of the new association, and therefore any parameter values intended to be used to initialise them have no effect. This is a basic conflict between the

reusability of metaobjects, supported by protocol parameters, and the convenience of retaining state information.

There is a similar issue involving parameter values and shared metaobjects when an existing object is associated dynamically with a protocol. If the shared metaobject already exists for objects of that class then parameter values intended to initialise it will not be used. One way to address these conflicts would be to extend the association operation to allow the programmer to specify whether already existing metaobjects should be retained or replaced. The current research has not investigated the potential usefulness of such an extension.

3.14 Metatypes

The term *metatype* has been used to refer to the type of the metaobject protocol associated with a base-level object [Sch01]. In this thesis the meaning of metatype is adjusted and refined to represent behavioural changes applied to a program, and it is used to impose constraints on how behavioural change may be applied.

3.14.1 The meaning of metatype

Schäfer proposed that each metaobject protocol, derived or not, represents only a single metatype [Sch01]. This thesis proposes that a protocol can represent a set of metatypes rather than just one. In particular, a derived protocol represents the metatypes of each of the protocols from which it is derived. Just as a class can be considered to represent a union of the types of its direct and indirect superclasses, a protocol can be considered to represent a union of the metatypes represented by each of its direct and indirect parent protocols. This proposal is based on the idea that a metatype corresponds to a specific behavioural change for a base-level class or object. Just as a class's type represents a defined public interface, a class's metatype (conferred on it by association with a metaobject protocol) represents a defined change in behaviour. A class associated with a derived protocol has a number of metatypes, representing the separate behavioural changes implemented by each of the protocols in the derivation tree. For the Iguana/J model of dynamic adaptation a metatype is defined as follows:

> The metatype of a class or object represents some coherent internal behavioural change from its original source code behaviour.

This refined concept of a metatype leads to a more structured approach to protocol derivation in which protocols are modular independent entities that are not altered by inclusion in a derivation sequence. Protocol derivation is not inheritance in the conventional subclassing sense of changing and adapting a parent. It is a process of accumulation of metatypes, where each protocol in a derivation sequence delegates to its parent protocols and thus assembles a set of metatypes.

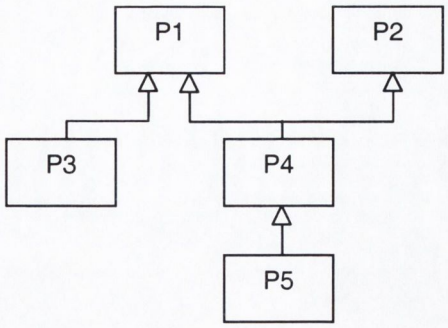


Figure 3.15: Accumulation of metatypes by derived protocols.

The accumulated metatypes are independent of each other. Deriving one protocol from another is intended to support composition, not refinement, of protocols and the metatypes they represent. Refinement of a protocol, if it is required, can be carried out by subclassing the metaobject classes of the protocol and creating a new protocol that uses the subclasses. Mulet et al. [Mul95] suggested similar ideas of 'elementary' composable metaobjects each representing independent single 'customisations'.

In the protocol derivation diagram shown in figure 3.15 each of the protocols P1 to P5 reifies one or more categories of operation and provides replacement implementations for them, thus representing a metatype. Each of the derived protocols P3, P4 and P5 also include the metatypes represented by their parent protocols. A class associated with P5 will acquire the set of independent metatypes represented by P1, P2, P4 and P5, and therefore will exhibit each of the independent behavioural changes defined in P1, P2, P4 and P5.

If a derived protocol represents a simple accumulation or composition of the independent metatypes of its parents then it follows that the behavioural changes that those metatypes represent should not be modified by the derivation process. This is enforced by the delegation technique used to implement protocol derivation. Each derived protocol implements its own behavioural change and delegates operations to its parent protocols, resulting in an accumulation of unmodified behavioural changes, and therefore an accumulation of the metatypes of each of its parent protocols. If the derivation process supported modification of the behavioural change of a parent protocol (by inheritance, for example) then the derived protocol could no longer claim to represent the individual behavioural changes implemented by each parent protocol, and therefore it could no longer claim to represent the metatypes defined by each parent.

3.14.2 Metatypes and composition

The assumption that the declarations within each protocol represent an independent coherent metatype was the basis for a number of decisions, particularly in how some of the more complex metaobject composition situations are treated. An example, as described previously, is that metaobjects of a protocol that is inherited by more than one path are included only once in a derived protocol. On the other hand, if a single metaobject class is used in two different parent protocols then it is included twice in the derived protocol. These

decisions were based on the idea that a particular metatype is not required more than once in the set accumulated by a derived protocol, and that metaobjects used in the implementation of two separate metatypes are independent even when they are instances of the same class.

The decision to allow a maximum of one metaobject class per reification category in a protocol declaration helps to ensure that protocols, and therefore metatypes, are simple and internally coherent. It avoids the temptation to compose metaobjects within a single protocol, thereby avoiding metatypes that do not represent a coherent behavioural change. If the implementation of a behavioural change for one reification category requires the use of a number of utility or delegation classes these can be referred to in the normal way by the metaobject class. If the requirement is to combine two separate behavioural changes then this should be implemented as two separate protocols combined into a third protocol by derivation.

3.14.3 Dependence and independence

The metatype concept proposed by this thesis relies heavily on the idea that protocol derivation is a process of composition of independent entities rather than conventional inheritance. An examination of an example where the concept encounters some difficulties will help to understand the limits of this assumption better.

Figure 3.16 shows a simple protocol that logs each method invocation to a file whose name is specified when the protocol is associated with a class or object. For example, every invocation of a method of the class `Employee` could be logged to a file called employee.log by using the following static declaration of association:

```
Employee ==> LogProt("employee.log");
```

Figure 3.17 shows a protocol derived from `LogProt`, but in such a way that it results in a dependence. The dependence arises because the derived protocol, `SyncProt`, supplies the file name needed by `LogProt` as a specific value rather than passing it through from its own parameters. `SyncProt` uses `LogProt` to carry out logging for it, and therefore the metatypes represented by the two protocols are no longer fully independent. `LogProt` has been included to perform a subsidiary operation for `SyncProt` rather than to provide an independent behavioural change for a base-level class.

The conflict between this style of derivation (called *dependent derivation* in this thesis) and the metatype concept becomes apparent by examining what happens if the derived protocol is used with another that is also derived from `LogProt`. Figure 3.18 shows such a derived protocol, called `AccessProt`. This protocol uses one of its own parameters to pass the file name through to `LogProt` so that the operation of the two protocols remain independent of each other. The conflict appears when `SyncProt` and `AccessProt` are combined, producing multiple path inheritance of `LogProt` as illustrated by the protocol called `Combined` in figure 3.19. Of the derivation paths up to `LogProt` only the first encountered is traversed, and therefore in this example the derivation path from `SyncProt` to `LogProt` is ignored. The result is that when the protocol `Combined` is used, by associating it with a class, no data will be written to 'sync.log', the file specified by `SyncProt`. Log data will be written instead to the file name given as a parameter to `Combined` at the time it is associated.

```
protocol LogProt(String d) {
  reify Invocation: Log(d);
}

class Log extends MExecute {
  private FileWriter o;
  public Sync(String s) {
    o = new FileWriter(s));
  }
  public Object execute(Object t, Object[] args, Method m) {
    o.write(m.getName() + " invoked.\n");
    return(proceed(t, args, m));
  }
}
```

Figure 3.16: A simple logging protocol and its metaobject class.

```
protocol SyncProt() : LogProt("sync.log") {
  reify Invocation: Sync;
}

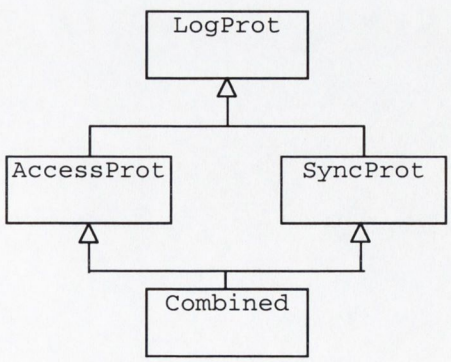
class Sync extends MExecute {
  // ... Implementation of behavioural change.
}
```

Figure 3.17: A dependent derived protocol and its metaobject class.

```
protocol AccessProt(String d) : LogProt(d) {
  reify Invocation: CheckInvoke;
}

class CheckInvoke extends MExecute {
  // ... Implementation of behavioural change.
}
```

Figure 3.18: A non-dependent derived protocol, and its metaobject class.




```
protocol Combined(String d) :
     Accessprot(d),SyncProt {
}
```

Figure 3.19: Composing `AccessProt` and `SyncProt`.

This example illustrates two potential conflicts with the concept of independent metatypes: one with dependent derivation of protocols and one with the use of protocol parameters by a protocol inherited by multiple paths. Dependent derivation of protocols should be avoided as a matter of good metatype programming style. The issue of protocol parameters and multiple path inheritance is more difficult to address but in the absence of dependent derivation it should be easily detected, and therefore avoided, by the meta-level programmer. It could be argued that the same protocol inherited by two separate paths but with different parameter values represents two independent metatypes and therefore that it should be included twice in the composition. However this is to slightly misunderstand the meaning of a metatype. A metatype represents a coherent behavioural change for a class or object regardless of the parameter values supplied to it, and therefore a protocol defines a single identifiable metatype. The metatype represented by the `LogProt` protocol, for example, does not change according to the name of the file it writes to. Two instances of the `LogProt` protocol, associated with different classes and writing to two different log files, still represent the same metatype.

3.14.4 Metatype relationships

As already discussed in section 3.2 constraints are required to provide a structure in which behavioural change can be applied in a controlled way. Metatype relationships provide these constraints, and support rudimentary reasoning about the scope of behavioural change, and how dynamic application of new behavioural changes will affect changes already in place. The restrictions imposed by Iguana/J control the relationships between the metatypes of related classes and objects. The restrictions are that the metatypes of a class must include the metatypes of its superclass, and that the metatypes of an object must include the metatypes of its class. Metatypes are conferred on classes and individual objects by their association with specific metaobject protocols, so the metatype restrictions imply a derivation relationship between the protocols associated with a class and its superclass, and between the protocols associated with a class and its instances. Similar relationships, but expressed in terms of protocol derivation rather than metatypes, were proposed for Iguana/C++.

These restrictions produce a metatype structure that parallels the conventional type structure. Moving down the base-level class inheritance tree, conventional type widens to include the types of each class in the inheritance path. The metatype also widens (or at least cannot narrow) as the base-level inheritance tree is descended because the metatypes of each base-level subclass must include the metatypes of its superclass.

The Iguana/J metatype constraints are similar in many ways to the metaclass constraints introduced in SOM [Dan94] [For98], although SOM does not address unanticipated dynamic adaptation. Both Iguana/J and SOM attempt to ensure that changes applied to a class are retained by subclasses, and both achieve this by imposing constraints on the relationship between behavioural change components associated with a class and those associated with subclasses of the class. In SOM a behavioural change is implemented as a metaclass, and creating a class as an instance of the metaclass gives the class the characteristics implemented by the metaclass. SOM metaclasses support dynamic addition and replacement of class methods and fields, and redirection of method invocation, so a metaclass can implement both structural and behavioural changes for a class. The justification for metaclass constraints in SOM is essentially structural, and is based on the

need to maintain substitutability (polymorphism). Substitutability can be damaged if a class X makes use of a method provided by its metaclass, but Y, a subclass X, has a metaclass that does not provide that method. In this situation an instance of Y cannot be substituted for an instance of X without causing method resolution errors. To solve this SOM imposes a requirement that the metaclass of Y must be the same as or inherit from the metaclass of X. This requirement is similar to the requirement in Iguana/J that a metaobject protocol associated with a class must be equal to or derived from the metaobject protocol associated with its superclass.

Iguana/J does not support structural changes, and therefore the justification for imposing constraints is based not on method resolution requirements for substitutability but on more abstract notions of defining the scope of behavioural change to make the meta-level programmer' s task easier. Iguana/J' s metatype relationship restrictions ensure that behavioural changes, although implemented separately, are integrated into the class inheritance hierarchy. A metatype in Iguana/J represents an internal behavioural change applied to a base-level class. Unlike a SOM metaclass an Iguana/J metatype is not concerned with changes to the structure of a class. The metatype of a class can be considered to be orthogonal to the conventional type of the class, independent of the public interface and the internal structure of the class. An Iguana/J metatype is not, and is not implemented as, a metaclass.

The scope of Iguana/J metatypes, and therefore of metatype constraints, is broader than that of SOM metaclasses. The concept of metatype covers not just behavioural changes to classes, but also behavioural changes to individual instances of classes. Therefore the metatype constraints extend beyond class inheritance relationships to instance-of relationships. There are a number of other important differences between Iguana/J metatypes and SOM metaclasses, particularly differences that affect how the constraints are applied. The dynamic nature of Iguana/J allows the metatype of a class or object to be changed during the life of the class or object. SOM does not support changing the metaclass of a class, and therefore the metaclass constraints need only be applied when the class is being constructed. Thus Iguana/J constraints must take into account that subclasses or instances of a class may already exist, whereas SOM constraints do not.

The Iguana/J metatype structure is given consistency by the introduction of the null protocol. As described in section 3.9 all protocols are ultimately derived from the null protocol, and all classes and objects are associated with the null protocol by default. The null protocol corresponds to a null metatype, and represents no behavioural change. By introducing the concept of a null protocol the metatype structure becomes consistent, allowing the metatype relationships to remain valid throughout the base-level class hierarchy without any exceptions.

The principal motivation for introducing this metatype structure is to ensure that behavioural change introduced by associating a protocol with a base-level class cannot be omitted from subclasses of the class, or from instances of it or its subclasses. This provides a more robust environment for dynamic adaptation, where the base-level class structure cannot be made to appear broken by having a subclass with a narrower behaviour than its superclass or an object with a narrower behaviour than its class.

3.14.5 Metatype rules

We can formalise the metatype relationships described in the previous subsections as a set of four simple rules. The first two establish the basic metatype relationships.

§ 1 The set of metatypes of a class must include the set of metatypes of its superclass.

§ 2 The set of metatypes of an object must include the set of metatypes of its class.

The second two rules deal with the propagation of metatypes, particularly under dynamically changing conditions. These rules help to ensure that §1 and §2 remain valid even when a metatype of a class is changed at runtime.

§ 3 The metatype of a class is inherited by its subclasses.

§ 4 The metatype of a class is propagated to all its existing and future instances.

Dynamic manipulation of metatypes means that it is not possible to perform static checking against these metatype rules. Therefore the runtime system must support comprehensive metatype checking whenever an association of a class with a protocol is attempted. To ensure that the upward metatype relationship is valid a check is made that the new protocol is equal to or derived from the protocol associated with the base-level superclass. Association with a class is propagated down to all subclasses automatically (§3), which ensures that the downward metatype relationship is valid. Association with any class is also propagated to all instances of the class (§4), and that ensures that the object-class metatype relationship is valid for the class. When an attempt is made to associate an individual object with a protocol a check is made that the new protocol is equal to or derived from the protocol associated with the object's class.

The overall effect of these metatyping rules is to ensure that new behaviour is applied to a system in a consistent manner. That is not to claim that a system's overall behavioural integrity is maintained, only that the new behaviour, whatever it may be, is applied consistently across the system. In particular, the rules will ensure that inconsistencies such as a class omitting a behavioural change applied to its superclass, or an object omitting a behavioural change applied to its class, cannot occur under any combination of dynamic association of protocols.

Figure 3.20 illustrates a typical metatype structure, using a hierarchy of three classes named `A`, `B` and `C`. Class `A` is explicitly associated with `P1`, and this association is inherited by its subclass `B`. Thus `A` and `B` have the same metatype. The association would also be inherited by a further subclass, `C`, but it is overridden by an explicit association with protocol `P2`. This is only permissible (by §1) because `P2` is derived from `P1`, and therefore `C` has metatypes `P2` and `P1`.

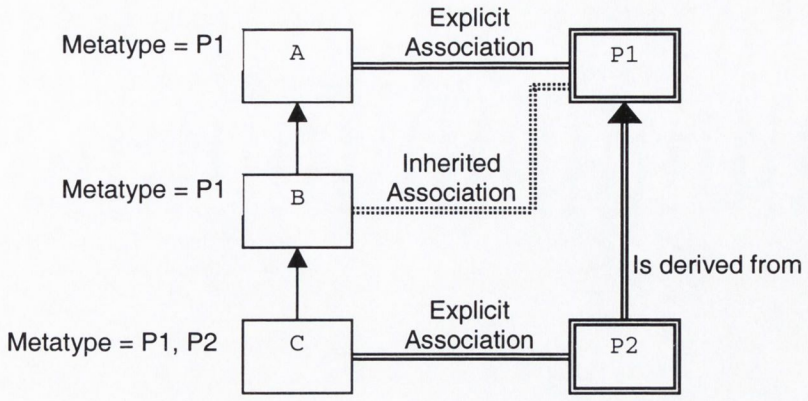


Figure 3.20: A metatype structure in parallel with a type structure.

3.15 Summary

The Iguana/J architecture provides a structured and coherent model for applying unanticipated behavioural changes to a running program. The model uses a refined concept of a metatype to represent a behavioural change and to support a simplified and ordered approach to manipulating changes as independent entities. Metaobject protocols, implementing behavioural change, can be composed with each other and can be applied to arbitrary classes and objects while still maintaining the conventional class and object relationships.

The architecture uses reflective techniques to separate the running system into two identifiable levels, the base-level and the meta-level, representing the original program and the behavioural changes. Behavioural changes are applied by creating associations between classes or objects in the base-level and metaobject protocols in the meta-level. By dynamically creating and replacing associations behavioural changes can be applied to or removed from any part of a program at runtime. The separation maintained between the base-level and the meta-level ensures that particular associations do not need to be anticipated by source code alterations or bytecode preprocessing.

Reusability of the components representing behavioural changes is maximised by providing a parameter mechanism to specialise generic behaviours at association time, and by separating the definition of the behavioural change from the specification and initiation of its association. The model provides this without extending the Java language, allowing the programmer to implement new behaviour using familiar syntax.

Iguana/J does not attempt to support direct structural changes to the application classes or objects. By supporting reification of base-level operations it focuses on behavioural rather than structural changes. Metaobject classes implementing behavioural changes can be created by extending appropriate classes from a small simple library, and can be integrated into coherent metaobject protocols using a simple declaration syntax. Metaobject classes can be used to replace the reified operation in any appropriate way, or to proceed with the original operation, and can include any required operations before or after it. The model supports

decision-making in metaobject classes by providing access at runtime to the arguments, results and execution context of reified operations. It builds on the structural reflection provided by the existing Java reflection classes, and can be viewed as extending the Java structural reflection model to include behavioural reflection.

Chapter 4

Implementation

Then beauty of style and harmony and grace and good rhythm depend on simplicity.

- Plato.
The Republic, 360 BC.

This chapter describes the implementation of the Iguana/J architecture, and discusses some of the issues and conflicts that arose and the choices made for the implementation.

4.1 Introduction

The primary objectives of an implementation of the Iguana/J architecture are that it should support the interception of all reifiable operations, it should support dynamic association between the base-level program and meta-level components, and it should maintain a strong separation between the two levels at runtime. A secondary objective is that it should have as little impact as possible on the execution time of the program. The selection of an implementation approach was considered with these objectives in mind.

The dynamic nature of the Iguana/J architecture means that an implementation of it must support low level manipulation of the execution and runtime structures of a program to be adapted. It must be possible to intercept reifiable operations in any part of the program, and to form the association link between those operations and the appropriate metaobjects. Iguana/J needs to create and remove these interceptions and association links while the program is running, without anticipation of them in the design and development of the program. There are two possible approaches to achieving this without modifying the program's source code [Bou01]. One is to modify the program's bytecode before or as it enters the JVM (program transformation), and the other is to leave the bytecode unchanged but use a modified JVM (interpreter transformation).

Modification of bytecode can be by preprocessing the class files before load time, or by using a special class loader in the JVM at load time. In either case it achieves the objective by inserting 'guard' bytecode blocks, implementing interception hooks, around or instead of code representing reifiable operations. The hooks can be used at runtime to support dynamic association of components implementing behavioural change. This approach is used by two of the architectures reviewed in section 2.3 (Reflex and Java Aspect Components). Bytecode modification before runtime is also used by other architectures, such as Kava and Javassist (see section 2.2), and Weave.NET [Laf03], to carry out the adaptation (or weaving) itself rather than to insert interception hooks to support dynamic adaptation. Inserting interception hooks suffers from a conflict between runtime efficiency and adaptation scope. If the extra bytecode is inserted in every class around every reifiable operation then it is extremely difficult to avoid a serious slowdown of the program, even when none of the operations are actually reified. On the other hand, if the extra bytecode is inserted only for some subset of classes or reifiable operations it limits the scope of dynamic adaptation that may be carried out while the system is running, and can be considered as requiring anticipation of which operations in which classes will be adapted.

A modified JVM supports interception and association using either (or a combination) of two techniques. One is to modify bytecode dynamically within the JVM, inserting and removing 'guard' bytecodes as required. The other is to change the JVM's interpretation of instructions representing reifiable operations. These techniques can provide support for dynamic application and removal of interception mechanisms throughout a running system, while minimising the impact on program execution times and removing any need for anticipation of adaptation. The principal disadvantage is that they usually restrict the portability of the dynamic adaptation architecture by requiring a special virtual machine[8]. Variations of the modified JVM approach were used to implement the architectures supporting unanticipated dynamic adaptation reviewed in chapter 2. Both MetaXa [Gol99] and Guaraná [Oli99] require a specialised JVM to run.

The modified JVM approach was chosen for implementing Iguana/J, because it supports the objectives and provides the best opportunities for dynamic adaptation of arbitrary classes and objects in unanticipated ways. The resulting portability restrictions were not regarded as critical because this prototype implementation of Iguana/J is intended only to be a research platform. The implementation is not complete although it is sufficient to demonstrate and evaluate the model. Some reification categories are not fully implemented, but the prototype is robust and adequate for experimental use. Documentation for installing and using it has been produced [Red03].

4.2 Overview

The implementation of Iguana/J uses a modified JVM approach but it does not depend on source code modification of an existing JVM or creation of a new JVM. Instead, the implementation modifies an existing standard JVM at runtime using a novel technique. This section gives an overview of the structure of the implementation. Issues such as implementing interception, association, composition, meta-level classes and metatypes are discussed in more detail in subsequent sections of this chapter.

[8] An exception to this is the PROSE architecture [Pop01], which uses the JVM debug interface (JVMDI) to intercept operations.

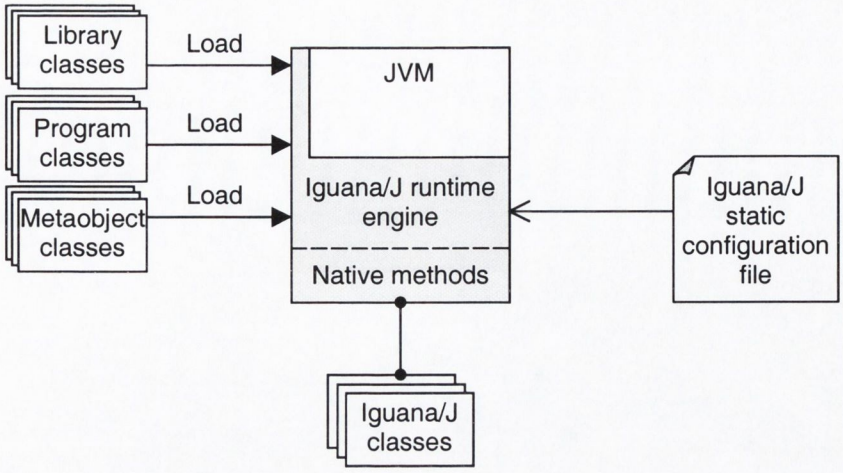


Figure 4.1: Overview of the elements of the implementation.

Iguana/J has been implemented for version 1.3 of the Java virtual machine (JVM) from Sun [Sun03], running on the Windows 32-bit platform. The core of the implementation is a dynamic link library (DLL) containing the Iguana/J runtime engine. The DLL integrates tightly with the JVM via the just-in-time (JIT) compiler interface (described in section 4.3 below), effectively modifying and extending the JVM to provide the features needed by Iguana/J. The original JVM code has not been altered in any way. The runtime engine monitors all class loading by the JVM and has direct access to many of the internal data structures of the JVM. The DLL is written in C. A programmer's interface to the DLL is provided by the set of eleven classes discussed in section 3.4. Many of the methods of these classes are native methods, with their implementations in the Iguana/J DLL.

As each class is loaded by the JVM, the runtime engine intercepts the process and makes some changes to the class's representation within the JVM to allow a metaobject protocol to be associated with the class and its instances if required (see section 4.4.1). At load time the JVM also consults its list of metaobject protocol association information to find any explicit or inherited association for the class. If a metaobject protocol is to be associated with the class then the runtime engine creates the required metaobjects, inserts interception mechanisms into the class being loaded, and creates the links from the class to the metaobjects. Loading of the class then continues as normal. The list of protocol association information for classes is initialised by reading the Iguana/J static configuration file, and is modified over the life of a running system as new associations are created by calls to the Iguana/J `Meta.associate()` methods. Figure 4.1 represents the relationships between the principal elements of the system at runtime.

The runtime engine also ensures that new objects are associated with appropriate metaobject protocols as they are created. If a class has a non-null protocol associated with it then the engine intercepts creation of instances of that class and associates the same metaobject protocol with each new instance.

Iguana/J uses a number of mechanisms to intercept operations in a class or object (see section 4.7). These mechanisms are inserted and removed dynamically, as part of protocol association at class load time and in

response to dynamic association changes while the class or object is in use. The interception mechanisms are implemented by altering the data or bytecode of the class, so interpretation of code in classes with no associated metaobject protocol is not affected in any way. When execution of any operation that has an interception mechanism in place is attempted it is directed initially to a preparation function within the Iguana/J runtime engine, and then to the appropriate metaobject.

The implementation uses and manipulates names of all classes (including base-level classes, protocol classes, and metaobject classes) as fully qualified names, and therefore Java packages are fully respected in all the activities of the implementation. In initiating association operations, statically or dynamically, classes and protocols must be specified using their fully qualified names.

4.3 Integrating with the JVM

The Iguana/J library (DLL) is integrated with the Sun JVM using an interface intended mainly for attaching a just-in-time compiler module. The formal name of the interface is the 'Java Native Code API' [Yel96], although it is commonly referred to as the JIT compiler interface to avoid confusion with the Java Native Interface (JNI) [Sun97] [Lia99]. The JNI is a separate interface for supporting native code methods in Java classes.

The JIT compiler interface is intended to support just-in-time compilation of classes as they are loaded. It is used, for example, by OpenJIT [Oga00], shuJIT [Shu98], and TYA [Kle00]. A native code library can be connected to the interface by specifying the library on the command line when the JVM is started. The JVM will give the library low level access to all classes as they are loaded, with support for making extensive changes to each class. Although the interface is intended for use by a JIT compiler, it may be used for anything that requires interception of class loading and low level access to JVM data structures. The interface has never been part of the Java standard, and is not present on all JVMs. While it is present on Sun JVMs up to version 1.3 the shift by Sun from add-on JIT compilers to the integrated HotSpot compiler technology [Sun01] means that the JIT compiler interface is not present from version 1.4 onwards. This is not a serious problem for Iguana/J because the JIT compiler interface is a prototype implementation technique, not an integral part of the model. The JIT interface is only available in the 'classic' mode of the Sun JVM, but selecting classic mode disables the HotSpot JIT compiler. This means that even though version 1.3 of the Sun JVM supports both the JIT compiler interface and the HotSpot JIT compiler they cannot be used at the same time and so this implementation of Iguana/J cannot take advantage of HotSpot.

This technique for implementing Iguana/J was selected (instead of creating a new JVM or modifying the source code of an existing one) to avoid diverting effort into issues not central to the objectives of this research. By basing the implementation on a known standard JVM with a full implementation of the Java reflection features the research avoided any need to implement a JVM or any part of it. Using the JIT compiler interface instead of modifying the source code of the JVM ensures that the implementation can be used with any Sun JVM up to version 1.3, thus maintaining a limited portability for the implementation. It is not necessary to install a separate JVM to use Iguana/J.

4.4 Trapping class load

The JIT compiler interface operates by allowing the attached library to hook into various JVM actions, notably class loading. As each class is loaded by the JVM, but before its internal representation is fully initialised, the JVM invokes the library's class load hook passing a reference to the loaded class data. This is unrelated to the more conventional Java class loader replacement technique used by some runtime adaptation systems to make changes to classes at load time (e.g. BCA [Kel98], JOIE [Coh98], Javassist [Tat01], Kava [Wel01]). These systems use a subclass of the normal Java class loader to load program classes, thus creating an opportunity to modify the classes. Although this technique provides portable solutions, it has some drawbacks. Replacing the Java class loader gives limited access to classes, is subject to various security restrictions, and can be inadvertantly circumvented by components that specify a different class loader.

In contrast to this, the class load hook provided by the JIT compiler interface traps almost all class loads, even core system classes, and is not affected by which Java class loader is in use. The only class loads that are not trapped by the JIT compiler interface are of a very few of the core system classes that are loaded before the JVM installs the Iguana/J DLL on the interface. These untrapped classes do not receive the Iguana/J extensions and therefore cannot have non-null metaobject protocols associated with them. The classes affected in this way are those that are needed to support the basic class loading and execution functions of the JVM, and therefore are loaded very early in the life of the JVM. This is a limitation of the implementation, not of the Iguana/J model. It affects classes such as:

```
java.lang.Object        java.lang.ClassLoader
java.lang.Class         java.lang.System
java.lang.String        java.lang.Throwable
java.lang.Exception     java.lang.Error
```

4.4.1 Extending class and object structure

The Iguana/J runtime engine makes certain changes to the structure of every class as it is loaded by the JVM, whether or not the class is known to be associated with a metaobject protocol. These changes are to insert default association information into the class, and to increase the specified size of each instance of the class to allow for per-object association and reification information. These changes do not themselves affect the normal operation of the class or its instances. They only provide space for use in a possible future association with a metaobject protocol. At a more abstract level these changes make the class appear to be associated with the null metaobject protocol.

The changes include adjustments to ensure that the JVM's garbage collector will find and mark any object references that may be stored in the extended class and object space. Making the changes to every class ensures that every class and object (with the exceptions of the early-loaded classes noted in the previous subsection) has the capability to be associated with a metaobject protocol. This supports Iguana/J's unanticipation of adaptation.

4.4.2 Access to JVM data structures

The JIT compiler interface allows the Iguana/J runtime engine to find and, if required, manipulate the data relating to previously loaded classes, objects, constructors, methods and fields directly. It also enables low level activities such as direct searching of the JVM's object pool. This supports implementation of the mechanisms for dynamic manipulation of association and reification.

4.4.3 Functions of the library

The Iguana/J runtime engine can be considered as providing three general functions to implement the model. These general functions are activated in three different ways (illustrated in figure 4.2):

- Trapping a class load: This is used to prepare the class data structures, and to associate a metaobject protocol with the class if it is required.
- Trapping an intercepted program operation: This is used to transfer reified operations to the meta-level.
- Invoking a method of the Iguana/J class library: This supports the meta-level programmer's interface to the Iguana/J runtime engine, particularly manipulation of associations between protocols and classes or objects.

Outside these three situations the original JVM operates as normal, without any involvement or influence from Iguana/J. If the Iguana/J library is loaded but no classes or objects are associated with a non-null metaobject protocol then the only effect of the library's presence (other than the memory space occupied) is a slight slowdown of the loading of each class. That is caused by the Iguana/J runtime engine trapping the load of each class in case it needs a protocol to be associated with it.

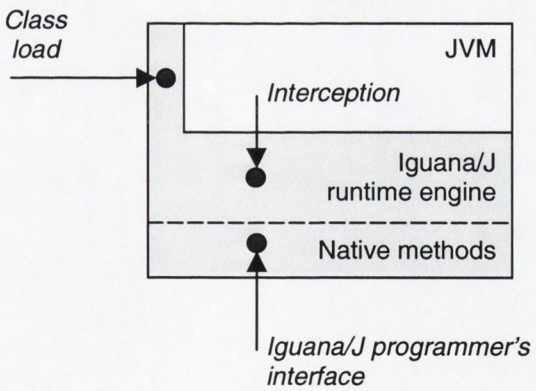


Figure 4.2: The three types of event that activate Iguana/J.

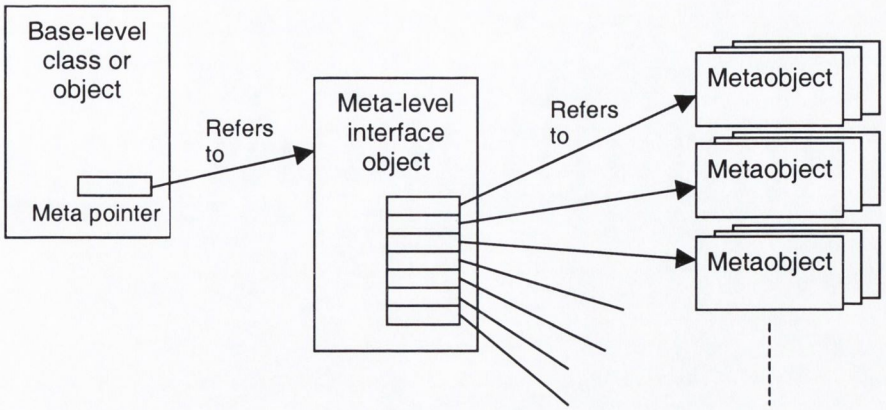


Figure 4.3: The meta-level interface object refers to metaobjects reifying each operation category.

4.5 Association

The Iguana/J model allows a metaobject protocol to be associated with a class or with an object. The implementation mechanism must support the dynamic nature of the association, while maintaining a strong separation between the program components and the meta-level components.

4.5.1 Implementing association

As described earlier (see section 3.6) each metaobject protocol is represented at runtime by a Java class generated by the protocol compilation process as a subclass of class `Meta`. An instance of this protocol class acts as a meta-level interface object for each association with that protocol. The association is implemented as a reference, called the *meta pointer*, from the program class or object to the meta-level interface object. The meta pointer is only visible to the Iguana/J engine; it is not directly accessible from Java code. The meta pointer is the only link between the base-level and the meta-level components.

The meta-level interface object contains references to the set of metaobjects reifying the protocol's categories (see figure 4.3). When there is more than one metaobject for a category the meta-level interface object implements Iguana/J's linear composition model. When a reified operation occurs in the base-level class or object the meta pointer is followed to find the meta-level interface object and then the metaobject(s) reifying that category of operation.

4.5.2 Creating an association

When it is required to create an association between a protocol and a class or object three steps are carried out by the runtime engine to put the association in place:

1. Create the meta-level interface object and the metaobjects.

   The meta-level interface object is an instance of the class of the same name as the metaobject protocol. The runtime engine discovers the structure of the protocol by examining the protocol class, including examination of all its parent protocols. This examination provides details of the protocol's reification categories and metaobject classes. It enables the engine to construct the metaobjects and link them into composition chains, using fields and data structures within the meta-level interface object.

2. Insert the interception mechanisms required by the reification categories.

   From step 1 the runtime engine knows the categories of operation reified by the protocol, and in this step it inserts the interception mechanisms for each of those categories. It does not insert interception of operation categories that are not reified (with the exception of object creation – see section 4.5.3 below).

3. Set the meta pointer from the base class or object to the meta-level interface object.

The only preparations that a base-level class or object must have before it can be associated with a protocol are the structural extensions that are carried out when the class is loaded. As the data structures for all classes have these adjustments (with the exception of the early-loaded classes noted in section 4.4), and the three steps above can be carried out on any class at any time, it is apparent that any protocol can be associated with an arbitrary class at any time without anticipating it in the class before runtime. In addition, the association can be undone at any time by deleting the meta pointer value and removing the interception mechanisms.

4.5.3 Association with new objects

When a class has an associated metaobject protocol the Iguana/J model requires that each new instance of that class have an association with the same protocol from the moment it is created. This means that the runtime engine must be able to detect and intercept object creation for any class that has a protocol associated with it. This in turn means that the interception mechanism used to reify object creation must be inserted for every class that is associated with a protocol, even if that protocol does not reify object creation.

When a creation operation is intercepted the runtime engine will check whether creation is reified by the protocol associated with the base-level class. If it is then the creation operation will be passed to the appropriate metaobject. When the meta-level handling of creation finishes, or if creation is not reified, the runtime engine will associate the new object with the protocol that is associated with the object's class. Associating the new object with a protocol involves the three steps described in section 4.5.2.

4.5.4 Interaction between object associations

As required by the Iguana/J model the implementation supports association of different instances of a class with different metaobject protocols. The principal mechanism supporting this is that a class and each of its instances have separate meta pointers and separate meta-level interface objects. However, instances of one class share method code so for some reification categories it is difficult to implement interception without

affecting all instances equally. This becomes an issue if one or more instances of a class are associated with a different protocol than other instances of the same class, and the two protocols reify different categories. The categories of operation intercepted for instances of the class will be the union of the categories reified by the two protocols.

As a result of this some operations will be intercepted unnecessarily. It is not easy to avoid this without complex class duplication or shadowing (e.g. MetaXa [Gol99]). The Iguana/J implementation allows the interception to take place but detects that reification is not required when it attempts to transfer the operation to the meta-level. In fact, this detection takes place very quickly, as each object contains a set of flags indicating which categories are reified for it. When an interception occurs for an object the appropriate flag is examined and if it is not set then the Iguana/J engine returns control to the interpreter without further delay. The effect of this is that unreified operations on an object, where those operations are reified for another instance of the same class, suffer a small performance penalty.

4.5.5 Inheritance of association

In the Iguana/J model the semantics of association of a protocol with a class include association of the same protocol with the class's subclasses. The implementation must support this whether the subclass is loaded before or after the class has become associated with a protocol.

To implement loading a subclass when its superclass already has an association with a protocol, the runtime engine maintains an internal class association table, recording all known protocol associations required by classes. Each entry in the table describes one explicitly initiated association, including the name of the base-level class, the name of the protocol, and the values of any parameters supplied when the association was initiated. The table allows the runtime engine to discover whether a class being loaded should be associated with a (non-null) protocol, even if the association is inherited.

Entries are made in the association table in two ways, caused by static and dynamic initiation of association. The association table is initialised by the Iguana/J library at startup by reading the static association declarations in the Iguana/J configuration file. Each declaration causes one entry to be made in the table; it does not cause any classes to be loaded. Dynamic initiation of association with classes, via invocations of the method `Meta.associate()`, modifies the contents of the association table during runtime. Each dynamic initiation of association with a class causes an entry to be made in the association table, or an entry to be replaced if there is already one for the base-level class named in the association. Any existing entries for subclasses of the named base-level class are removed from the table, in order to ensure that further subclasses that may be loaded in the future will be associated with the newly specified protoocol. Thus the association table describes all known associations between classes and protocols, although at any given time some of the classes for which it describes associations may be loaded and some may not.

During the normal course of system operation loading of each class is trapped by the runtime engine, which checks the class name against the association table. If the table contains an entry describing an association for the class or any of its superclasses then the class is associated with that protocol as it is being loaded. If there are multiple possibilities the implementation selects the most specific association. The

association table actually performs two roles. It allows statically initiated associations to take effect even if the base-level class is not loaded until after the system has been initialised, and it ensures that associations will be inherited by subclasses that are loaded after the association is initiated (statically or dynamically) with their superclass.

Inheritance of association in the other possible case, where a subclass is already loaded when a protocol association is initiated for one of its superclasses, is implemented by a different mechanism. When a protocol is to be associated with a class, the runtime engine also associates it with any loaded subclasses. It finds the subclasses by examining the JVM's list of loaded classes. This case can only occur for dynamically initiated association. It cannot occur for an association applied to a class as it is being loaded, because it is not possible for a class to be fully loaded before its superclasses.

4.5.6 Association with an existing class

When a dynamically initiated association with an already-loaded class occurs there are a number of issues that must be addressed by the implementation. One is that the semantics of association with a class includes not just association with its subclasses, as discussed above, but also association with all existing instances of the class and its subclasses. The implementation of Iguana/J supports this by searching the JVM's object pool for all instances of each class and individually associating the protocol with each one found. This means that dynamically initiating an association of a protocol with a class has the potential to require a significant amount of work and to cause the creation of a large number of new (meta-level) objects. Association of a protocol with a class when it is being loaded does not have the same potential to require extra work, because at load time no subclasses are loaded and no instances exist yet.

4.5.7 Shared metaobjects

A shared metaobject, as described in section 3.11, requires that a single instance of it be shared by all base-level objects of one class that are associated with that protocol or any protocol derived from it. This means that when the runtime engine is creating an association between a base-level object and a protocol that includes shared metaobjects, it must check for any other occurrence of that protocol in the associations with the base-level object's class or other instances of the class.

A straightforward implementation would involve a potentially lengthy search of the protocol derivation tree of every protocol associated with the base-level class or any of its instances. To avoid this the Iguana/J runtime engine maintains a list for each base-level class of references to shared metaobjects in protocols associated with the class or any instances of it. When an association being created requires a shared metaobject the engine checks the list for the base-level class, and if an entry is found the appropriate metaobject reference is copied. If an entry is not found the metaobject is created and an entry is added to the list.

4.5.8 Changing association

The Iguana/J model requires that when an existing protocol association for a class or object is being replaced, some metaobjects from the existing association may have to be retained for the new association. This occurs when the existing protocol has one or more contributing protocols in common with the replacement protocol (see section 3.13). To implement this the runtime engine compares the two sets of contributing protocols (existing and replacement), when the replacement association is initiated, and identifies common ones. The metaobjects contributed by the common protocols are retained for the new configuration, instead of creating new metaobjects, as would otherwise be the case.

It is apparent that assembling the collection of metaobjects needed for an association between a metaobject protocol and a class or object is a potentially complex operation. Some metaobjects may have to be retained from a previous protocol, some metaobjects may have to be shared with other associations, and some metaobjects have to be created.

4.5.9 Package-wide association

The Iguana/J model supports a limited use of wildcard characters in base-level class names for associations, as described in section 3.10.4. This allows a protocol to be associated with all classes of a Java package using a single association statement (statically or dynamically initiated), and avoids a need to know the names of all the classes in the package. Package-wide association is intended to have the same effect as individual associations with every class of the package (see section 3.10.4). One of the implications of this is that package-wide association must also affect classes of the package that are loaded after the package-wide association is initiated. That is, it should make no difference whether the package-wide association is applied and then an affected class is loaded, or a class is loaded and then a package-wide association that affects it is applied. The class should end up with the same protocol association in both cases. This requires that package-wide association descriptions be stored in the Iguana/J internal association table at runtime, just like other association descriptions for base-level classes, and be consulted as each class is loaded.

However, supporting package-wide association introduces a potential difficulty in ensuring that the correct association is used for a class being loaded. The difficulty occurs if both a package-wide association for its package and a different association for a superclass that is outside its package are applied before the class is loaded. When the class is loaded there is an issue in ensuring that the class ends up with the same association as it would have if it had been loaded before those associations were applied. This occurs because of overlapping influences from a package-wide association and an association inherited from outside the package. In this situation the implementation should ensure that it make no difference to the end result whether a class is loaded before or after application of any associations that affect the class[9].

[9] If this were not the case then it would be possible that a change in a program that resulted in a different program class load order might cause different protocol associations. The reasons for such different associations are not likely to be obvious to a programmer.

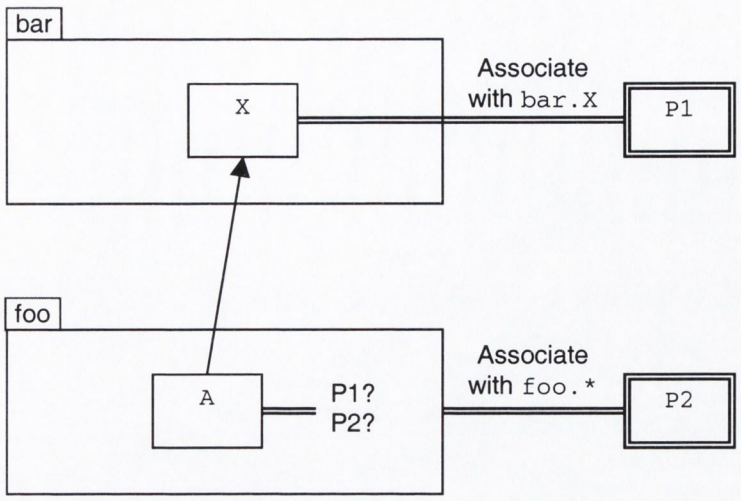


Figure 4.4: Conflict between package-wide and inherited association.

To illustrate the difficulty consider the two classes in figure 4.4. Class `foo.A` inherits from a class in another package, class `bar.X`. Protocol `P1` is associated explicitly with `bar.X`, and protocol `P2` has a package-wide assoication with `foo.*`[10]. Consider the effect on class `foo.A` of a number of different sequences of association and class load events. First, consider the following sequence of events:

4. Class `bar.X` is loaded.

5. Class `foo.A` is loaded.

6. Protocol `P1` is associated with `bar.x`.

7. Protocol `P2` is associated with `foo.*`.

This sequence of events causes no particular difficulties because all affected classes are already loaded when the associations are initiated. Both associations will affect `foo.A` (the first because `A` is a subclass of `X`, and the second because `A` is in the package `foo`). `foo.A` ends up with the most recent association; that is, it ends up associated with protocol `P2`.

Now consider the situation if `foo.A` is not loaded until the end of the sequence of events:

8. Class `bar.X` is loaded.

9. Protocol `P1` is associated with `bar.x`.

10. Protocol `P2` is associated with `foo.*`.

11. Class `foo.A` is loaded.

[10] To avoid violating metatype rules (see section 3.14.5) `P2` must be derived from `P1`.

At the time that `foo.A` is loaded there are two associations recorded in Iguana/J' s internal association table that both potentially affect `foo.A`. To ensure that the end result is the same as if `foo.A` had been loaded before either of the associations had been applied (as in the first sequence above) it is apparent that the package-wide association with `P2` should be applied rather than the inherited association with `P1`.

Lastly, consider the situation in the two sequences above if the order of applying the two associations is reversed. That is, if the package-wide association of `P2` with `foo.*` is applied before the association of `P1` with `bar.x`. If `foo.A` is loaded before these associations are applied then `foo.A` will end up associated with `P1` (the last of the two association to be applied). If `foo.A` is loaded after these associations are applied it should have the same result, so `foo.A` should be associated with `P1`. This is the opposite of the result in the second sequence listed above.

It is apparent from the examples above that if there is both a package-wide association and a different inherited association that could be applied to a class being loaded then it is the most recently initiated of the two possible associations that should be used. This is easily implemented in Iguana/J, as the associations are recorded in the internal association table in order of initiation.

4.6 Composition

Composition of metaobjects for a single reification category is implemented in the Java class `ie.tcd.iguana.Meta`. For each possible reification category the class defines an array of references to metaobjects. At runtime each meta-level interface object (an instance of a protocol class, which is a subclass of `Meta`) stores references to all its metaobjects in these arrays. When a reified operation is transferred to the meta-level interface object for dispatching, it is passed in sequence to each of the metaobjects in the array appropriate to the category of the operation. Any metaobjects that are shared between base-level objects will have references to them from multiple meta-level interface objects.

Potentially, the linear composition model of Iguana/J could be altered by subclassing `Meta`, modifying its dispatching methods, and then using the subclass as the basis for protocol classes. However, this is not directly supported by the current model or implementation. In particular, the protocol declaration does not provide any way to specify alternative composition strategies, and the Iguana/J protocol compiler does not support the use of any class other than `Meta` as the direct superclass of all protocol classes.

4.7 Interception

Interception is the process of trapping operations in the base-level so that they may be reified. However, not all intercepted operations are reified. As discussed earlier (see section 4.5.4) this can occur when a category is reified for one instance of a class but not for others. It can also occur when a category reified in a subclass affects methods or fields inherited from a superclass where they are not reified. The interception mechanisms must be inserted to support reification in the subclass, and this will also cause the interception to occur when the same operation is attempted in the superclass. The interception mechanisms attempt to detect

these cases quickly, and to resume the intercepted operation. In general the interception mechanisms must be as efficient as possible in execution time, to reduce their impact on both reified operations and intercepted but unreified operations. To avoid affecting execution times of operations that are not reified by any instance or class the implementation allows interception to be inserted only when and where it is needed. This implies support for dynamic insertion and removal of interception mechanisms for each reification category.

Characteristic	Bytecode	Interpreter
Execution speed	Slower	Faster
Insertion complexity	High	Low
Scope of interception	Localised	Global

Table 4.1: Comparison of interception techniques.

Interception mechanisms can be implemented by dynamic alterations to the bytecode in a base-level class or by alterations to the interpretation process. These two general techniques are compared briefly here. A summary of the comparison is given in table 4.1. In general alterations at or around the bytecode representing the operation to be intercepted will require the insertion of extra bytecode instructions. On the other hand, alterations to the interpretation of the operation to be intercepted will be in native code and therefore will execute faster.

Inserting a bytecode-based interception sequence into a class is likely to be complex, because the insertion may alter some code offsets and may require modifications to internal linking tables in the class. These kinds of changes are typically carried out using a load time tool such as Javassist [Tat01], JOIE [Coh98], BCA [Kel98], or BCEL [Dah99]. The main operational difficulty with the modified-interpretation technique is that in its simplest form it affects the interpretation of every reifiable operation, slowing each one by a very small amount. This occurs because a typical implementation of this technique is to include in the interpretation of each reifiable operation a check as to whether or not the operation is reified for this class or object (e.g. Guaraná [Oli99]).

In fact this implementation of the Iguana/J model cannot make use of the simple modified-interpretation technique outlined above, because the JIT compiler interface does not support modification of the JVM's interpretation of bytecodes. However, the design of the Sun JVM used as the basis for this implementation of Iguana/J allows for a variation on the modified-interpretation technique for intercepting some categories of operation. It retains the characteristics of faster execution speed and low insertion complexity, but provides a localised rather than a global scope for each interception.

In the Sun JVM the data structure for every method or constructor includes a pointer to the function within the JVM that is to be used to invoke it. These *invoker functions* prepare the Java stack frame and perform any other preparation needed to begin execution of the method that has been called. The JVM uses this technique because it has a number of possible method invokers, each optimized for methods with different combinations of characteristics such as synchronised, compiled, or native. To intercept invocation of a method Iguana/J simply changes the method's invoker pointer to point to a special function in the Iguana/J runtime engine. This allows Iguana/J to intercept invocation of that method without altering

bytecode and without altering the interpretation of all invocation instructions. If the intercepted invocation is not to be reified then the original invoker function is called to resume the intercepted operation. If the interception mechanism is to be removed for that method at some later time, then the method's invoker function pointer is simply changed back to its original value.

Changing invoker function pointers is also used to intercept object creation, but the overall process is more complex than for intercepting method invocation. By changing the invoker function pointer for all constructors in a class the runtime engine intercepts creation of any instance of that class, whether initiated by using `new` or by using the `newInstance()` method of class `Constructor` (or even by using object creation functions in native code methods). While this is a simple way to intercept object creation it is not sufficient to support reification of creation. For greatest flexibility reification of creation should give the appropriate metaobject full control over the creation operation, including supplying an instance of some other (compatible) class instead of the class in which creation was intercepted. The difficulty is that object creation is actually a sequence of operations, of which calling the constructor is almost the last (see [Lin99] §2.17.6 and §7.8). When the constructor invocation is intercepted the object has already been created but has not yet been fully initialised. In order to allow the creation to be properly reified the partially initialised object is discarded, the metaobject is called, and the object returned by the metaobject is used to replace the discarded object.

Every Java constructor always calls a constructor of its direct superclass as its first action ([Gos00] §12.5). If object creation is reified in the superclass then a call to its constructor will be intercepted even if it comes from a subclass constructor. The implementation of Iguana/J must ensure that interception of a superclass constructor call does not transfer the creation to the meta-level of the superclass, as a transfer to the meta-level has already been carried out by interception of the subclass constructor call. This is implemented by an examination of some of the data values in the object at the start of the intercepting invoker function in the runtime engine. If the examination shows that creation of the object has already been processed by the Iguana/J runtime engine then it will allow the superclass constructor to proceed as normal.

4.8 Transfer to meta-level

When an operation is intercepted and is to be transferred to the meta-level the intercepting function in the Iguana/J runtime engine must prepare the operation for the transfer. Up to three data items must be passed to fully represent the operation being transferred. They are the object in which the operation was intercepted, the arguments (if any) to the operation, and the method, field, or constructor that is the target of the operation. The principal actions required to do this are to marshall any arguments to the operation, and to acquire a reference to the Java reflection object reifying the target of the operation. The Java reflection classes are used to provide structural reflection for Iguana/J (see section 3.4), and the classes that reify methods, fields, and constructors support operations such as invocation of methods and constructors and access to fields. To illustrate transfer of an operation to the meta-level consider what happens when a reified method invocation is intercepted by the Iguana/J runtime engine. Argument marshalling is carried out by creating an array of type `Object`, of length equal to the number of arguments to the method. Then the arguments to the invocation are retrieved from the Java stack and stored in the array, creating appropriate

wrapper objects for any arguments of primitive types. Then a reference to an instance of class `java.lang.reflection.Method` reifying the target method is acquired. The three transfer arguments are then a reference to the base-level object, a reference to the argument array, and a reference to the `Method` object. To perform the transfer the runtime engine uses the meta pointer of the base-level class or object to find the meta-level interface object, finds the identifier for the dispatch method in the meta-level interface object, creates a new Java stack frame to contain the transfer arguments and invokes the dispatch method.

Much of this preparation work does not vary between reified invocations of the same method, and therefore it can be done once and the results stored for use each time a reified invocation of the method occurs. The time required to create an object is a multiple of the time to invoke a method (because creation includes invoking at least one constructor), so avoiding repetitive creation of the argument array and wrapper objects for primitive types is important for minimising the execution time of a reified invocation. This is implemented by caching and reusing these objects for each method whose invocation is reified. Each time such a method returns, references to the argument array and any primitive wrappers are retrieved from the Java stack and stored in a per-method cache. Each time an invocation of the method is being prepared for transfer to the meta-level an array and any appropriate primitive wrappers are taken from the cache. If the cache for that method is empty then new array and wrapper objects are created.

One of the issues that causes complexities in the class library for many Java adaptation architectures is how to allow for the variation in return types from a dispatch or handler method for a reification. The return type for a Java method invocation may be any of eight primitive types, a named class or interface, or `void`. While a general dispatch or handler method used to reify invocation of methods can be designed to accept arbitrary numbers and types of marshalled arguments it is more difficult to design it to handle an arbitrary return type. Most solutions involve wrapping the return value in a special object, or creating a set of dispatch or handler methods, one for each return type (e.g. MetaXa [Gol99]). Typically the programmer must be aware of the issue and must explicitly unwrap the return value or take steps to use the dispatcher or handler with the appropriate return type. For example, MetaXa requires the meta-level programmer to use methods whose name includes the name of the return type.

The implementation of Iguana/J deals with this issue with minimum involvement from the meta-level programmer, using wrapper objects in a way that is consistent with the treatment of return types in the Java reflection classes. All handler methods in metaobjects that return values are declared to return type `Object`. This allows any type to be returned from the metaobject, with primitive types wrapped in instances of their equivalent Java classes, and `void` type returned as null. If the metaobject generates its own return value, and that value is of a primitive type, then it must be wrapped explicitly before it can be returned. Otherwise if the metaobject uses the `proceed()` method to carry out the operation then the result is wrapped automatically before being returned by `proceed()`. In any case the return path through the dispatch method in class `Meta` automatically unwraps primitive return types, and effectively casts other types to the expected type. If there is an error in the type returned the Iguana/J runtime engine throws a `ClassCastException` back to the base-level code containing the reified operation (see section 4.14 for a description of how exceptions are handled).

4.9 Recursive reflection

In a reflective system the possibility of recursive reflection [Mae87] must be addressed. This can occur if a metaobject attempts an operation that is reified by an instance of itself, and it can lead to an infinite chain of recursive interceptions and level transfers. In practice the circumstances leading to this situation can occur very easily. For example, consider a metaobject that reifies method invocation in a base-level object. If, as part of its operation, the metaobject attempts to invoke any method of the base-level object then that invocation will itself be reified. The reification will cause the operation to be transferred to the same metaobject, where the invocation will be attempted again. In some reflective architectures the problem may even arise if a metaobject attempts to proceed with the base-level invocation that it reifies.

A solution to this problem must ensure that a normally reified operation on a base-level object is not reified when control has already been transferred to the meta-level. Some reflective architectures do this by providing special non-reflective techniques to be used for access to reifiable operations from meta-level components. These special techniques bypass the normal interception mechanisms, and thus avoid the reification. For example, Guaraná [Oli99] provides non-reflective versions of methods such as `hashCode()`, `equals()` and `toString()` that must be used by metaobjects. MetaXa [Gol99] provides method `continueExecution()` that must be used by the metaobject to carry out the base-level operation that it reifies, in order to bypass the reification mechanism.

The implementation of Iguana/J solves the problem without requiring any action by the programmer, and without imposing any programming restrictions. When a normally reified operation is intercepted by the runtime engine the Java stack is examined to determine whether the operation has originated in the base-level or the meta-level. If it originated in the base-level then the operation is reified. If it originated in the meta-level then the operation is not reified, and is just carried out in the normal way. The stack examination determines which level the operation originated in by detecting stack frames caused by the transfer of control to the meta-level interface object, and stack frames caused by reflective access to the base-level (such as `Method.invoke()`).

The stack examination solution to the infinite reflection problem also causes an association of a protocol with a metaobject to have almost no effect. Although there is nothing to prevent such an association being created most operations will not be reified no matter what is declared in the protocol. Direct operations on the metaobject, originating in a base-level object, will be reified but all others will not. They will be intercepted, but the stack examination will determine that they originated in the meta-level so they will not be transferred to the higer level reifying metaobject. This means that in this implementation it is not possible to use a metaobject protocol to change the behaviour of a metaobject; it is not possible to create a meta-level of a meta-level This is not considered to be a serious restriction, as the Iguana/J architecture is not intended to support unanticipated dynamic adaptation of unanticipated dynamic adaptation.

```
protocol P2(String s, int i, boolean b) : P1(i) {
  local: reify Creation: GuardCreate(b);
  shared: reify Invocation: GuardExecute(s);
}
```

Figure 4.5: A sample protocol P2 derived from P1 and reifying two categories.

```
public final class P2 extends Meta {
  // ...
  public static void makeMetaobjects(Meta m, int n, String _s,
                      int _i, boolean _b) {
    m.mCreate[n] = new GuardCreate(_b);
    m.mCreate[n].shared = false;
    m.mExecute[n] = new GuardExecute(_s);
    m.mExecute[n].shared = true;
    P1.makeMetaobjects(m, n+1, _i);
  }
}
```

Figure 4.6: A simplified fragment of the class created by compiling protocol P2.

4.10 Protocol classes

As described in section 3.5 every metaobject protocol is represented at runtime by a protocol class created from the protocol declaration by the Iguana/J protocol compiler. The compiler itself is implemented as a Java program, separate from the Iguana/J runtime engine and class library. The protocol compiler reads and parses a protocol declaration and uses the information to write the source code of a specialised Java class. The protocol compiler then invokes the Java compiler to create the class file.

Every protocol class is created as a direct subclass of `ie.tcd.iguana.Meta`. The details of the protocol are represented in the class as statements that create the declared metaobjects and request any direct parent protocols to create their metaobjects. Consider the example protocol `P2`, shown in figure 4.5. This declares that it is derived from `P1`, that it expects 3 parameters (one of which is passed on to `P1`), and that it reifies creation with a local metaobject and invocation with a shared metaobject. Figure 4.6 shows a part of the protocol class created by using the Iguana/J protocol compiler on the declaration in figure 4.5. The method `makeMetaobjects()` (shown here in a slightly simplified form) is called by the runtime engine when it wants to associate this protocol with a base-level class or object. The method creates instances of each of the metaobjects declared explicitly by `P2` and invokes the `makeMetaobjects()` method of its parent protocol to create the remaining metaobjects. The metaobject references are accumulated in a set of sparsely populated arrays, one per reification category. These arrays are part of the metaobject composition mechanism[11], and filling them by recursive invocations of `makeMetaobjects()` implements the depth-first traversal of the protocol derivation tree described in section 3.8.2. The `makeMetaobjects()` method also includes code (omitted in figure 4.6) to avoid adding metaobjects by the same protocol more

[11] There is a preliminary traversal of parent protocols (not shown in the listings) to determine required array sizes just before they are used.

than once. This implements the policy on multiple path derivation described in section 3.8.5. `makeMetaobjects()` also checks for circular protocol derivations, and throws `ie.tcd.iguana.IguanaError` if one is encountered.

It is apparent from this example that, as described in section 3.8.6, the class representing a derived protocol has a compiled-in dependency on the name of its direct parent protocols and on the number and type of parameters expected by them. The upward delegation of metaobject creation at runtime means that it has no dependency on the categories reified by its direct parents, or on the names of the metaobject classes used by its direct parents, or on the number or details of indirect parent protocols.

4.11 Metatype rules

The metatype rules described in section 3.14.5 are implemented in the Iguana/J runtime engine by performing checks on the superclass when a protocol is to be associated with a class or on the class when an association is with an object. There is just one complication to this procedure. When a class is being associated with a protocol as it (the class) is being loaded, it is possible that the class's superclass is not yet fully initialised and therefore that the protocol associated with the superclass cannot be checked at that time. This only happens if the superclass was not previously loaded, and is being loaded now because a class's superclass must be loaded before its own load can be completed.

If a set of superclasses are being loaded because a subclass is required then the sequence used by the JVM is that they are all loaded and partially initialised, and then the JVM passes them to the JIT compiler interface one by one starting from the subclass and moving up through the superclasses[12]. This means that when the subclass is being associated with a protocol by Iguana/J its superclass has not yet been passed to Iguana/J for a possible protocol association, and therefore Iguana/J cannot check at that time that there is a valid metatype relationship between the protocols of the two classes.

The solution adopted by this implementation of Iguana/J is to postpone the metatype checking if the superclass is not yet fully initialised. The requested protocol association for the base-level class is completed, but the class is added to an internal list for later metatype checking. Each time the Iguana/J runtime engine finishes processing a class being loaded by the JVM it also checks the list of classes whose metatype checking has been postponed. If any classes on the list have a superclass whose initialisation is now complete then the metatype relationship between the class and the superclass is checked. Exception `ie.tcd.iguana.IguanaException` is thrown if a metatype rule violation is detected.

4.12 Null metaobject protocol

As described in section 3.9 the null protocol is different from other metaobject protocols in a number of ways:

[12] The recursive class initialisation sequence is defined in § 2.17.5 of the JVM specification [Lin99]. Each class is passed to the JIT compiler interface between steps 6 and 7 of that sequence.

1. It does not reify any category of operation.

2. It is the default parent of all protocols that do not declare a parent.

3. It is associated by default with every class and object that is not associated with another protocol.

4. A class to represent it is provided as part of the Iguana/J class library.

Because association with the null protocol seems likely to be the common case in many systems the implementation attempts to keep such associations as lightweight as possible. The principal way in which this is achieved is to treat the null protocol as a special case, with supporting code in the Iguana/J runtime engine and in some of the Iguana/J classes. This mainly consists of using and detecting special data values for classes and objects associated with the null protocol. For example, the meta pointer in an object associated with the null protocol is itself a null value. As the contents of every object are automatically initialised to zero this avoids a need to intercept creation for every class in the system in order to associate every new instance with the null protocol. Any code that attempts to retrieve the meta pointer value detects the null value and substitutes a reference to the single shared instance of the null protocol class.

Item 2 in the list above is implemented in the metatype checking functions of the runtime engine by ensuring that they always return 'true' if the parent protocol of the pair being tested is the null protocol. Item 3 is implemented for assocation with classes by initialising the internal class association table (described in section 4.5.5) with a single entry associating `java.lang.Object` with the null protocol. This causes this association to be automatically inherited by every class that is not associated with some other protocol.

4.13 Class loaders

When a protocol class or metaobject class is to be loaded the question of which class loader to use arises. A class loader is an instance of `java.lang.ClassLoader` or a subclass of it. The concept of a class loader, and the ability to create new class loaders, provides support for Java programs to load classes in application-specific ways [Lia98]. The Java virtual machine also assigns classes to separate class namespaces according to the loader used to load them. By default the loader used for a class is the same as the one used to load the class causing the load. For example, if a class `A` causes a class `B` to be loaded, `B` will be loaded using the same loader that was used to load `A`. This behaviour can be changed by using class load methods of class `Class` or class `ClassLoader`, and the use of the alternative class loader will automatically propagate down through a class loaded in that way.

The difficulty is how to decide what loader should be used for a given protocol class and the metaobject classes to which it refers. Should it be the loader of the base-level class that the protocol will be associated with or should it be the loader of the class creating the association?[13] Or should it be the 'system class loader', the initial default loader used by the JVM for application classes? Choice of class loader can affect

[13] Obviously this choice is only important if the base-level class and the class making the association have different loaders.

where the system searches for the protocol and metaobject classes, and affects which namespace the loaded protocol and metaobject classes are placed in.

This implementation uses the system class loader to load all protocol and metaobject classes. This places the classes in the intersection of all class namespaces, and therefore makes the protocol and metaobject classes available to all other classes. It avoids loading multiple copies of them for separate namespaces, and it avoids the need for this implementation of the Iguana/J runtime engine to be concerned with class namespaces at all.

The only limitation introduced by always using the system class loader is that the protocol and metaobject classes cannot be loaded from any source other than the normal Java classpath. In other words it is not possible to use a custom class loader to load protocol and metaobject classes in some application-specific way.

4.14 Exceptions

There are three different potential sources of exceptions that concern the Iguana/J implementation. They are the base-level program, the meta-level components, and the Iguana/J runtime engine. They are discussed separately below. In considering the treatment of exceptions that may be thrown by the handler method in a metaobject it should be remembered that handler methods override default ones inherited from the appropriate metaobject class in the Iguana/J class library. Java does not allow an overriding method to declare to throw more exceptions than the method it overrides, so it is not possible to add extra exceptions to the `throws` clause of handler methods in metaobject classes.

The three possible sources of exceptions are discussed below. Some of the separate cases are illustrated in figure 4.7.

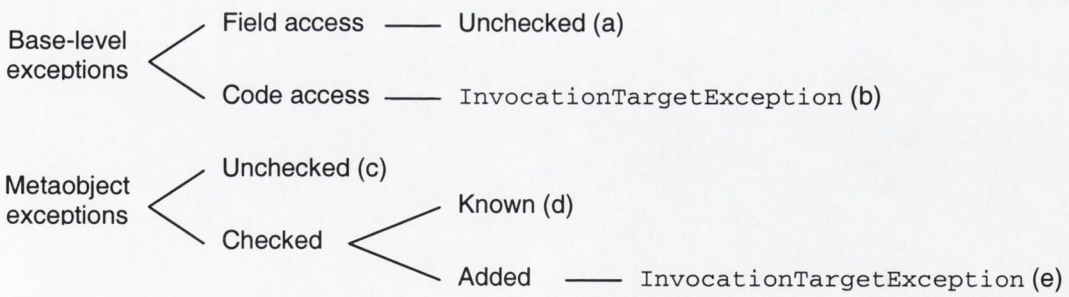


Figure 4.7: Cases of exception sources and handling

- Exceptions thrown by a base-level operation that has been reified and then executed from the meta-level.

  These are uncaught exceptions thrown in the course of a base-level operation, and they may be checked or unchecked exceptions[14]. The only consideration for the implementation of Iguana/J is that it must pass them back through the metaobject and the return half of the interception mechanism, and it must ensure that they are thrown in the base-level program as if the reification had not intervened. If the reified operation is field access then only unchecked exceptions are possible (see figure 4.7 case (a)); they are thrown in the metaobject, and if not caught there are passed back and rethrown in the base-level program at the point where the operation was intercepted. If the reified operation involves execution of a method or constructor then both checked and unchecked exceptions are possible, but in either case the Java reflection class representing the method or constructor will wrap the exception in an instance of `java.lang.reflect.InvocationTargetException` before throwing it (see figure 4.7 case (b)). This supports generalised treatment of all checked exceptions without causing compile time errors. A metaobject can catch this exception and examine the underlying one if required. If the metaobject rethrows the `InvocationTargetException` it is automatically unwrapped by the Iguana/J runtime engine before the underlying exception is thrown in the base-level code at the point where the operation was intercepted.

- Exceptions thrown by a metaobject while handling a reified operation.

  Unchecked exceptions thrown by a metaobject are thrown back to the base-level program, as if they had occurred in the operation that the metaobject reified (see figure 4.7 case (c)). Checked exceptions are a little more complex because when the metaobject class is compiled the compiler must be satisfied that they are handled correctly. For this discussion checked exceptions are divided into two kinds. The first kind consists of those that are known to be possible in the handler method of any metaobject (see figure 4.7 case (d)); they are explicitly declared in the `throws` clause of the default handlers (in the metaobject classes in the Iguana/J class library). These include exceptions thrown if the metaobject attempts a base-level operation for which it has insufficient privilege (e.g invocation of a private method), or if the metaobject passes invalid arguments to a base-level operation. If one of these occurs it is rethrown in the base-level program, in the same way as an unchecked exception. The second kind of checked exception that must be dealt with consists of those that a meta-level programmer wishes to throw explicitly within a handler method (by using `throw`) but are not declared in the method's `throws` clause (see figure 4.7 case (e)). These exceptions cannot be added to the `throws` clause (see above). To deal with this, handler methods in all Iguana/J metaobject classes are declared to throw `InvocationTargetException`, even though the field access metaobject classes do not normally need to deal with this exception. This allows a meta-level programmer to throw any exception, wrapped in an `InvocationTargetException`, from any handler method. It will be automatically unwrapped and rethrown in the base-level program.

- Exceptions thrown by the Iguana/J runtime engine itself.

  The Iguana/J runtime engine may throw an exception or error when it encounters an operational problem such as inability to load a protocol class required for an association, a metatype rule violation, or an

[14] Checked exceptions are those that the Java compiler requires be dealt with explicitly, using a try/catch construct or a throws clause. Unchecked exceptions do not need this, and are recognised by the compiler because they are subclasses of `java.lang.RuntimeException`.

internal problem such as a data inconsistency. These exceptions and errors are thrown in the context of the current Java thread, so they are handled by the active methods of the thread according to the normal Java exception handling rules. An exception that occurs when the runtime engine has been activated via the JIT compiler interface (i.e. as a result of trapping a class load) is thrown in the context of the method that attempted the class load. An exception that is thrown when the runtime engine has been activated via an intercepted base-level operation is thrown in the context of the method containing the intercepted operation.

Some of these exceptions will be rethrown in the context of a method that does not expect them. For example an exception thrown by a metaobject during reification, or an exception thrown by the runtime engine during trapping of a class load, will be rethrown in the context of a base-level method that does not expect them. This does not cause any difficulty for the base-level program because at runtime all unexpected exceptions are treated as unchecked. In fact, only the Java compiler draws a distinction between checked and unchecked exceptions; the Java virtual machine does not differentiate between them (see [Gos00] §11.2 and [Lin99] §2.16.4). Any unexpected exception, whether checked or unchecked, will be thrown back to the invoking method. Thus, exceptions thrown by a metaobject or by the Iguana/J runtime engine are likely to cause a system abort unless some enclosing try/catch block catches all exceptions by specifying `java.lang.Exception`.

4.15 Summary

This chapter described the implementation of the Iguana/J architecture, focusing on its support for unanticipated dynamic adaptation and the novel technique used to integrate it with the JVM. It allows runtime manipulation of interception hooks and association links between base and meta-level components without pre-runtime preparations. It implements the Iguana/J metatype structure, and handles protocol association consistently in complex situations involving shared and retained metaobjects.

The implementation hides much complexity from the programmer, presenting a simple set of interfaces for initiating and manipulating association. Issues that are dealt with transparently, without involving the programmer, include variations in return types from reified operations, avoidance of infinite recursive reflection, and handling of exceptions during reified operations. The implementation supports association of protocols with almost all classes loaded by the JVM. This includes most core system classes, but excludes a small number of classes loaded very early in the life of the JVM.

The implementation meets the objectives set out for it. Interception mechanisms can be inserted and removed at runtime, metaobject protocols can be associated with arbitrary base-level classes (subject to metatype rules), and a clear separation is maintained between base-level and meta-level components even while they are associated. The implementation also minimises its effect on the execution time of the base-level program. This is achieved by inserting interception hooks only when and where they are needed, leaving non-intercepted operations to execute without any slowdown at all.

Chapter 5

Evaluation

> *Increasingly, people seem to misinterpret complexity as sophistication, which is baffling.*
>
> -Niklaus Wirth.
> Address to Conference on Innovation and Technology
> in Computer Science Education, Denmark, June 2002.

This chapter evaluates the Iguana/J architecture against the objectives introduced in chapter 1. Those objectives include providing a structured approach to unanticipated dynamic adaptation, supporting a strong separation of concerns and providing solutions to some acknowledged challenges in the field of dynamic adaptation. The principal characteristics of Iguana/J that contribute to achieving its objectives are its treatment of behavioural change as an inherited metatype, imposition of metatype constraints, specialisation of metaobjects at runtime, support for context examination, and a simple class library. This chapter evaluates Iguana/J by demonstrating its solutions to a set of examples based on acknowledged challenge problems and issues in the field of unanticipated dynamic adaptation, and by comparing the ability of Iguana/J to address them with that of other architectures.

The evaluation demonstrates that the constraints and structure imposed by Iguana/J's metatypes help to address the need for control and discipline in the use of reflection that has been noted in the literature [Mae87] [Mit97] [Men97]. Although SOM [For98] introduced constraints on the use of reflection, Iguana/J applies constraints to behavioural rather than structural changes. Iguana/J's constraints operate in a more dynamic setting, supporting a strong separation of concerns with no anticipation of changes and no invasive modifications to the program. The evaluation shows that metatypes can help to control the scope of unanticipated dynamic adaptation, while still supporting flexible and useful adaptation. The examples and comparisons in this chapter illustrate the safer environment that Iguana/J provides for applying behavioural change, in which relationships between the behaviour of related classes and objects remain predictable under all dynamic adaptations.

5.1 Evaluation approach

The evaluation focuses on the characteristics of Iguana/J that are not addressed fully by existing architectures supporting unanticipated dynamic adaptation, or for which Iguana/J provides some improvement. The evaluation also demonstrates that for many challenges and issues Iguana/J provides a simpler and more structured approach to unanticipated dynamic adaptation.

The evaluation is carried out under six headings:

- Reusability:
  Support for designing behavioural change components to be reusable (in their compiled form) with arbitrary components in a program.
- Context sensitivity:
  Support for behavioural change components to make runtime decisions based on execution context.
- Inheritance anomaly:
  Support for maintaining under inheritance the encapsulation of separately implemented non-functional concerns such as synchronisation.
- Separation of concerns:
  Support for separate design, implementation, and deployment of behavioural change components, independently of each other and of program components.
- Inheritance of change:
  Support for inheriting behavioural change within an adapted program.
- Performance cost:
  The effect on a program's execution time of the various mechanisms of the dynamic adaptation architecture.

The first five of these represent areas of particular challenge for dynamic adaptation, and the sixth heading is the performance cost of using the architecture. Under each of the five challenge headings a representative example is used to illustrate the challenge. The examples for the first three are derived directly from published challenge problems, and the examples for the other two have been created to demonstrate those particular issues. These areas of challenge have been specifically identified as important for any architecture that attempts to support design and use of separate behavioural change components. The most notable catalogue of such challenges is the report of the *Workshop on Aspects and Dimensions of Concern* (Cannes, June 2000) by Tarr, D'Hondt, Bergmans and Lopes [Tar00]. The two-day workshop specifically set out to identify goals, requirements and issues for approaches to advanced separation of concerns. The report details requirements and challenge examples representing the issues identified as outstanding or difficult. For this evaluation a set of issues relating to runtime support for dynamic adaptation has been selected from the report, and representative sample problems used to demonstrate how Iguana/J provides a combination of solutions that has not been available up to now.

For each evaluation heading a solution was developed for a representative problem using Iguana/J, and compared to solutions using other approaches. The other architectures used for the comparisons are MetaXa, Guaraná, and PROSE, which were each reviewed in detail in sections 2.5, 2.6 and 2.7. A table listing and

comparing characteristics of Iguana/J and the other three architectures is given in section 5.8 at the end of this chapter.

The six evaluation headings are covered in the next six sections of this chapter. Each of the sections begins with a short outline of the challenge, and the example used to illustrate it. This is followed by a description of a solution using Iguana/J, and by an outline of solutions using other architectures for unanticipated dynamic adaptation. To avoid overly complex code listings some of the examples have been simplified by omitting some code statements not directly related to the attribute or feature being evaluated, and by omitting exception handling.

5.2 Reusability

Reusability of components representing behavioural change allows them to be used with arbitrary program components, as discussed in section 1.2.1. For maximum reusability a behavioural change component should not be tied, even in its compiled form, to any particular program component. Where a change component requires specific information about the program component it is adapting, that information should be supplied at the time the association is created between the two components [Tar00]. Reusability of behavioural change components supports independent implementation of general concepts such as persistence, synchronisation, distribution or access control that can be used with any class or object in a program

5.2.1 The reusability challenge

The example for this challenge is adapted from [Tar00], where it is used to represent the need for reusability in systems that support dynamic association of behavioural change with a running program. The general requirement for the example is to design a component that implements synchronisation constraints and is reusable with any class. To be reusable the component implementing synchronisation should have no dependency on the names of classes or methods with which it will be used. The example requires that a single component be used to make either of two unrelated classes, one implementing a stack and the other a queue, thread safe (see figure 5.1). Both classes have a method to write a data value and a method to read a value, but there is no naming consistency between them. In addition, the data items handled by the classes are of different types, unrelated to each other, and both classes may have other methods that are not to be synchronised. To make `Stack` or `Queue` thread safe, only one write or read should be allowed at a time on any instance of either of them. This behavioural change is to be implemented in a single separate component and applied to the two classes during runtime.

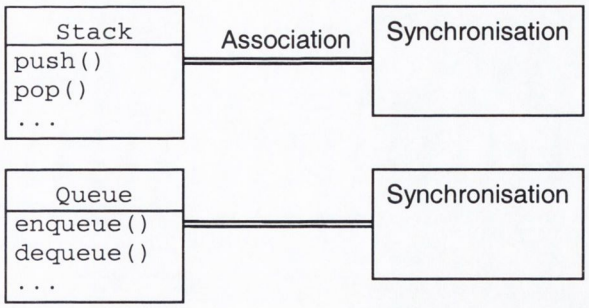


Figure 5.1: One behavioural change component should be reusable with different classes.

5.2.2 The Iguana/J solution

Iguana/J addresses reusability by supporting association of any protocol with any class or object, and by supporting specialisation of protocols (and their metaobjects) at association time via protocol parameters. A solution to the reusability example, using protocol parameters to specialise a single metaobject class to apply synchronisation to either the `Stack` or the `Queue` class, is given in figures 5.2, 5.3 and 5.4.

The listing in figure 5.2 is a metaobject class called `Sync` that reifies method invocation and applies synchronisation to named methods. Its constructor (lines 4, 5, 6) accepts an array of strings giving the names of the methods to be synchronised, and this is stored in a private field for later use. The constructor is called automatically as part of the protocol association process. When a method invocation is passed to the metaobject it is handled by the `execute()` method (lines 7-15 of figure 5.2). It uses Java reflection to get the name of the method being invoked (line 9). If the method name matches any of those given in the array passed to the constructor then the method is executed within a synchronised block, taking a lock on the target object. Otherwise the method invocation proceeds without synchronisation (line 14).

To be used for dynamic adaptation the metaobject in figure 5.2 must be declared in an Iguana/J protocol, as shown in figure 5.3. The protocol accepts an array of strings as its only parameter and passes it to the metaobject, thereby matching the argument types expected by the metaobject's constructor. The protocol must be compiled with the Iguana/J protocol compiler to produce the protocol class that represents it at runtime. The protocol and metaobject are reusable in their compiled form because they contain no embedded static references to any base-level class or method. The protocol can be used to apply synchronisation to any set of methods of any class by associating it with the class and giving an array of the names of the methods as a parameter value. Association of the protocol with the `Stack` and `Queue` classes can be initiated dynamically by executing the statements shown in figure 5.4.

```
[1]  // Metaobject class:
[2]  public class Sync extends MExecute {
[3]    private String[] names;     // Method names.
[4]    public Sync(String[] s) {
[5]      names = s.clone();
[6]    }
[7]    public Object execute(Object o, Object[] args, Method m)
[8]                    throws InvocationTargetException {
[9]      if(inNames(m.getName())) {
[10]       synchronized(o) {
[11]            return(proceed(o,args,m));
[12]       }
[13]     }
[14]     else return(proceed(o,args,m));
[15]   }
[16]   private boolean inNames(String mname) {
[17]     for(int i=0; i < names.length; i++) {
[18]       if(mname.equals(names[i]) return(true);
[19]     }
[20]     return(false);
[21]   }
[22] }
```

Figure 5.2: An Iguana/J metaobject class to implement synchronisation.

```
protocol SyncProtocol(String[] s) {
  reify Invocation: Sync(s);
}
```

Figure 5.3: An Iguana/J protocol using the synchronisation metaobject.

```
//...
Object[] params = new Object[1];
params[0] = new String[] {"pop", "push"};
Meta.associate("Stack", "SyncProtocol", params);
//...
params[0] = new String[] {"dequeue", "enqueue"};
Meta.associate("Queue", "SyncProtocol", params);
//...
```

Figure 5.4: Code fragments to initiate association of `SyncProtocol` with `Stack` and `Queue`.

5.2.3 Other solutions

MetaXa, Guaraná, and PROSE all support reusability of metaobjects or aspects, using constructor arguments to specialise at association or weaving time in much the same way as Iguana/J. None of the three architectures require any static references between the program and the behavioural change component, so the behavioural change components are fully reusable. In MetaXa the method names can be passed to the

constructor of the metaobject when the association is being initiated. For example, initiation of association of a metaobject `SyncMetaobject` with the class `Stack` can be done by executing the statements[15]:

```
MetaObject mo = new SyncMetaobject("push", "pop");
Class cls = Class.forName("Stack");
mo.attachClass(cls);
mo.registerEventMethodCall(cls);
```

The convention in MetaXa is to include the attachment and registration operations in the constructor of the metaobject. If that is done in the `SyncMetaobject` class it makes possible a simpler approach to initiating the association, where the four lines above are replaced by:

```
new SyncMetaobject(Class.forName("Stack"), "push", "pop");
```

In Guaraná the metaobject is created first, passing the names of the base-level methods as constructor arguments, and then the `reconfigure()` method is used to associate it with the base-level class:

```
Guarana.reconfigure(Class.forName("Stack"),
          null, new SyncMetaobject("push","pop"));
```

The second argument to `reconfigure()` is a reference to the existing metaobject of the class, if there is one. This is part of Guaraná's meta-level security model (see section 2.6.1).

In PROSE the aspect class can be designed to expect the names of the class and the two methods as arguments to its constructor. Values for them can be passed to the aspect class at runtime. For example, if the aspect class is called `SyncAspect` the weaving might be initiated by executing:

```
ExtensionSystem.insert(new SyncAspect("Stack", "push", "pop"));
```

The aspect constructor creates an instance of the crosscut class, passing the arguments on to it for use in the PROSE specialiser. In MetaXa and Guaraná the approach is to create a metaobject, specialising it for the method names, and then to associate the metaobject with the base-level class. In PROSE both the name of the base-level class and the names of the methods in the class are used together in the specialiser of the crosscut. The PROSE specialiser for this example would look like:

```
setSpecializer(DeclarationS.inClass(c).AND(
            MethodS.named(m1).OR(
            MethodS.named(m2))));
```

where values for the strings `c`, `m1` and `m2` are received via arguments to the crosscut constructor.

In MetaXa synchronisation can be implemented in the metaobject in a manner similar to Iguana/J, simply by enclosing the execution of the reified operation in a Java `synchronized` block. A MetaXa metaobject tends to be complex because it has ten separate handler methods for reifying invocation of methods with different return types[16]. This means that the detection of invocation of the methods to be synchronised must

[15] Attaching a metaobject to a class is not documented fully in the published material for MetaXa, so this example is based on an interpretation of the available information.
[16] Eight primitive types, `Object`, and void.

be in each of the ten handler methods, as the metaobject design cannot assume any particular return type for the methods that it might be used to reify.

This particular example, requiring synchronisation, is difficult for Guaraná and PROSE because they implement separate handling of 'before' and 'after' intercepted operations. In these two architectures it is not possible to use a simple metaobject, or aspect, to apply synchronisation to a Java operation because the metaobject cannot act as a guard spanning both the invocation and the return of the method. There is no code statement in the handler method that represents carrying out the intercepted invocation, so there is no statement that can be enclosed in a Java `synchronized()` block.

In Guaraná it is possible to construct a rather complex alternative, by creating a generalised wrapper method that implements synchronisation. The wrapper simply encloses a call to the original method in a Java `synchronized` block. The 'before' handler can redirect the operation to this wrapper method by substituting it for the original target method, but it also needs to manipulate arguments to ensure that the wrapper calls the correct original method. This technique also requires some manipulation of the target object, as the wrapper method is in an instance of a different class. In PROSE this alternative wrapper technique is not available, as PROSE does not support redirection of intercepted operations, and therefore the synchronisation behaviour cannot be implemented as a PROSE aspect.

5.2.4 Summary of reusability

Iguana/J, along with the three architectures used for comparison, supports reusable behavioural change components. They all allow specialisation via constructor arguments when the metaobject or aspect instance is created, so the metaobject and aspect classes can be designed to be independent of association with any particular program class. However, the synchronisation example used here poses difficulties for Guaraná and PROSE, and demonstrates some limitations of the separated handling of 'before' and 'after' that they support.

5.3 Context sensitivity

Context sensitivity is the ability of a component implementing behavioural change to examine the execution context at runtime and to make decisions based on that examination. Context sensitivity has been proposed as an important requirement for architectures supporting separate implementation of behavioural change [Tar00]. It allows a change to be applied to a class or object depending on how, or by what other component, the class or object is being used. It requires support for examining arguments and return values of the reified operation, and it requires support for examination of the classes and methods in the invocation chain at the time of the reified operation. This last requirement implies support for examination, in some form, of the runtime stack.

```
public class List {
   public void addOne(ListItem l) {
    //...
   }
   public void addAll(ListItem[] la) {
      for(int i=0; i < la.length; i++) {
         addOne(la[i]);      // addAll() uses addOne().
      }
   }
}
```

Figure 5.5: An outline of the `List` class, in which one public method uses another.

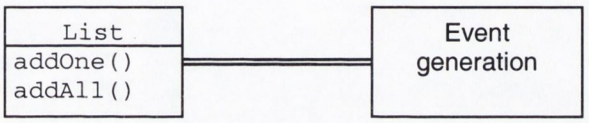


Figure 5.6: Event generation as a behavioural change component.

5.3.1 The context sensitivity challenge

The example used is adapted from one originally proposed by Brichau [Bri00] to represent this issue (referred to as the 'jumping aspects' problem). An existing `List` class has two public methods called `addOne()` and `addAll()` that add elements to the list. The method `addAll()` adds a set of elements, and uses `addOne()` repeatedly to do this (see figure 5.5). It is required to associate behavioural change with the class dynamically (see figure 5.6) so that a notification event is generated when one or more elements are added to the list. The simplest way to do this is to associate the event generation behaviour only with the `addOne()` method, but this will cause multiple events to be generated when `addAll()` is invoked. A more efficient approach is to associate the event generation behaviour with both methods, and to arrange that a call to `addOne()` only generates an event if it was not invoked from `addAll()`.

It is assumed, for this example, that an instance of class `Handler` may be used to pass events for processing, and that it has a `notify()` method for this purpose.

5.3.2 The Iguana/J solution

Iguana/J supports examination of the Java stack using the `CallStack` class, as already described in section 3.6. `CallStack` supports examination (but not modification) by a metaobject of certain contents of the stack, notably the names of methods invoked. If a method name is found in the stack then that method is part of the current execution context. That is, the method has been invoked and is carrying out, directly or via further invocations, the operation being reified.

The Iguana/J solution to the context sensitivity example consists of a generalised metaobject class, called `GenEvent`, that detects invocations of a specified method that have not occurred from a second specified method (see figure 5.7). The two method names (`addOne()` and `addAll()` for this example) are expected as arguments to the class's constructor. They are supplied via protocol parameters at association time, and are stored for later use (lines 5 and 6). While it is possible to embed the two method names in the code of the metaobject class, passing them as parameters maximises the reusability of the metaobject class. In line 11 of figure 5.7 the metaobject class proceeds with the method invocation that is being reified, regardless of the name of the method. After the invocation returns, the metaobject class checks the name of the method whose invocation is being reified (line 12), and examines the stack (line 13). If the reified invocation is of the first named method and the invocation is not in the context of the second named method, then it generates and sends an event. It also sends the event for all invocations of the second named method (line 14).

Figure 5.8 contains a declaration of the protocol that uses the `GenEvent` metaobject class to reify method invocation. Figure 5.9 shows how the protocol can be associated with the `List` class so that an event is generated when `addOne()` or `addAll()` is called, but not when `addOne()` is called from `addAll()`.

```
[1]  public class GenEvent extends MExecute {
[2]    private String name;
[3]    private Handler handler;
[4]    public GenEvent(String m1, String m2) {
[5]      name1 = m1;
[6]      name2 = m2;
[7]      handler = new Handler();
[8]    }
[9]    public Object execute(Object o,Object[] args,Method m)
[10]                          throws InvocationTargetException {
[11]     Object result = proceed(o,args,m);
[12]     if((m.getName().equals(name1) &&
[13]                          !getCallStack().containsCallTo(name2))
[14]                          ||    m.getName().equals(name2)) {
[15]       Event e = new Event(o);
[16]       handler.notify(e);      // Pass the event.
[17]     }
[18]     return(result);
[19]   }
[20] }
```

Figure 5.7: An Iguana/J metaobject class that generates an event depending on context.

```
protocol EventProtocol(String m1, String m2) {
  reify Invocation: GenEvent(m1, m2);
}
```

Figure 5.8: A protocol that includes the event generator metaobject.

```
//...
Object[] params = new Object[] {"addOne", "addAll"};
Meta.associate("List", "EventProtocol", params);
//...
```

Figure 5.9: A fragment of code to associate the protocol with the `List` class.

5.3.3 Other solutions

None of the three architectures used for this evaluation support context examination, and therefore none of them can address this challenge problem. The example requires support for examining the execution context at runtime, which implies examination of the contents of the stack. MetaXa, Guaraná, and PROSE do not provide any means for examining the stack. There are no apparent architectural reasons why the first two do not. As both MetaXa and Guaraná are implemented by creating or modifying a JVM there is no technical barrier to providing a structural reification class to represent the stack. PROSE is implemented outside the JVM, using the JVMDI to communicate with it, so it is probably difficult to implement the low-level support needed for a stack reification class. While the JVMDI supports some stack operations it does not directly support detailed examination of it.

5.3.4 Summary of context sensitivity

Iguana/J is the only unanticipated dynamic adaptation architecture supporting context sensitivity, and therefore it is the only one that can address the 'jumping aspects' problem. The `CallStack` class reifies the Java stack, and effectively extends the structural reflection already supported by the Java reflection classes.

5.4 Inheritance anomaly

The inheritance anomaly, described already in section 1.3.3, has been identified in dynamic adaptation as a difficulty in inheriting and extending behavioural change that represents a non-functional concern for related base-level classes [Tar00]. In brief, behavioural change applied to a class should be inherited by its subclasses, and it should be possible to extend the inherited behavioural change without needing to modify or restate the superclass's behavioural change component.

5.4.1 The inheritance anomaly challenge

The example selected to represent the inheritance anomaly here is adapted from the buffer example used by Matsuoka to illustrate the original anomaly [Mat93]. A version of this example was also used by Tarr to illustrate the issue for dynamic adaptation [Tar00]. The example is based on a class called `Buffer` that implements a simple first-in-first-out buffer, as shown in figure 5.10. The class has a subclass, called

PopBuffer, as shown in figure 5.11. In the Buffer class it is required that the two methods put() and get() be made thread safe, which requires that they be made self exclusive and mutually exclusive[17]. These behavioural changes are to be implemented as a metaobject class. It is also required, separately, to apply constraints to make the popLast() method of the PopBuffer class thread safe. The constraints are required because the popLast() method cannot be executed safely with either the get() method (because popLast() and get() both remove items from the buffer) or the put() method (because popLast() and put() both alter the write pointer). The constraints for the popBuffer class must provide mutual exclusion between popLast() and get(), and between popLast() and put(), and must make the popLast() method self exclusive. The challenge is to implement the synchronisation constraints for the functionality added by the subclass, popBuffer, independently of the synchronisation constraints for the functionality defined by the superclass, Buffer. The requirements are illustrated in figure 5.12.

```
public class Buffer {
   protected int wr = 0;
   protected int rd = 0;
   protected final int SIZE = 20;
   protected Object[] data = new Object[SIZE];
   public boolean put(Object d) {
      if(wr-rd == SIZE) return false;
      data[wr%SIZE] = d;
      wr++;
      return true;
   }
   public Object get() {
      if(rd == wr) return null;
      Object result = data[rd%SIZE];
      rd++;
      return(result);
   }
}
```

Figure 5.10: The basic Buffer class.

```
class PopBuffer extends Buffer {
   public Object popLast() {
      if(rd == wr) return null;
      Object result = data[--wr];
      return(result);
   }
}
```

Figure 5.11: The subclass adds a popLast() method that uses shared fields.

[17] In fact, in the implementation shown in figure 5.10 mutual exclusion is not required if the statements in both methods are guaranteed to be executed in source code order. One method alters only the read pointer and the other alters only the write pointer, and in both cases the pointer is not altered until after the buffer operation is complete. For any interweaving of the two methods they will both see consistent data at all times.

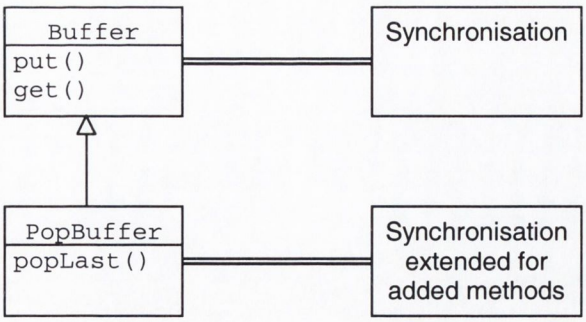


Figure 5.12: Synchronisation should be inherited and extended for a subclass.

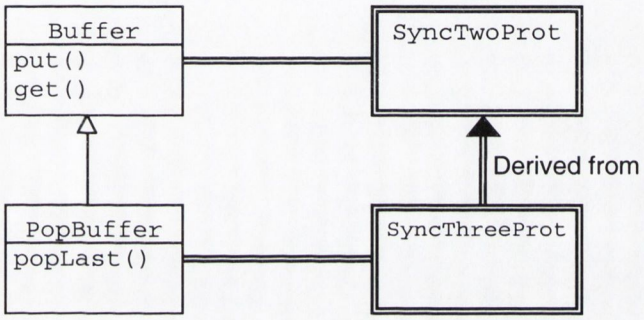


Figure 5.13: Inheritance and extension of behavioural change using Iguana/J

5.4.2 The Iguana/J solution

The Iguana/J solution to this example is to create two protocols, one derived from the other, to be associated with `Buffer` and `PopBuffer` as illustrated in figure 5.13. To comply with Iguana/J's metatype rules the protocol associated with `PopBuffer`, `SyncThreeProt`, must be derived from the protocol associated with `Buffer`, `SyncTwoProt`. The `SyncTwoProt` protocol uses a single metaobject class, `SyncTwo`, to reify invocation and to apply synchronisation to the two methods `put()` and `get()`. The metaobject class `SyncTwo` is listed in figure 5.14 and the protocol declaration is given in figure 5.15. By inserting a single synchronisation guard around execution of `put()` and `get()` the metaobject causes entry to the two methods to be mutually and self exclusive.

```
[1]  public class SyncTwo extends MExecute {
[2]    private String name1;
[3]    private String name2;
[4]    private Object lock = new Object();
[5]
[6]    public SyncTwo(String name1, String name2) {
[7]      this.name1 = name1;
[8]      this.name2 = name2;
[9]    }
[10]
[11]   public Object execute(Object o, Object[] args, Method m)
[12]                                  throws InvocationTargetException {
[13]     Object result;
[14]     String s = m.getName();
[15]     if(s.equals(name1) || s.equals(name2)) {
[16]       synchronized(lock) {
[17]            result = proceed(o, args, m);
[18]       }
[19]     }
[20]     else {
[21]       result = proceed(o, args, m);
[22]     }
[23]     return(result);
[24]   }
[25] }
```

Figure 5.14: A metaobject class to apply mutual and self exclusion to two methods.

```
protocol SyncTwoProt(String m1, String m2) {
  reify Invocation: SyncTwo(m1, m2);
}
```

Figure 5.15: A protocol that reifies invocation using the `SyncTwo` metaobject class.

Association between the protocol `SyncTwoProt` and the class `Buffer` can be dynamically initiated by executing the statements:

```
Object[] params = new Object[] {"put", "get"};
Meta.associate("Buffer", "SyncTwoProt", params);
```

This association is inherited by all subclasses of `Buffer`, including `PopBuffer`, but extra synchronisation constraints must be added for the `popLast()` method of `PopBuffer`. The requirement is that `popLast()` be made self exclusive, and that `popLast()` be made mutually exclusive with each of `put()` and `get()`. These constraints are implemented by the `SyncThree` metaobject class listed in figure 5.16. The metaobject class implements self exclusion for one method, and mutual exclusion between that method and each of two others. It uses a semaphore to implement the mutual exclusion in preference to the simpler approach of wrapping synchronisation blocks around the `proceed()` calls because that would have had the side effect of implementing self exclusion for the second two methods. This is not required for the `PopBuffer` class. Although self exclusion is required for each of `put()` and `get()` it should properly be implemented in the metaobject class to used with the `Buffer` class, and not in the metaobject class to be used with the `PopBuffer` class.

```
[1]  public class SyncThree extends MExecute {
[2]    private String name1;
[3]    private String name2;
[4]    private String name3;
[5]    private Object lock = new Object();
[6]    private int sema = 0;
[7]
[8]    public SyncThree(String name1, String name2,
[9]                     String name3) {
[10]     this.name1 = name1;
[11]     this.name2 = name2;
[12]     this.name3 = name3;
[13]   }
[14]
[15]   public Object execute(Object o, Object[] args, Method m)
[16]                    throws InvocationTargetException {
[17]     Object result = null;
[18]     String s = m.getName();
[19]     if(s.equals(name1)) {
[20]       synchronized(lock) {
[21]         while(true) {
[22]           if(sema == 0) {
[23]             break;
[24]           }
[25]           else {
[26]             lock.wait();  // Releases lock while waiting.
[27]           }
[28]         }
[29]         result = proceed(o, args, m);
[30]       }
[31]     }
[32]     else if(s.equals(name2) || s.equals(name3)) {
[33]       synchronized(lock) {
[34]         sema++;
[35]       }
[36]       result = proceed(o, args, m);
[37]       synchronized(lock) {
[38]         sema--;
[39]         lock.notify();
[40]       }
[41]     }
[42]     else {
[43]       result = proceed(o, args, m);
[44]     }
[45]     return(result);
[46]   }
[47] }
```

Figure 5.16: A metaobject class that provides mutual exclusion between one method and each of two others.

```
protocol SyncThreeProt(String pop, String rd, String wr)
                                    : SyncTwoProt(rd, wr) {
  reify Invocation: SyncThree(pop, rd, wr);
}
```

Figure 5.17: A derived protocol that reifies invocation using `SyncThree` and `SyncTwo`.

The SyncThree metaobject is used in a second protocol, called SyncThreeProt (see figure 5.17). This protocol is derived from SyncTwoProt and therefore it reifies method invocation with a composition of the two metaobject classes, SyncTwo and SyncThree. It can be associated with the PopBuffer class by executing:

```
Object[] params = new Object[] {"popLast", "put", "get"};
Meta.associate("PopBuffer", "SyncThreeProt", params);
```

This association provides self exclusion for popLast() and provides mutual exclusion between popLast() and each of the other two methods. Because the protocol SyncThreeProt is derived from the protocol SyncTwoProt it includes the SyncTwo metaobject, which provides self and mutual exclusion for put() and get(). The synchronisation constraints imposed by the SyncThree metaobject class concern only the new issues introduced by the method added in the base-level subclass, PopBuffer. SyncThree does not restate or depend upon any of the synchronisation constraints imposed by SyncTwo. The constraints implemented by SyncTwo for the Buffer class could be modifed (statically) if required, without affecting the constraints implemented by SyncThree for the PopBuffer subclass.

The Iguana/J solution to the inheritance anomaly is essentially the guard solution discussed by Matsuoka [Mat93] and by McHale [Mch94]. Each metaobject represents another guard, and metaobject composition represents a nested guard structure. The guards (metaobjects) are implemented and inherited independently of the implementation of the program on which they operate.

5.4.3 Other solutions

Like Iguana/J MetaXa places execution of 'before' code, the reified operation, and 'after' code under the control of a handler method in the metaobject. This supports the use of a Java synchronized() block in the metaobject in a manner similar to the Iguana/J metaobject shown in figure 5.14. In MetaXa, just as in Iguana/J, two metaobjects can be created, one implementing the synchronisation required for the put() and get() methods, and one implementing the synchronisation required for the popLast() method. However, MetaXa does not support inheritance of metaobjects at the base-level. This means that a metaobject associated with the Buffer class is not automatically associated with the PopBuffer class. To create a solution to the challenge problem the first metaobject must be associated with Buffer, and then the first and the second metaobject must be separately associated with PopBuffer. MetaXa automatically composes the two metaobjects so that the 'before' of each is executed, then the reified operation, then the 'after' code of each metaobject. This achieves the same nested guard structure used in the Iguana/J solution to this challenge.

As this example is based on synchronisation the same difficulties arise for PROSE and Guaraná as in the reusability challenge (see section 5.2.3). This is because it is not possible to enclose an intercepted operation in a synchronized block when the architecture separates the handling of 'before' and 'after' code. In Guaraná it is possible to construct an alternative using a wrapper method, as for the reusability example. One metaobject must be created to implement the synchronisation constraints for the Buffer class by redirecting invocations of put() and get() to the synchronising wrapper method. A second separate metaobject must

be created to implement the synchronisation constraints for the `PopBuffer` class. Like MetaXa, Guaraná does not support inheritance of metaobject association and therefore the two metaobjects must be explicitly associated with `PopBuffer`. Because Guaraná's composition model is separate delegation of 'before' and 'after' events, a complex wrapping and delegation structure must be constructed to achieve the nested guards for this example.

Although PROSE does support inheritance of aspects (if it is explicitly included in the specialiser conditions) the issue of separated 'before' and 'after' handlers makes it difficult to provide a solution to the inheritance anomaly. PROSE does not support redirection of the original operation, so it is not possible to work around the separated 'before' and 'after' using the technique suggested above for Guaraná. A different inheritance anomaly example might allow PROSE to avoid this particular limitation. However, without support for redirecting or blocking intercepted method invocations it is very difficult for the aspect to influence the invocation of the intercepted method. This means that PROSE cannot be used to address most of the examples commonly used to illustrate the inheritance anomaly, because they all involve blocking or redirecting invocation.

5.4.4 Summary of inheritance anomaly

Only Iguana/J and PROSE support inheritance of behavioural change. Only Iguana/J and MetaXa support the 'guard' construct needed to apply the synchronisation in the example. The technique of redirection to a wrapper method that can be used with Guaraná is rather complex and not really practical, particularly when composing multiple metaobjects that all use that technique.

Iguana/J supports both inheritance and the nested guard construct, and therefore it is the only one of these architectures that provides a solution to the inheritance anomaly challenge described in [Tar00].

5.5 Separation of concerns

The degree of separation between the program to be adapted and the components implementing behavioural change influences the usefulness of an architecture for unanticipated dynamic adaptation, as discussed in section 1.2.1. A strong separation of concerns is identified in [Tar00] as a critical requirement for architectures supporting dynamic adaptation. It helps to support reusability and unanticipation of behavioural change, and supports the association of arbitrary behavioural changes with arbitrary components of a program at runtime. To achieve a strong separation of concerns there must be no need for references from the program to the change components, or from the change components to the program, before they become associated during runtime. Initiation of an association necessarily requires the use of references to the behavioural change component and to the affected program component. Therefore if the association is initiated by code embedded in the program or in a change component the presence of the references to the other component weakens the separation of concerns. It is apparent that a strong separation of concerns can only be achieved when the association is initiated by an entity that is somehow outside both the program and the behavioural change components (see figure 5.18). This leads to a difficulty. If invasive changes to the

program are to be avoided, so that it does not anticipate the changes in any way, then how is the entity that initiates associations to be activated?

Dynamically initiating an association between a program component and a behavioural change component usually implies invocation of an appropriate method. However, this can only be carried out by inserting it into the progam (thereby anticipating change), or by including it in a component that must itself be inserted into the program. In either case an 'invasive' change [Cza00] must be made to the program before runtime to cause association statements to be executed. It is difficult to cause execution of code statements that dynamically initiate associations without anticipating the behavioural change and weakening the separation of concerns.

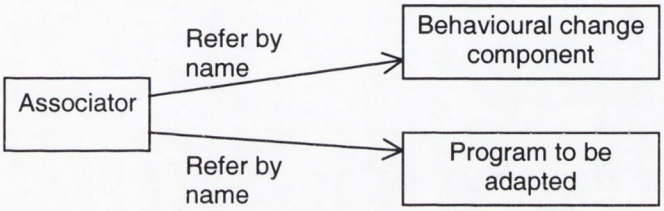


Figure 5.18: For a strong separation of concerns the associator must be independent.

5.5.1 The separation of concerns challenge

Almost any behavioural change can be used to represent this challenge, because the issue is to do with the association process rather than the nature of the behavioural change. The example selected is to update a program to use a new version of a third party communications class. The old class is called `Channel`, and the new class is called `NewChannel` and is a subclass of `Channel`[18]. The new class has the same method names and signatures as the old one. It is required that all operations on instances of the old class be redirected to an instance of the new class (see figure 5.19).

The challenge is to apply this change to a program during runtime, without modifying the program before runtime to support the change. The solution should exhibit the characteristics of a strong separation of concerns, with no invasive changes or anticipation in the program.

5.5.2 The Iguana/J solution

There are two parts to providing a solution for this example. The first is to create the metaobject class and protocol implementing the behavioural change, and the second is to associate the protocol with the program. The second part, association, addresses the challenge of separation of concerns, but the first part must be completed before association can be examined.

[18] The new class is specified as a subclass of the old one to avoid complicating the example with issues not directly related to separation of concerns (e.g. argument conversion for a new method).

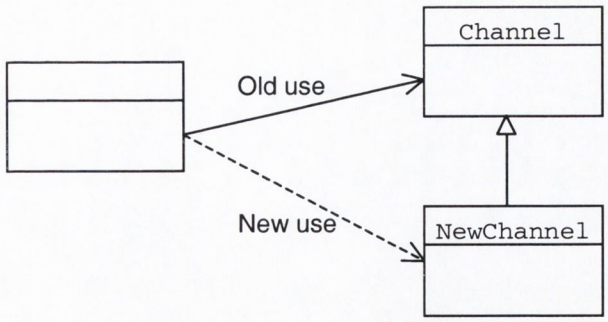


Figure 5.19: Access to instances of `Channel` must be redirected to instances of `NewChannel`.

```
[1]  class Redirect extends MExecute {
[2]    private Class newCls;
[3]    private Object newObj;
[4]
[5]    public Redirect(String cName) {
[6]      newCls = Class.forName(cName);
[7]    }
[8]    public Object execute(Object o, Object[] args, Method m)
[9]                          throws InvocationTargetException {
[10]     if(newCls.isInstance(o)) return(proceed(o, args, m));
[11]     if(newObj == null) {
[12]       newObj = newCls.newInstance();
[13]       // ... Use Java reflection to transfer state from
[14]       //     old to new object.
[15]     }
[16]     Method nm = newCls.getMethod(m.getName(),
[17]                                  m.getParameterTypes());
[18]     return(proceed(newObj, args, nm));
[19]   }
[20] }
```

Figure 5.20: A metaobject class to redirect method invocations.

```
protocol RedirectProt(String cls) {
  reify Invocation: Redirect(cls);
}
```

Figure 5.21: A protocol that uses the `Redirect` metaobject class.

The Iguana/J approach to updating a program's use of a class is to reify all accesses to each instance of the old class and to redirect them to instances of the new class. A metaobject class, called `Redirect`, to do this is listed in figure 5.20. The metaobject is intended to be used in a protocol associated with each instance of the old class, with local scope so that there is one metaobject per instance of the old class. The metaobject class `Redirect` makes extensive use of the Java reflection classes. When it is first used it uses Java reflection to create an instance of the new class (line 12), and it copies state information from the old instance

(omitted from figure 5.20 at line 13). It uses Java reflection again to find the `Method` object corresponding to the method with the same signature in the new class (lines 16 and 17) and redirects the reified operation to this new method of the new instance (line 18). To allow for parts of the program that may use instances of the new class directly the metaobject must detect such access and avoid redirecting it to a second instance of the new class. This is done in line 10 of figure 5.20. A declaration of a protocol, `RedirectProt`, that uses the `Redirect` metaobject is given in figure 5.21.

While an association with the protocol could be dynamically initiated by using `Meta.associate()`, this would require invasive modification of the program in order to cause it to be executed. Statically initiated association, by a declaration in Iguana/J's configuration file, supports non-invasive adaptation with a stronger separation of concerns. By including a declaration such as:

```
Channel ==> Redirect("NewChannel");
```

in the Iguana/J configuration file the association, and therefore the behavioural change, can be carried out without making any changes at all to the original program. The static configuration file acts as a third entity, separate from the original program and the behavioural change components, that plays the role of associator (see figure 5.18). It allows the program and behavioural change components to remain completely separate, and it allows the association to be initiated without adding any code to any part of the program.

```
[1]  public class Console extends MExecute implements Runnable {
[2]     public Console() {
[3]        Thread thr = new Thread(this);
[4]        thr.start();
[5]     }
[6]
[7]     public Object execute(Object o, Object[] args, Method m)
[8]                                    throws InvocationTargetException {
[9]        return(proceed(o, args, m));
[10]    }
[11]
[12]    public void run() {
[13]       String protName = null;
[14]       String className = null;
[15]       Object[] params = null;
[16]       while(true) {
[17]          // ... Ask for name of protocol.
[18]          // ... Ask for name of class to be adapted.
[19]          // ... Ask for parameter values, if any.
[20]          Meta.associate(className, protName, params);
[21]       }
[22]    }
[23] }
```

Figure 5.22: Outline of a metaobject class to implement an association console.

However, the Iguana/J configuration file is only read at startup so the declaration of association must be included in it before the system is started. This is a form of anticipation of the adaptation, although it is non-invasive of the original program. A technique that can be used to avoid this anticipation of the specific association is to create a metaobject class and protocol that implement an association control console. Figure

5.22 gives an outline of the metaobject class `Console` that does this. Although the metaobject reifies invocation it just proceeds with the original operation, without modifying it in any way (line 9). The purpose of the metaobject is only to create a separate thread that accepts association information interactively from the keyboard (or other source) during runtime and dynamically initiates required associations. The thread is created by the constructor of the metaobject class (lines 3 and 4), and the code executed by the thread is in the `run()` method (lines 12 to 22). If a protocol that includes the `Console` metaobject is associated with any class of the program by including a declaration in the Iguana/J configuration file then during runtime the console can be used to associate any named class with any named protocol. The only anticipation needed to allow this is the statically initiated association of any class of the program with the protocol that includes the `Console` metaobject. Once this has been done any class can be adapted via the console interface during runtime, without anticipation and without invasive changes to the original program.

5.5.3 Other solutions

PROSE does not support redirection of intecepted operations, so it is not possible to use it to address this particular example. PROSE is intended to weave additional behaviour into a program rather than modify its existing behaviour. However, it should be pointed out that the mechanism used by PROSE to initiate association, or weaving, supports a very strong separation of concerns. As described in section 2.7.5, the JVMDI used by PROSE to communicate with the JVM supports remote insertion of breakpoints, and therefore PROSE can weave aspects into a program at runtime without any anticipation or invasive changes to the program.

MetaXa and Guaraná both support redirection of reified operations, so they can implement the behavioural changes needed by this example, using metaobjects similar in their effect to the one for Iguana/J. However, both architectures support only dynamic initiation of association and therefore both require changes to the base-level program or its startup to include execution of the association statements. The minimum required is to create a small specialised bootstrap program that starts the main (unchanged) program and performs some initial association operations for it.

5.5.4 Summary of separation of concerns

Iguana/J and the other three architectures used in this evaluation all support a separation of concerns between the original program and the behavioural change components, but there are variations in the strength of separation supported by them. They all have in common that no changes are required to the program before runtime. That is, no invasive change or anticipation is needed in the component to be adapted, and the behavioural change can be carried out without access to source code. In Filman's terms, the program remains oblivious to any change that might be applied to it [Fil00].

MetaXa and Guaraná suffer from a weakened separation of concerns because of the need to execute statements initiating association. They cannot support an associator entity that is completely outside both the program and the behavioural change components, as illustrated in figure 5.18. PROSE does not have this weakness, as its implementation technique supports dynamic initiation of association from outside the JVM.

Iguana/J also supports a strong separation, with its static initiation of association via declarations in the configuration file. Unanticipated associations can be initiated dynamically by using a console metaobject to accept commands during runtime. In fact, the association console technique could be extended to produce separation of concerns comparable to that supported by PROSE. This could be done by integrating the console functionality into the Iguana/J runtime engine and redesigning it to accept commands via a communications socket. A similar technique might improve the separation supported by MetaXa and Guaraná. However, its usefulness for MetaXa would be limited by the recommendation of the architecture's authors that metaobjects associated with a class should not be changed when instances exist [Gol97], and its usefulness for Guaraná would be limited by the fact that that architecture does not propagate behavioural change from a class to already existing instances of the class.

The separation of concerns discussed here relates to association of behavioural change with classes. There is also the issue of separation when associating change with individual objects. It is more difficult to maintain the kind of separation illustrated in figure 5.18, because objects do not have statically known identifiers in the way that classes have statically known names. Unless an individual object can be identifed by conditions that can be statically expressed outside the program, association of change with the object can only be initiated by a component that holds a reference to the object. This implies that association with an individual object can only be initiated by code that is part of the program, and therefore that invasive changes must be made to the program to support initiation of the association.

5.6 Inheritance of change

If dynamic adaptation is to be useful then it is important that each behavioural change applied to a class be inherited by its subclasses [Tar00]. This is part of the support for addressing the inheritance anomaly, and for ensuring that subclasses include all the behaviour of their superclasses. Without inheritance of change propagation of changes throughout part of a class hierarchy must be done explicitly, and it is difficult to apply behavioural change reliably to a set of related classes.

5.6.1 The inheritance challenge

The example chosen to represent this challenge is to adapt a graphical user interface to include a user activity timeout. The timeout should be triggered if no user interface activity is detected within some specified time, and some predetermined action should be taken as a result of a timeout. It is relatively simple to reify the reception of interface events (representing user activity) for graphical classes such as buttons and text boxes, and to generate a timeout if no event is received by any of them for a defined period. The difficulty is to apply this behavioural change reliably to all graphical classes that might be used in an interface. In addition to the large number of graphical display classes available in the standard Java class library a program can include its own special subclasses of them. It can be difficult to ensure that all required graphical classes have the behavioural change that allows them to participate in the activity detection. If any graphical class used in the interface is omitted from the timeout adaptation then some user activities will not be detected by the timeout component, and the timeout could occur even when the user is active.

```
public class Display implements ActionListener {
   Frame fMain = new Frame();;
   Frame fLogin = new Frame();
   Button btnLogin,btnNext,btnExit;
   // ... More display components.

   public Display() {
      // Setup the login frame:
      fLogin.setTitle("Login");
      // ... Create and add name & password text fields.
      btnLogin = new Button("Login");
      fLogin.add(btnLogin);
      fLogin.setVisible(true);

      // Setup the data frame:
      fMain.setTitle("Data");
      // ... Create and add various display components.
      btnNext = new Button("Next");
      fMain.add(btnNext);
      btnExit = new Button("Exit");
      fMain.add(btnExit);
      // ... Add various action listeners.
   }

   public void actionPerformed(ActionEvent e) {
      if(e.getSource().equals(btnLogin)) {
         if(passOk()) {
            fLogin.setVisible(false);
            fMain.setVisible(true);
         }
      }
      else if(e.getSource().equals(btnNext)) {
         // ... Next data set for display.
      }
      else if(e.getSource().equals(btnExit)) {
         fMain.setVisible(false);
         fLogin.setVisible(true);
      }
   }

   private boolean passOk() {
      // ...
   }
}
```

Figure 5.23: Outline of a display class with login and data frames.

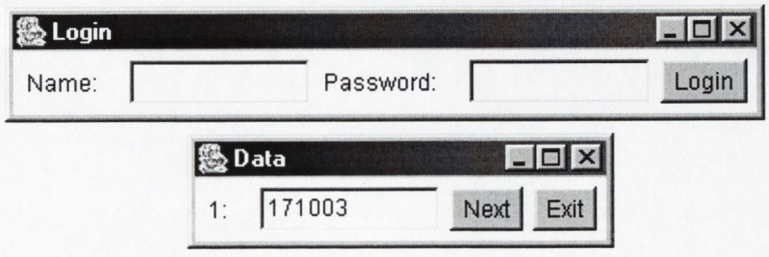


Figure 5.24: The simple login and data windows created by the `Display` class.

Reliable timeout behaviour can only be implemented if the behavioural change is inherited by all subclasses of display components. Without inheritance of change each class must be adapted individually. Even if an attempt is made to apply the change individually to every display class used in an interface a subsequent design change could introduce a new subclass. If the timeout behaviour is not explicitly applied to that new subclass then the activity detection for the timeout will be broken.

An outline of the user interface class for this example is given in figure 5.23. The class, called `Display`, creates a login window and a main data window (see figure 5.24). After a user logs in, the data window is presented. When the user clicks 'Exit' in the data window the interface returns to the login window. The challenge is to provide a behavioural change component that automatically returns the user interface to the login window if there is no user activity for a specified period.

5.6.2 The Iguana/J solution

In Iguana/J detection of activity by the user is implemented by a metaobject intended to reify invocations of `processEvent()` in graphical components. Every graphical class inherits `processEvent()` from `java.awt.Component`, and it is called by the graphical subsystem every time an interface event of any kind occurs on the visible component represented by the graphical class. By monitoring invocation of `processEvent()` on all components it is possible to detect user activity. The metaobject class `Timeout` is intended to do this and is listed in figure 5.25. For clarity exception handling and some code details are omitted from the listing. For reusability of the metaobject class the name of the method to monitor and the timeout period are expected as arguments to its constructor (line 6). On each invocation of the named method the metaobject records a timestamp (line 18). A separate thread, started by the constructor (lines 9 and 10), continuously monitors the timestamp (lines 24 to 30) and manipulates the displayed windows if it detects that the time since the timestamp is greater than the timeout period (lines 25 to 28).

Figure 5.26 lists a protocol that uses this metaobject class. The protocol expects a method name and a timeout period as two parameters, and it passes them to the metaobject. Figure 5.27 shows how the timeout can be applied to all the graphical components in a single operation. The class `java.awt.Component` is the superclass of all the graphical components in Java's abstract window toolkit (AWT). Iguana/J's automatic inheritance of association means that the protocol is associated with all subclasses, so the single initiation of association in figure 5.27 associates the protocol with all graphical classes, even subclasses of which the programmer is unaware.

5.6.3 Other solutions

MetaXa, Guaraná, and PROSE all support creation of a behavioural change component that monitors method invocation and implements a timeout. The timeout metaobjects for MetaXa and Guaraná are similar in basic structure to the Iguana/J metaobject listed in figure 5.25. They record the time at each invocation of a named method, and use a separate thread to monitor the timeout period. The timeout behaviour can be applied to a graphical class by associating the metobject with it and specifying the name of the `processEvent()` method. However, neither MetaXa nor Guaraná support inheritance of metaobject association so the

metaobject must be associated with each graphical class individually. The risk that the timeout will not operate reliably is increased, particularly if a future change to the user interface adds a previously unused graphical class. It is also difficult to use these architectures to apply a timeout to a third party user interface where a full list of the graphical classes used is not easy to obtain.

```
[1]  public class Timeout extends MExecute implements Runnable {
[2]    volatile private long stamp;
[3]    private String name;
[4]    private int period;
[5]
[6]    public Timeout(String name, int period) {
[7]      this.name = name;
[8]      this.period = period;
[9]      Thread thr = new Thread(this);
[10]     thr.start();
[11]   }
[12]
[13]   public Object execute(Object o, Object[] args, Method m)
[14]                                throws InvocationTargetException {
[15]     Object result;
[16]     String s = m.getName();
[17]     if(s.equals(name)) {
[18]       stamp = System.currentTimeMillis();
[19]     }
[20]     return(proceed(o, args, m));
[21]   }
[22]
[23]   public void run() {
[24]     while(true) {
[25]       if((System.currentTimeMillis() - stamp) > period) {
[26]            Frame[] f = Frame.getFrames();
[27]            // ... Show login frame & hide all other frames.
[28]       }
[29]       Thread.sleep(1000);
[30]     }
[31]   }
[32] }
```

Figure 5.25: A metaobject class to implement an inactivity timeout.

```
protocol TimeoutProt(String m, int t) {
  reify Invocation: Timeout(m, t);
}
```

Figure 5.26: A protocol to use the `Timeout` metaobject class.

```
Object[] params = new Object[2];
params[0] = "processEvent";
params[1] = new Integer(30000);
Meta.associate("java.awt.Component", "TimeoutProt", params);
```

Figure 5.27: Applying a timeout of 30 seconds to all graphical components.

In PROSE a crosscut class can be designed to record the time of each invocation of a named method, and to start a thread that monitors the period since the last invocation. PROSE supports inheritance of behavioural change by providing a way to specify that a crosscut includes subclasses of the named class. This is done by including the 'in subclass' condition in the specialiser used to refine the scope of the crosscut. A PROSE specialiser to apply advice to invocations of a method named `m` in all subclasses of a class named `c` would look like:

```
setSpecializer(MethodS.named(m).AND(DeclarationS.inSubclass(c)));
```

For the user interface timeout the strings 'processEvent" and 'java.awt.Component" would be supplied as argument values to the constructors of the PROSE aspect and crosscut classes, and passed on to this specialiser expression as values for `m` and `c` respectively.

5.6.4 Summary of inheritance

Only Iguana/J and PROSE support inheritance of behavioural change, although in PROSE it is not automatic. In MetaXa and Guaraná subclasses do not inherit the changes applied to their superclasses, and therefore it can be complex and error-prone to apply a change to a set of related classes. Iguana/J is the only architecture in which inheritance of change is enforced, and in which a programmer can have confidence that a subclass includes all the behaviour of its superclass under all conditions.

A similar issue occurs for the relationship between a class and its instances. Neither MetaXa nor Guaraná enforce any rules relating behavioural change applied to a class with change applied to individual instances of the class, so instances may not include behavioural change associated with their class. Guaraná does not automatically propagate behavioural change from a class to its instances. PROSE does not support weaving of aspects with individual objects at all. Only Iguana/J supports changes to individual objects and enforces a relationship that means every instance always includes behavioural changes applied to its class.

5.7 Performance

Architectures that support unanticipated dynamic adaptation with a strong separation of concerns inevitably suffer some performance penalty. This is because maintaining a separation between the components of the original program and the components implementing behavioural change implies a need to transfer execution back and forth between them at the points in the program where the changes are to take effect. This transfer is an execution-time overhead, additional to carrying out the behavioural change. In implementation terms the overhead is caused by interception mechanisms, marshalling of arguments, and invocation of handler methods. Different implementation approaches result in different delays in the transfer and handling of operations. This section compares the performance cost of Iguana/J with that of other architectures supporting unanticipated dynamic adaptation.

5.7.1 Iguana/J performance

Indicative performance figures were obtained for Iguana/J by running a set of tests and measuring the time taken for each. The tests were run using Sun's JVM version 1.3.0. As the JIT Compiler Interface used by Iguana/J is not compatible with Sun's Hotspot JVM, all the tests were run using the 'classic' mode of the JVM.

The tests involved two operations, method invocation and object creation, with no arguments. Each test consisted of a base-level program executing the test operation repeatedly in a loop. Timing measurements were done by taking the system time before entering and after leaving the test loop. The loop counts were set to 10^6, to lengthen the overall time for each test and thereby reduce the effect of any time measurement error. For each test the initial memory allocation for the JVM was set to a figure large enough to avoid garbage collection during the test. The protocols used for the tests used a single metaobject to intercept the operation of interest. The metaobject did nothing but carry out the intercepted operation using the `proceed()` method (in effect an empty metaobject).

The test results are summarised in table 5.1. The figures in each column are normalised to the time to perform the base-level operation without Iguana/J present. The five rows in the table represent different test conditions, as follows:

1. Without Iguana/J:

   The tests were run without the Iguana/J runtime engine at all. This provides a baseline measurement of 'normal' operation, and by definition gives normalised figures of 1.

2. No protocol associated:

   The Iguana/J runtime engine was loaded, but the base-level object being tested had no associated protocol. This measures the general effect of the presence of the Iguana/J runtime engine.

3. Protocol but no reification:

   The base-level object being tested had an associated protocol that did not reify the operation being tested. For example, the protocol reified creation but the test involved invocation of a method. This measures the effect of association on non-reified operations.

4. Reification on a subclass:

   A subclass of the test class had the operation under test reified for an inherited member. This caused the interception mechanism to be inserted into the method, affecting the class under test. This measures how quickly the runtime engine can detect that the target object does not need the interception that has occurred.

5. Reification on test class:

   The test class had the operation under test reified. This test measures the time that it takes to transfer an intercepted operation to the meta-level, to carry it out at the meta-level and to transfer control back to the base-level. This measures the total overhead for a reified operation.

	Test condition	Invocation	Creation
1	Without Iguana/J	1	1
2	No protocol associated	1	1
3	Protocol but no reification	1	25.7
4	Reification on a subclass of the test class	1.71	1
5	Reification on the test class	24.1	34.5

Table 5.1: Normalised performance figures for Iguana/J.

It can be seen by comparisons of rows 1 and 2 in table 5.1 that the presence of the Iguana/J runtime engine has no effect on objects that have no associated meta-level protocol. From row 3 it can be seen that method invocation is not affected at all by associating a protocol that does not intercept invocation, but creation is strongly affected even if the associated protocol does not intercept creation. This is because associating a protocol with a class means that an instance of the protocol class and a set of metaobjects must be constructed for each new instance of the class, even if creation is not reified for the protocol. This figure indicates the time required to construct these meta-level objects for a new base-level object.

From row 4 in the table it can also be seen that the effect of Iguana/J on method invocations that have the interception mechanism in place but that are not required to be intercepted by the current target object is an increase of 71% in the time to invoke the method[19]. This measures the time for the interception code to recognise that no reification is needed for this target object. Ideally this delay should be zero, as no extra functionality is required under these conditions. Detailed examination of the causes of the delay has shown that approximately one third of it is due to the distance in memory between the original JVM code and the Iguana/J runtime engine, leading to significant instruction cache misses every time the Iguana/J interception code is executed. It is expected that the delay would be reduced significantly if the Iguana/J interception code were integrated directly into the JVM code. Row 4 also shows that object creation is not affected by associating a protocol with a subclass, because no meta-level has to be constructed for instances of the superclass and because constructors are not inherited.

Line 5 of the table shows that transferring an operation to the meta-level has a significant performance cost, with a factor of 24.1. This is in line with expectations, as the transfer involves invocation of a chain of functions, and the use of the Java reflection classes to carry out the operation. Comparison of lines 3 and 5 for object creation show that most of the factor of 34.5 is due to the construction of the new meta-level objects for the base-level object being created, leaving a factor of 8.8 due to the transfer of the creation operation to the appropriate metaobject and the use of the Java reflection classes to carry it out.

These results can be summarised as:

- There is no performance cost at all for operations on a class or its instances where there is no protocol associated with the class or the instances, or with any subclass of the class.
- There is a small performance cost (a factor of 1.71) for invocations that are intercepted but not reified.
- There is a cost factor of 25.7 for object creation for any class that has a protocol associated with it.
- There is a performance cost of 24.1 for reified method invocation.

[19] Note that this is the increase in time to *invoke* the method. The time to execute the invoked method is not affected.

5.7.2 Relative performance

It is difficult to be sure that published performance figures for other architectures supporting unanticipated dynamic adaptation are measured using comparable tests. However, one of the performance figures available for most architectures is the overhead suffered by a reified (or adapted) method invocation. The test condition for this is almost always reification of invocation of an empty method, with no behavioural change. By comparing this to direct (unreified) invocation of an empty method in the same architecture a good general indication of the runtime efficiency of the interception and transfer mechanisms in that architecture is obtained. The figure for Iguana/J is 24.1, as in row 5 of table 5.1.

Published figures for other dynamic adaptation architectures indicate delays of the same order. Guaraná gives a factor of over 40 for method invocation when a JIT compiler is not used [Oli98]. PROSE gives a factor of 70 or more for method invocation [Pop01]. Figures for MetaXa show a slowdown of 28 times for method invocation [Gol97].

Two of these architectures, PROSE and Guaraná, also have other negative influences on their performance costs. The PROSE architecture is implemented using the JVM debug interface (JVMDI) as described in section 2.7.6, and therefore a performance cost is incurred even if no behavioural changes are in place. Enabling the JVMDI causes a slowdown of 2 to 5 times for all method invocations and all field accesses in all classes [Pop01]. Guaraná reifies all operations in any base-level class that has a metaobject associated with it, so the reification overhead is incurred for all operations even when only one type of operation is of interest.

One other major influence on potential performance speed for these architectures is their support for using a JIT compiler. Iguana/J cannot be used with the Sun HotSpot compiling JVM because it is incompatible with the way that Iguana/J integrates with the JVM (see section 4.3). PROSE cannot be used with HotSpot for a similar reason (the JVMDI is incompatible with using HotSpot[20]). MetaXa is based on a proprietary JVM, and some work has been done on a JIT compiler for it although it is stated to be incomplete in the most recently published information [Gol97]. Guaraná is based on the Kaffe JVM [Kaf03], which supports JIT compilation.

	Cost of reifying invocation	**Classes with no association are unaffected?**	**Unreified operations are unaffected?**	**Supports JIT compiler?**
Iguana/J	24.1	✓	✓	✗
MetaXa	28	✓	✓	✓ (incomplete)
Guaraná	40	✓	✗	✓
PROSE	70	✗	✗	✓ (footnote 20)

Table 5.2: Some performance characteristics of architectures supporting unanticipated dynamic adaptation.

[20] Since version 1.4 of Sun's JVM it is possible to use HotSpot when the JVMDI is enabled, although any method that has a breakpoint set anywhere in it is interpreted rather than compiled. For PROSE this means that only methods with no aspects woven within them can be compiled by HotSpot..

Performance characteristics for Iguana/J and the three other reviewed architectures are summarised in table 5.2. MetaXa, which has execution speed as a principal objective, is closest to Iguana/J in performance characteristics. It seems likely that in real applications Iguana/J would be a little slower than MetaXa, as the MetaXa implementation takes a number of steps (notably the use of shadow classes) to minimise insertion of interception mechanisms for objects that do not need them.

5.8 Summary

This evaluation shows that Iguana/J has characteristics that allow it to address a wider set of challenges than are addressed by other architectures. Its support for context examination provides a solution to the 'jumping aspects' problem [Bri00] [Tar00]. Inheritance of behavioural change and derivation of protocols provide a solution for the inheritance anomaly in dynamic adaptation [Mat93] [Tar00]. The strong separation of concerns supported by Iguana/J allows for non-invasive unanticipated dynamic adaptation under a wider range of circumstances than MetaXa and Guaraná. The automatic enforcement of inheritance of change makes it easier and safer to apply changes to a set of related classes and objects. Iguana/J also supports reusability, composition, association, and adaptation features equal to most of those of the other architectures. It can be used to apply behavioural changes representing non-functional concerns such as synchronisation, as well as those that alter functional concerns by, for example, redirecting operations to an updated library class. Its performance is at the faster end of the range represented by the other architectures, and it has a significantly simpler class library and programmer's interface.

Table 5.3 lists a number of characteristics of Iguana/J and the three architectures used in this evaluation. Each of the other three has their own strong point. MetaXa minimises the performance cost, Guaraná supports security in manipulating the meta-level, and PROSE provides portability and a strong separation of concerns. Iguana/J addresses a range of issues. It provides inheritance of change enforced by the metatype structure, context sensitivity, a simpler class library, stronger separation of concerns than MetaXa and Guaraná, and faster performance than Guaraná and PROSE.

Characteristic	Iguana/J	MetaXa	Guaraná	PROSE
Reusability				
Reusable behavioural change components	✓	✓	✓	✓
Composition and association				
Explicit composition of concerns	✓	×	×	×
Dynamic composition of concerns	×	✓	✓	✓
Association with classes	✓	✓	✓[1]	✓
Association with individual objects	✓	✓	✓	×
Inheritance of association	✓	×	×	✓
Enforcement of inheritance of change	✓	×	×	×
Apply change to operations on private members	✓	✓	✓	×
Separation				
Non-invasive change	✓	✓	✓	✓
Associator separate from program & change component	✓	×	×	✓
Adaptation				
Alteration of operation arguments	✓	✓	✓	×
Alteration of return values	✓	✓	✓	×
Redirection or avoiding an operation	✓	✓	✓	×
Context sensitivity	✓	×	×	×
'Guard' type adaptations	✓	✓	×	×
Performance				
Effect on reified method invocation	x24.1	x28	x40	x70
Operations on unassociated classes not slowed	✓	✓	✓	×
Operations on unassociated objects not slowed [2]	×	✓	×	×
Unreified operations on associated classes not slowed	✓	✓	×	×
Simultaneous use of JIT compiler	×	✓[3]	✓	✓[4]
Implementation				
Portable	×[5]	×	×	✓
Number of classes in programmer's model	11	30 [6]	15	>40 [7]
Number of methods in programmer's model	28	>200 [6]	>100	>100 [7]

Notes:
[1] Association with a class is not propagated to existing instances of the class.
[2] When other instances of the same class are associated with a behavioural change component.
[3] The JIT compiler for MetaXa is incomplete.
[4] Methods that have aspects woven with them cannot be compiled.
[5] The Iguana/J implementation has some limited portability, as it uses an unmodified standard JVM.
[6] Excludes a large number of classes and methods supporting bytecode manipulation.
[7] Numbers are estimates, as the documentation does not distinguish between PROSE implementation classes and classes intended for use by aspect programmers.

Table 5.3: Comparison of characteristics of Iguana/J and other architectures

Chapter 6

Conclusion

God keep me from ever completing anything. This whole book is but a draught - nay, but a draught of a draught. Oh Time, Strength, Cash and Patience.

- Herman Melville.
Moby Dick, 1851.

Unanticipated dynamic adaptation allows change to be applied to systems that cannot or should not be halted and restarted. There is a wide range of systems that cannot tolerate interruptions, from large mission-critical financial systems to small mobile devices. The types of change that might be required also cover a wide range, from persistent maintenance upgrades to transient adaptations to account for temporary variations in available resources. There are existing architectures supporting unanticipated dynamic adaptation. The principal motivation for this research was to explore how support might be improved by addressing issues such as structure, constraint, simplicity in use, and various published challenges in the field.

6.1 Achievements

This thesis presents a model for supporting unanticipated dynamic adaptation based on behavioural reflection. It describes the implementation of the model and presents an evaluation of it. The model supports the creation of independent components implementing behavioural change, and the application of the change by reifying specified operations in selected classes and objects in the program.

Since software reflection was first introduced [Smi82] it has tended to be regarded as overly powerful and difficult to use [Men97]. Although it has been demonstrated by various architectures to be one of the most powerful techniques for supporting unanticipated dynamic adaptation, existing architectures using reflection to apply behavioural change are complex and allow undisciplined application of change. The architecture proposed by this thesis presents a simpler model of reflection that supports a safer more constrained application of behavioural change. This thesis demonstrates that reflection need not present a complex

programmer's model, and that it can be constrained to be less powerful in some sense while still supporting unanticipated dynamic adaptation.

6.1.1 Structure

Iguana/J brings a new structure to unanticipated dynamic adaptation. Inheritance of change, protocol derivation, and the metatype rules impose a structure in which change can be applied with more confidence. A programmer can be surer of the scope of applied change, and surer that it will remain in place within that scope. There is less risk that previously applied change will be inadvertantly lost when new change is applied, or omitted when new classes are added to the program.

The requirement that separate behavioural change components be explicitly composed (via protocol derivation) reduces the risk of conflict between incompatible changes, and ensures that the meta-level programmer is aware of the composition structures in use. This thesis also identifies and proposes solutions for some detailed issues within the structure. These include the treatment of multiply inherited parent protocols, shared metaobjects, and metaobjects in common between an existing and a new protocol association.

6.1.2 Simplification

Existing architectures supporting unanticipated dynamic adaptation present complex programmer's models. These require the programmer to cope with large class libraries, and to explicitly address details such as separate handler methods for different return types.

Iguana/J has a number of characteristics that simplify the implementation and application of behavioural change. It has a significantly smaller class library than other architectures, and therefore it should be much easier for a programmer to begin to use. There are no extensions to the Java language. Metaobject classes, implementing behavioural change, are standard Java classes expressed in normal Java syntax. The arguments and targets of operations to be adapted are represented using the existing Java reflection classes, so many metaobjects can be implemented without reference to the Iguana/J classes at all. The sole exception is when examination of the current execution stack is required; this requires use of the `CallStack` class. Iguana/J metaobject classes contain a single handler method, so programmers do not have to consider separate handling of operations depending on the return type. Iguana/J also includes automatic detection and avoidance of potential reflective recursion, so the need for the meta-level programmer to select between different invocation techniques is removed.

6.1.3 Constraint

Iguana/J is the first attempt to impose ordered constraints on the use of reflection for unanticipated dynamic adaptation. Although constraints on reflection were introduced by Forman and Danforth (SOM, [For98]) the constraints introduced by this thesis support a more dynamic non-invasive model of adaptation. The

justification for constraints in Iguana/J is based on behavioural arguments rather than the structural requirements for substitutability used to justify constraints in SOM. Iguana/J constraints control the way in which behavioural change can be applied to both classes and individual objects, and provide a safer more predictable environment for dynamic adaptation. The constraints are supported by the concept of metatype, and ensure that the propagation of behaviour from class to subclass and from class to instance remains predictable under all behavioural changes. The constraints proposed in this thesis help to further the understanding of how the power of reflection might be controlled to become more useful and more usable for unanticipated dynamic adaptation.

While in principle Iguana/J allows association of any behavioural change with arbitrary classes and objects, in practice this is limited by the metatype rules. There are no invasive or anticipative changes to the program or references embedded in the behavioural change component to impose limits. Associations are limited only by the Iguana/J metatype rules (and of course by the semantics of the behavioural change in the context of any particular program class or object). The metatype rules ensure that a behavioural change applied to a class is always applied to its subclasses (recursively) and to its instances, and cannot be removed from those subclasses and instances.

6.1.4 Challenges

Iguana/J addresses several important acknowledged issues in the field of separate implementation of behavioural change for dynamic adaptation. It provides solutions to a wider range of these issues than other architectures supporting unanticipated dynamic adaptation. Iguana/J addresses these issues by providing the following combination of features for behavioural change components:

- They are reusable in their compiled form, and can be specialised at runtime via parameters;
- They are inherited.
- They can be designed to examine the runtime context of the operation that they adapt. Context includes arguments, return values, and invocation chain.
- They can alter arguments and return values.
- They can redirect or cancel the operation that they adapt.
- They can apply wrapper adaptations, such as synchronisation, that need to enclose both the invocation and the return from the adapted operation.

Iguana/J also addresses the form of the inheritance anomaly identified in the field of separate implementation of non-functional concerns. It supports non-invasive adaptation of programs, with a very strong separation between behavioural change components, the program to be adapted, and the initiator of the association between them. This supports the ternary approach to system development, in which the behavioural change component and the program can be developed and deployed independently, and then a separate adaptation architect can create associations between them as required.

6.2 Future work

The architecture proposed here opens a door to a simpler and more useful approach to unanticipated dynamic adaptation, and shows how the power of reflection can be controlled to provide a safer and more predictable adaptation scope. As always, there are many issues that need to be explored further in any future research in this area. Some possibilities are suggested below.

6.2.1 Dynamic derivation of protocols

Iguana/J supports static creation and derivation of protocols. It is not possible to create or derive protocols at runtime. As protocol derivation is the means to compose metaobjects this also means that it is not possible to create a new metaobject composition at runtime. Explicit composition of metaobjects is an important aspect of the Iguana/J model, and reduces the risk of inadvertantly composing conflicting behavioural changes. While dynamic composition of metaobjects could support a more flexible approach to manipulation of independent behavioural changes it does not appear to sit easily with the dynamic nature and constraints of Iguana/J. SOM [For98] supports automatic derivation of new metaclasses to satisfy constraints at runtime, and this represents composition of components implementing change. It would be interesting to investigate to what extent dynamic composition could be supported in the more dynamic and unanticipated environment of Iguana/J without eroding the benefits of its structure.

6.2.2 Alternative composition strategies

The architecture proposed by this thesis supports only a single metaobject composition strategy, the commonly used linear chain. This is simple to implement and understand, and it is sufficient for many applications. However, it limits the kinds of composite behavioural changes that can be created. Support for more complex composition strategies might allow Iguana/J to address a wider class of adaptation problems. On the other hand, it might also bring about a closer coupling between protocols, and thus affect Iguana/J's basic assumption that each protocol implements an independent behavioural change represented as a metatype. Research into how alternative composition strategies could be supported within the metatype structure would be a useful extension of this research.

6.2.3 Security implications

Iguana/J supports application of behavioural change to any class and any object, including Java system classes and objects. This obviously has the potential to be a serious security problem. For example, it could be used to defeat any security permission by altering `AccessController.checkPermission()` to always act as if every permission was granted. While some work has been done on reflection and security [Car01] a more specific examination of the security implications of Iguana/J and how they might be addressed is needed.

6.2.4 Alternative implementation techniques

While the implementation technique used for this research was adequate, and proved to be reasonably efficient in execution time, it can only be used with a JVM that supports the JIT interface. As this exists only on Sun JVMs, the Iguana/J architecture is not portable to JVMs from other companies. Even on Sun JVMs the JIT interface is no longer supported from version 1.4 onwards, so it will not be possible to use or incorporate future Java features in Iguana/J.

Alternative implementation techniques might be able to overcome these limitations. Possible examples include integration into the source code of an existing open sosurce JVM, or using a standard JVM feature such as the Debug Interface (JVMDI) or the recently added class replacement support [Dmi02]. Some alternative implementation techniques might bring other benefits, such as improved execution times and ability to use a JIT compiler with Iguana/J.

6.2.5 More abstract support frameworks

The architecture proposed here is effectively a low level mechanism to support dynamic adaptation; there are a number of higher level issues that are not addressed by this architecture, but which could be addressed by a more abstract dynamic adaptation framework. These include issues such as admission and timing of change, and state transfer across a change. At an even higher level it would be useful to examine the potential usefulness of tools to support creation and application of coordinated behavioural change to large sets of program components.

6.2.6 Static type checking

Static checking of protocol parameter types in statically or dynamically initiated associations is not supported by Iguana/J because of the strong separation that the architecture provides between the runtime entities involved. Parameter value types cannot be statically checked because values are supplied as strings or are dynamically marshalled into `Object` arrays, and in the case of statically initiated associations they cannot be statically checked because the declaration of association is not compiled before runtime. The conflict between strong separation and static type checking seems to be a common one in dynamic adaptation architectures, and research into how this might be resolved or reduced could help to improve the safety of dynamic adaptation.

There is also a type checking issue in relation to metatypes. The Iguana/J metatype rules influence whether or not a given behavioural change may be applied to a particular class or object at runtime, but it is not simple to check in advance whether a particular proposed metatype will be accepted or not. It might be useful to be able to perform some kind of static metatype checking.

References

[Aks96] Mehmet Aksit, Bedir Tekinerdogan, and Lodewijk Bergmans. Achieving adaptability through separation and composition of concerns. In Max Muhlhauser, editor, *Special Issues in Object-Oriented Programming, Workshop Reader of the 10th. European Conference on Object-Oriented Programming*, pages 12-23. Dpunkt-Verlag, 1997.

[Ald00] Jonathan Aldrich. Challenge problems for Separation of Concerns. Workshop on Advanced Separation of Concerns on Object Oriented Systems at OOPSLA 2000 (position paper).

[Ama99] Noriki Amano and Takuo Watanabe. An Approach for Constructing Dynamically Adaptable Component-based Software Systems using LEAD++. In *Proceedings of the first OOPSLA Workshop on Reflection and Software Engineering (OORaSE'99)*. Walter Cazzola, Robert J. Stroud, and Francesco Tisato, editors. Published by University of Milano Bicocca, 1999.

[Ama02] Noriki Amano and Takuo Watanabe. A software model for flexible and safe adaptation of mobile code programs. In *Proceedings of 2002 International Workshop on Principles of Software Evolution (IWPSE-02)*, pages 57-61, ACM Special Interest Group on Software Engineering, ACM Press, 2002.

[Bak02] Jason Baker and Wilson Hsieh. Runtime Aspect Weaving through Metaprogramming. In *Proceedings of the 1st International Conference on Aspect-oriented Software Development*, pages 86-95. ACM Press, 2002.

[Ber03] Lodewijk M.J. Bergmans. Towards Detection of Semantic Conflicts between Crosscutting Concerns. Workshop on Analysis of Aspect Oriented Software at ECOOP 2003 (position paper).

[Ber97] Paul L. Bergstein. Maintenance of Object-Oriented Systems During Structural Schema Evolution. In Karl Lieberherr and Roberto Zicari, editors, *Theory and Practice of Object Systems*, volume 3, number 3, pages 185-212. John Wiley, New York, 1997.

[Bie92] James M.Bieman. Deriving Measures of Software Reuse in Object Oriented Systems. In T.Denver, R.Herman, and R.Whitty, editors, *Formal Aspects of Measurement*, pages 79-82. Springer-Verlag, 1992

[Bie95] J. M. Bieman and J. X. Zhao. Reuse Through Inheritance: a Quantitative Study of C++ Software. In S. Samadzadeh and M. Zand, editors, *Proceedings of the Symposium on Software Reusability*, pages 47-52. ACM Special Interest Group on Software, ACM Press, 1995.

[Bou98] Noury Bouraqadi-Saâdani, Thomas Ledoux and Fred Rivard. Safe Metaclass Programming. In *Proceedings of the 1998 Conference on Object-Oriented Programming Systems, Languages and Applications*. ACM Special Interest Group on Programming Languages, ACM Press, 1998.

[Bou99] Mohammed Nouraddine Bouraqadi-Saâdani. *Une MOP Smalltalk pour l'étude de la composition et de la compatibilité des métaclasses.* PhD Thesis, l'École Nationale Supérieur des Techniques Industrielles et des Mines de Nantes, Université de Nantes, France. July 1999.

[Bou01] Noury Bouraqadi-Saâdani and Thomas Ledoux. How to Weave? Workshop on advanced separation of concerns at ECOOP 2001 (position paper).

[Bri00] Johan Brichau, Wolfgang de Meuter and Kris de Volder. Jumping Aspects. Workshop on aspects and dimensions of concern at ECOOP 2000 (position paper).

[Bri01] Johan Brichau, Maurice Glandrup, Siobhán Clarke and Lodewijk Bergmans. *Advanced Separation of Concerns.* Report of workshop on advanced separation of concerns at ECOOP'2001. http://trese.cs.utwente.nl/Workshops/ecoop01asoc/ws.pdf. Visited September 2003.

[Bru01] Eric Bruneton and Michel Riveill. Experiments with JavaPod, a Platform designed for the Adaptation of Non-functional Properties. In Akinori Yonezawa and Satoshi Matsuoka, editors, *Metalevel Architectures and Crosscutting Separation of Concerns*, proceedings of Reflection 2001, volume 2192 of *Lecture Notes in Computer Science*, pages 52-72. Springer-Verlag, 2001.

[Cap01] Licia Capra, Wolfgang Emmerich and Cecilia Mascolo. Reflective Middleware Solutions for Context-Aware Applications. In Akinori Yonezawa and Satoshi Matsuoka, editors, *Metalevel Architectures and Crosscutting Separation of Concerns*, proceedings of Reflection 2001, volume 2192 of *Lecture Notes in Computer Science*, pages 126-133. Springer-Verlag, 2001.

[Car01] Denis Caromel and Julien Vayssière. Reflections on MOPs, Components and Java Security. In Knudsen, J, editor, *ECOOP 2001*, volume 2072 of *Lecture Notes in Computer Science*, pages 256-274. Springer-Verlag, 2001.

[Chi95] Shigeru Chiba. A Metaobject Protocol for C++. In *Proceedings of the 1995 Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 285-299. ACM Special Interest Group on Programming Languages, ACM Press, October 1995. Also SIGPLAN Notices 30(10), October 1995.

[Coh98] Geoff A.Cohen, Jeffrey S.Chase, and David L.Kaminsky. Automatic program Transformation with JOIE. In *Proceedings of 1998 USENIX Annual Technical Conference, June 1998.*

[Con00] Constantinos Constanides, Atef Bader and Tzilla Elrad. A Two-dimensional Composition Framework to Support Software Adaptability and Reuse. In William B Frakes, editor, Software Reuse: Advances in Software Reusability, volume 1844 of *Lecture Notes in Computer Science.* Springer-Verlag, 2000.

[Cza00] Krzysztof Czarnecki and Ulrich W.Eisenecker. *Generative Programming.* Addison Wesley, 2000.

[Dah99] Marcus Dahm. Byte Code Engineering. In Clemens H. Cap, editor, *JIT '99, Java-Informations-Tage 1999*, Informatik Aktuell, pages 267-277. Springer, 1999.

[Dan94] Scott Danforth and Ira R. Forman. Reflections on Metaclass Programming in SOM. In *Proceedings of the 1994 Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 440-452. ACM Special Interest Group on Programming Languages, ACM Press, 1994. Also SIGPLAN Notices volume 29 number 10, October 1994.

[Dav02] Peirre-Charles David and Thomas Ledoux. Dynamic Adaptation of Non-Functional Concerns. Workshop on unanticipated software evolution at ECOOP 2002 (position paper).

[Dij76] Edsger W.Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

[Dmi01] Mikhail Dimitriev. *Towards flexible and safe technology for runtime evolution of Java language applications.* Workshop on Engineering Complex Object-Oriented Systems for Evolution (ECOOSE) at OOPSLA 2001..

[Dou01] R. Douence, M. Südholt. A Generic Reification Technique for Object-Oriented Reflective Languages. In *Higher-Order and Symbolic Computation*, volume 14, number 1, March 2001, pages 7-34. Kluwer Academic Publishers, 2001.

[Dow00] Jim Dowling, Tilman Schäfer, Vinny Cahill, Peter Haraszti and Barry Redmond. Using Reflection to Support Dynamic Adaptation of System Software. In Walter Cazzola, Robert J.Stroud and Franceso Tisato, editors, *Reflection and Software Engineering,* volume 1826 of *Lecture Notes in Computer Science*, pages 171-190. Springer-Verlag, June 2000.

[Dow01] Jim Dowling and Vinny Cahill. The K-Component Architecture Meta-model for Self-Adaptive Software. In Akinori Yonezawa and Satoshi Matsuoka, editors, *Metalevel Architectures and Crosscutting Separation of Concerns*, proceedings of Reflection 2001, volume 2192 of *Lecture Notes in Computer Science*, pages 81-88. Springer-Verlag, 2001.

[Duc02] Frédéric Duclos, Jacky Estublier and Philippe Morat. Describing and Using Non-functional Aspects. In *Proceedings of the 1st international conference on Aspect-oriented Software Development*, pages 65-75. ACM Press, 2002.

[Eid01] Eric Eide, Alastair Reid, Matthew Flatt and Jay Lepreau. Aspect Weaving as Component Knitting: Separating Concerns with Knit. Workshop on Advanced Separation of Concerns in Software Engineering at ICSE 2001 (position paper).

[Fab76] R.S.Fabry. How to design a system in which modules can be changed on the fly. In *Proceedings of 2nd International Conference on Software Engineering*, pages 470-476. IEEE Computer Society Press, 1976.

[Fab98] Jean-Charles Fabre and Tanguy Pérennou. A Metaobject Architecture for Fault Tolerant Distributed Systems: The FRIENDS Approach. In *Transactions on Computers*, volume 47 number 1, January 1998, pages 78-95. IEEE Computer Society.

[Fer89] Jacques Ferber. Computational Reflection in Class based Object Oriented Languages. In *Proceedings of the 1989 Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 317-326. ACM Special Interest Group on Programming Languages, ACM Press, 1989. Also SIGPLAN Notices volume 24 number 10, October 1989.

[Fer95] Szabolcs Ferenczi. Guarded Methods vs. Inheritance Anomaly - Inheritance Anomaly Solved by Nested Guarded Method Calls. In *ACM SIGPLAN Notices*, Volume 30 number 2, pages 49-58, February 1995

[Fil00] Robert Filman and Daniel Friedman. Aspect Oriented programming is Quantification and Obliviousness. Workshop on Advanced Separation of Concerns at OOPSLA 2000 (position paper).

[For98] Ira R. Forman and Scott H. Danforth. *Putting Metaclasses to Work.* Addison Wesley, 1998.

[Gam95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison Wesley, 1995.

[God02] Michael Godfrey and Qiang Tu. Tracking Structural Evolution using Origin Analysis. In *Proceedings of 2002 International. Workshop on Principles of Software Evolution (IWPSE-02)*, pages 117-119, ACM Special Interest Group on Software Engineering, ACM Press, 2002. (position paper).

[Gol97] Michael Golm. *Design and Implementation of a Meta-Architecture for Java.* Master's Thesis, Dept. of Computer Science, Friedrich-Alexander-University, Erlangen-Nuremberg, 1997.

[Gol99] Michael Golm and Jürgen Kleinöder. Jumping to the Meta Level: Behavioural Reflection can be Fast and Flexible. In Pierre Cointe, editor, *Meta-Level Architectures and Reflection*, volume 1616 of *Lecture Notes in Computer Science*, pages 22-39. Springer-Verlag, 1999.

[Gos00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification.* 2nd edition, Addision-Wesley, 2000.

[Gow97] Brendan Gowing. *A Reflective Model and Language for Dynamically Modifying Compiled Code.* PhD Thesis, Department of Computer Science, University of Dublin, Trinity College, 1997.

[Gup96] Deepak Gupta, Pankaj Jalote, and Gautam Barua. A Formal Framework for On-line Software Version Change. In *IEEE Transactions on Software Engineering*, volume 22, number 2, pages 120-131, February 1996.

[Hag02] Daniel Hagimont and Noël De Palma. Non-functional Capability-based Access Control in the Java Environment. In Zohra Bellahsene, Dilip Patel and Colette Rolland, editors, *Proceedings of the 8th International Conference on Object-Oriented Information Systems*. volume 2425 of *Lecture Notes in Computer Science*, pages 323-335. Springer-Verlag, 2002.

[Han02] Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings of the 2002 Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 161-173. ACM Special Interest Group on Programming Languages, ACM Press, 2002.

[Hic01] Michael Hicks, Jonathan T.Moore, and Scott Nettles. Dynamic Software Updating. In *Proceedings of 2001 conference on Programming Language Design and Implementation*, pages 13-23. ACM Special Interest Group on Programming Languages, ACM Press, June 2001.

[Hol99] David Holmes. *Synchronisation Rings: Composable Synchronisation for Object-Oriented Systems.* PhD Thesis, Macquarie University, Sydney, Australia, 1999.

[Hur95] Walter Hürsch and Cristina Videira Lopes. *Separation of Concerns.* Northeastern University technical report NU-CCS-95-03, Boston, February 1995.

[Hur96] Walter L. Hursch and Linda M. Seiter. Automating the Evolution of Object-Oriented Systems. In Kokichi Futatsugi and Satoshi Matsuoka, editors, *Object Technologies for Advanced Software, Second JSSST International Symposium*, volume 1049 of *Lecture Notes in Computer Science*, pages 2-21. Springer-Verlag, 1996.

[Joh88] Ralph E. Johnson and Brian Foote. Designing Reusable Classes. In *Journal of Object-Oriented Programming*, volume 1, number 2, pages 22-35, June/July 1988..

[Kaf03] The Kaffe Open Source Virtual Machine Project. http://www.kaffe.org, visited August 2003.

[Kel98] Ralph Keller and Urs Hölzle. Binary Component Adaptation. In Eric Jul, editor, *ECOOP'98 - Object-Oriented programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 307-329. Springer-Verlag, 1998.

[Kic91] Gregor Kiczales, Jim des Rivières and Daniel Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[Kic97] G.Kiczales, J.Lamping, A.Mendhekar, C.Maeda, C.Lopes., J.M.Loingtier and J.Irwin. Aspect-oriented Programming. In Proceedings of the 11th European Conference on Object-Oriented Programming, June 1997. Volume 1241 of *Lecture Notes in Computer Science*, pages 220-242, Springer-Verlag, 1997.

[Kic01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm and William G.Griswold. An Overview of AspectJ. In L.J.Knudsen, editor, *ECOOP 2001 - Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327-353. Springer-Verlag, 2001.

[Kle96] Jürgen Kleinöder and Michael Golm. MetaJava: An Efficient Runtime Meta Architecture for Java. In *Proceedings of 5th International Workshop on Object Orientation in Operating Systems*, 1996, pages 54-61. IEEE Computer Society Press, 1996.s

[Kle00] Albrecht Kleine. TYA JIT compiler. http://www.sax.de/~adlibit/. Visited September 2003.

[Kni02] Günter Kniesel, Joost Noppen, Tom Mens, and Jim Buckley. Unanticipated Software Evolution. In Juan Hernández and Ana Moreira, editors, *Object-Oriented technology - ECOOP 2002 Workshop Reader*, volume 2548 of *Lecture Notes in Computer Science*, pages 92-106. Springer-Verlag, 2002.

[Laf03] Donal Lafferty and Vinny Cahill. Language-Independent Aspect-Oriented Programming. In *Proceedings of the 2003 Conference on Object-Oriented Programming Systems, Languages and Applications*. ACM Special Interest Group on Programming Languages, ACM Press, 2003.

[Leh96] M.M. Lehman. Laws of software evolution revisited. In Carlo Montangero, editor, *Software Process Technology - Proceedings of the 5th European Workshop*, volume 1149 of *Lecture Notes in Computer Science*, pages 108-124. Springer-Verlag, 1996.

[Lia98] Sheng Liang and Gilad Bracha. Dynamic Class Loading in the Java Virtual Machine. In *Proceedings of the 1998 Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 36-44. ACM Special Interest Group on Programming Languages, ACM Press, 1998.

[Lia99] Sheng Liang. *The Java Native Interface: Programmer's Guide and Specification.* Addison-Wesley, 1999.

[Lie86] Henry Lieberman. Using Prototypical Objects to Implement Shared Behaviour in Object Oriented Systems. In *Proceedings of the 1986 Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 214-223. ACM Special Interest Group on Programming Languages, ACM Press, 1986.

[Lie99] Karl Lieberherr, David Lorenz and Mira Mezini. *Programming with Aspectual Components.* Technical report NU-CCS-99-01, College of Computer Science, Northeasttern University, Boston MA, 1999.

[Lin96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* 1st edition, Addison Wesley, 1996.

[Lin99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* 2nd edition, Addison Wesley, 1999.

[Lis92] Andrew Lister. Design of Dependable Real-time Systems (extended abstract). In *Proceedings of the 14th International Conference on Software Engineering*, pages 35-36. ACM Press, 1992.

[Lop96] Cristina Videira Lopes and Karl Lieberherr. AP/S++: Case-study of a MOP for purposes of software evolution. In Gregor Kiczales, editor, *Proceedings of First International Conference on Metalevel Architectures and Reflection*, 1996.
http://www2.parc.com/csl/groups/sda/projects/reflection96/ Visited September 2003.

[Mae87] Pattie Maes. *Computational Reflection*. PhD Thesis, Vrije Universiteit Brussel, 1987. Also published as technical report 87-2, Artificial Intelligence Laboratory, Vrije Universiteit Brussel, 1987.

[Mal01] Jacques Malenfant, Maria-Teresa Segarra, and Françoise André. Dynamic Adaptability: The MolèNE Experiment. In Akinori Yonezawa and Satoshi Matsuoka, editors, *Metalevel Architectures and Separation of Crosscutting Concerns*, volume 2192 of *Lecture Notes in Computer Science*, pages 110-117. Springer-Verlag, 2001.

[Mat91] Satoshi Matsuoka, Takuo Watanabe, and Akinori Yonezawa. Hybrid Group Reflective Architecture for Object-Oriented Concurrent Reflective Programming. In P.America, editor, *ECOOP'91 European Conference on Object-Oriented Programming*, volume 512 of *Lecture Notes in Computer Science*, pages 231-250. Springer-Verlag, 1991.

[Mat93] Satoshi Matsuoka and Akinori Yonezawa. Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107-150. MIT Press, 1993.

[Mca95] Jeff McAffer. Meta-level Programming with CodA. In Walter G. Olthoff, editor, *ECOOP'95 - Object-Oriented Programming*, volume 952 of *Lecture Notes in Computer Science*, pages 190-214. Springer-Verlag, 1995.

[Mch94] Ciaran McHale. *Synchronisation in Concurrent, Object Oriented Languages: Expressive Power, Genericity and Inheritance*. PhD Thesis, Department of Computer Science, University of Dublin, Trinity College, 1994.

[Men97] Kim Mens, Cristina Lopes, Bedir Tekinerdogan, and Gregor Kiczales. Proceedings of the Aspect-Oriented Programming Workshop. In Jan Bosch and Stuart Mitchell, editors, *Object-Oriented Technology: ECOOP 1997 Workshop Reader*, volume 1357 of *Lecture Notes in Computer Science,* pages 483-496. Springer-Verlag, 1998.

[Mez97] Mira Mezini. Maintaining the Consistency of Class Libraries during their Evolution. In *Proceedings of 1997 Conference on Object Oriented Programming Systems Languages and Applications*, pages 1-22. ACM Special Interest Group on Programming Languages, ACM Press, 1997.

[Mez03] Mira Mezini and Klaus Ostermann. Conquering Aspects with Caesar. In *Proceedings of 2nd International Conference on Aspect-Oriented Software*, pages 90-99. ACM Press, 2003

[Mit97] Stuart Mitchell and Robert Stroud. Reflective Real-Time Object-Oriented Programming and Systems. In Jan Bosch and Stuart Mitchell, editors, *Object-Oriented Technology: ECOOP 1997 Workshop Reader*, volume 1357 of *Lecture Notes in Computer Science,* pages 51-54. Springer-Verlag, 1998.

[Mul95] Phillipe Mulet, Jacques Malenfant, and Pierre Cointe. Towards a methodology for explicit composition of metaobjects. In *Proceedings of the 1995 Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 316-330. ACM Special Interest Group on Programming Languages, ACM Press, 1995. Also SIGPLAN Notices volume 30 number 10, October 1995.

[Oga00] Hirotake Ogawa, Kouya Shimura, Satoshi Matsuoka, Fuyuhiko Maruyama, Yukihiko Sohda, and Yasunori Kimura. OpenJIT: An Open-Ended, Reflective JIT Compile Framework for Java. In Elisa Bertino, editor, *ECOOP 2000 - Object-Oriented Programming*, volume 1850 of *Lecture Notes in Computer Science*, pages 362-387. Springer-Verlag, 2000.

[Oka92] Hideaki Okamura, Yutaka Ishikawa and Mario Tokoro. AL-1/D: A Distributed Programming System with Multi-Model Reflection Framework. In Akinori Yonezawa and Brian C. Smith, editors, *Proceedings of the IMSA'92 Workshop on Reflection and Metalevel Architecture*. Tokyo, Japan, Nov. 1992.

[Oli99] Alexandre Oliva and Luiz Buzato. The Design and Implementation of Guaraná. In *Proceedings of 5th USENIX Conference on Object Oriented Technologies and Systems*. San Diego, USA, May 1999.

[Par72] David L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, volume 15, number 12, December 1972.

[Paw01] Renaud Pawlak, Lionel Seinturier, Laurence Duchien and Gérard Florin. JAC: A Flexible Solution for Aspect-Oriented Programming in Java. In Akinori Yonezawa and Satoshi Matsuoka, editors, *MetaLevel Architectures and Separation of Crosscutting Concerns*, volume 2192 of *Lecture Notes in Computer Science*, pages 1-24. Springer-Verlag, 2001.

[Paw99] Renaud Pawlak, Laurence Duchien and Gérard Florin. An Automatic Aspect Weaver with a Reflective Programming Language. In Pierre Cointe, editor, *Meta-Level Architectures and Reflection*, volume 1616 of *Lecture Notes in Computer Science*, pages 147-149. Springer-Verlag, 1999.

[Pop01] A.Popovici, Th.Gross and G.Alonso. *Dynamic Homogenous AOP with PROSE*. Technical report, Department of Computer Science, Federal Institute of Technology, Zurich, March 2001. http://www.inf.ethz.ch/department/IS/iks/publications/proseTR01.ps. Visited September 2003.

[Pop02] Andrei Popovici, Thomas Gross and Gustavo Alonso. Dynamic Weaving for Aspect-Oriented Programming. In *Proceedings of 1st conference on Aspect-Oriented Software Development*, pages 141-147. ACM Press, 2002.

[Pri97] Margaretha W. Price and Steven A. Demurjian, Sr. Analyzing and Measuring Reusability in Object-oriented Design. In *Proceedings of 1997 conference on Object-Oriented Programming Systems, Languages and Applications*, pages 22-33. ACM Special Interest Group on Programming Languages, ACM Press, 1997.

[Red03] Iguana/J documentation. http://www.dsg.cs.tcd.ie/~redmondb/iguanaj/ or http://www.iguanaj.org. Visited September 2003.

[Rob99] Bert Robben, Bart Vanhaute, Wouter Joosen, and Pierre Verbaeten. Non-functional Policies. In Pierre Cointe, editor, *Meta-Level Architectures and Reflection*, volume 1616 of *Lecture Notes in Computer Science*, pages 74-92. Springer-Verlag, 1999.

[Sch01] Tilman Schäfer. *Supporting Meta-types in a Compiled Reflective Programming Language*. PhD Thesis, Department of Computer Science, University of Dublin, Trinity College, 2001.

[Seg89] M.E.Segal and O.Frieder. Dynamic Program Updating: A Software Maintenance Technique for Minimizing Software Downtime. In *Journal of Software Maintenance: Research and Practice*, volume 1, pages 59-79. John Wiley & Sons, 1989.

[Shu98] Kazuyuli Shudo. shuJIT - Java Just-in-Time Compiler. http://www.shudo.net/jit/. Visited September 2003.

[Smi82] Brian C.Smith. *Reflection and Semantics in a Procedural Language*. PhD Thesis, Massachusetts Institute of Technology, Cambridge Massachusetts, 1982.

[Spe02] Andreas Speck, Elke Pulvermüller and Mira Mezini. Reusability of Concerns. Workshop on Aspects & Dimensions of Concerns at ECOOP 2002 (position paper).

[Str93] Robert Stroud. Transparency and Reflection in Distributed Systems. In *ACM SIGOPS Operating Systems Review*, volume 27 number 2, April 1993, pages 99 - 103. ACM Press, 1993.

[Str96] R.J.Stroud and Z.Wu. Using Metaobject Protocols to Satisfy Non-Functional Requirements. In C.Zimmerman, editor, *Advances in Object-Oriented Metalevel Architectures and Reflection*. CRC Press, 1996

[Sun97] Java Native Interface Specification. Sun Microsystems, 1997. http://java.sun.com/j2se/1.3/docs/guide/jni/spec/jniTOC.doc.html. Visited September 2003.

[Sun98] Java Core Reflection. Sun Microsystems, 1998. http://java.sun.com/j2se/1.3/docs/guide/reflection/spec/java-reflectionTOC.doc.html. Visited September 2003.

[Sun99] Java Virtual Machine Debug Interface Reference. Sun Microsystems, 1999. http://java.sun.com/products/jpda/doc/jvmdi-spec.html. Visited September 2003.

[Sun01] The Java HotSpot Virtual Machine. Sun Microsystems, 2001. http://java.sun.com/products/hotspot/docs/whitepaper/Java_HotSpot_WP_Final_4_30_01.html. Visited September 2003.

[Sun03] Java 2 Platform, Standard Edition version 1.3.1. Sun Microsystems, 2003. http://java.sun.com/j2se/1.3/. Visited September 2003.

[Tan01] Eric Tanter, Noury Bouraqadi-Saâdani and Jacques Noyé. Reflex - Towards an Open Reflective Extension of Java. In Akinori Yonezawa and Satoshi Matsuoka, editors, *MetaLevel Architectures and Separation of Crosscutting Concerns*, volume 2192 of *Lecture Notes in Computer Science*, pages 25-43. Springer-Verlag, 2001.

[Tan03] Eric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. Partial Behavioral Reflection: Spatial and Temporal Selection of Reification. In *Proceedings of the 2003 Conference on Object-Oriented Programming Systems, Languages and Applications.* ACM Special Interest Group on Programming Languages, ACM Press, 2003.

[Tar00] Peri Tarr, Maja D'Hondt, Lodewijk Bergmans and Cristina Videira Lopes. Workshop on Aspects and Dimensions of Concern: Requirements on, and Challenge Problems for, Advanced Separation of Concerns. In Jacques Malenfant, Sabine Moisant and Ana Moreira, editors, *Object-Oriented Technology*, volume 1964 of *Lecture Notes in Computer Science*, pages 203-240. Springer-Verlag, 2000.

[Tar00a] Peri Tarr and Harold Osher. Hyper/J User and Installation Manual. IBM Research, 2000. http://www.alphaworks.ibm.com/tech/hyperj. Visited September 2003.

[Tat00] Michiaki Tatsubori, Shigeru Chiba, Marc-Olivier Killijian and Kozo Itano. OpenJava: A Class-based Macro System for Java. In Walter Cazzola, Robert J.Stroud and Franceso Tisato, editors, *Reflection and Software Engineering,* volume 1826 of *Lecture Notes in Computer Science*, pages 117-133. Springer-Verlag, June 2000.

[Tat01] Michiaki Tatsubori, Toshiyuki Sasaki, Shigeru Chiba and Kozo Itano. A Bytecode Translator for Distributed Execution of "Legacy" Java Software. In L.J.Knudsen, editor, *ECOOP 2001 - Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 236-255. Springer-Verlag, 2001.

[Tho94] Laurent Thomas. Inheritance Anomaly in True Concurrent Object Oriented Languages: A Proposal. In *Proceedings of IEEE TENCON'94*, pages 541-545, IEEE, August 1994.

[Vie98] John Viega, Bill Tutt, and Reimar Behrends. *Automated Delegation is a Viable Alternative to Multiple Inheritance in Class Based Languages.* University of Virginia, Department of Computer Science, technical report CS-98-03, March 1998.

[Wat88] Takuo Watanabe and Akinori Yonezawa. Reflection in an Object-Oriented Concurrent Language. In Norman Meyrowitz, editor, *Proceedings of the 1988 Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 306-315. ACM Press, September 1988. Also SIGPLAN Notices 23(11), November 1988.

[Wel99] Ian Welch and Robert Stroud. From Dalang to Kava - the Evolution of a Reflective Java Extension. In Pierre Cointe, editor, *Meta-Level Architectures and Reflection*, volume 1616 of *Lecture Notes in Computer Science*, pages 2-21. Springer-Verlag, 1999.

[Wel01] Ian Welch and Robert J. Stroud. Kava - Using Byte code Rewriting to add Behavioural Reflection to Java. In *Proceedings of 6th USENIX Conference on Object Oriented Technologies and Systems*. San Antonio, USA, January 2001.

[Yel96] Frank Yellin. *The Java Native Code API*. Sun Microsystems 1996. http://java.sun.com/docs/jit_interface.html. Visited September 2003.

A work is never completed except by some accident such as weariness, satisfaction, the need to deliver, or death.

- Paul Valéry.

Recollection, Collected Works, volume 1, 1972.