Coupling, Code Reuse and Open Implementation in Reflective Systems

Youssef Hassoun

A Thesis Submittrd in Fulfilment of the Requirements of
Doctor of Philosophy
in the
UNIVERSITY OF LONDON

March, 2005
School of Computer Science and Information Systems
BIRKBECK COLLEGE

Abstract

Little research has been conducted in investigating the implications of employing reflection in object-oriented (OO) systems on software qualities such as coupling and reuse.

In this thesis, we investigate the reflective capabilities of Java as a representative of mainstream OO languages and propose a behavioural reflection model, which complements the language's inextensible, introspective, structural reflection model. We show that reflective systems based on the proposed model support the principle of open implementation, and exhibit less coupling and a higher level of code reuse. Programming examples explain how the behaviour of such systems can be customised or adapted without changing their default behaviour or structure. We then show that the model is also applicable to distributed systems allowing for generic coding on the server side.

We address the question of assessing coupling and code reuse qualities of reflective systems quantitatively. For this purpose, we define a dynamic coupling metric and a measuring tool that allows the collection of object coupling data at runtime. A case study shows that our coupling metric can be measured automatically and confirms the hypothesis about the relatively low coupling of reflective systems in comparison with equivalent systems based on the classical non-reflective programming model. As for code reuse, we define OO reuse metrics by extending a set of metrics previously developed for procedural languages. A case study is used to confirm the hypothesis that reflective systems are more reusable than non-reflective systems exhibiting the same behaviour.

Contents

List of Tables

List of Figures

Acknowledgements

I would like to thank all, who through their support and help have allowed me to bring this work to the end, in particular my supervisors Dr. Roger Johnson and Dr. Steve Counsell for their constant support, encouragement, patience and collaboration.

I am grateful to the School of Computer Science and Information Systems at Birkbeck College for the Teaching Assistantship (TA) and the fruitful research atmosphere- Professor George Loizou for his interest and support, and Professor Alexandra Poulovassilis, Head of the School of Computer Science and Information Systems, for her encouragement and support, Roger Mitton, who managed the TA group, for his excellent leadership and Dr. Constantinos Constantinides for his collaboration at an early stage of my research.

Many thanks to the System Group in the School namely Phil Gregg, Phil Docking, Andrew Watkins and Graham Sadler for providing the computing facilities and their friendly and prompt response in case of problems. Mrs Betty Walters and the administrative group for providing an excellent working environment.

I would like to thank Professor Laurie Hendren, Professor of Computer Science, McGill University in Canada and Visiting Fellow at Oxford University Computing Laboratory, for her interest in the implementation of measuring tools for collecting coupling data at runtime.

I benefited from seminars organised by the British Computer Society (BCS) Advanced Programming Specialist Group. It was an invaluable opportunity to meet and discuss current research topics with the group and with speakers of international profile. In this context, many thanks to my supervisor Dr. Roger Johnson and to all other members of the organising committee especially chairman Professor John Florentin.

The research reported in this thesis has benefitted from anonymous reviewers. I would like to thank those reviewers, who through their incisive and useful critic have contributed significantly to improving the submitted papers and consequently the work presented here.

Lastly and by no means least, I thank my family for their encouragement, boundless love and the beautiful times during the few vacations from work.

Chapter 1

Introduction

1.1 Problem and motivation

It is widely accepted that most research conducted in the past on evaluating software quality of object-oriented (OO) systems can be characterised by the following features:

1. The definition of static metrics at the *class* level for measuring various quality attributes.

2. The analysis of system qualities where dynamic behaviour was only a minor and inconsequential issue.

Moreover, little research has been conducted into investigating software qualities of reflective systems. Only recently has research been focused on dynamic object metrics with the aim of investigating the impact of reflection and OO runtime mechanisms such polymorphism (in combination with inheritance) on system quality attributes such as coupling, reuse and fault-proneness [13, 14, 55, 81].

Reflective systems incorporate structures representing themselves. They are characterised by a variable dynamic behaviour, where, with the support of reflection, retrieval of program information and change of program behaviour as well as structure are permitted during execution. Maes [76] defines reflection (also known as meta-programming) as "the process of reasoning about and/or acting upon itself".

There are various reasons for considering the reflective programming model. Reflection makes software systems adaptable to change and allows for customising systems' behaviour according to changing conditions. Adaptability means adjusting the system to new requirements in a non-invasive manner thus potentially reducing the costs of change. In the context of programming language design, Kiczales et al. [67] observed that

> "as languages become higher and higher level and their expressive power becomes more and more focused the ability to cleanly integrate something outside of the language's scope becomes more and more limited. An open language implementation that provides control over the language permits the programmer to shift the language focus somewhat so that it becomes a better vehicle for expressing what they have in mind."

Many OO programming languages including popular (or mainstream) languages such as Java [7], C++ [92], C# [30] and Smalltalk [48] support reflection to varying degrees. For example, the Meta-Object Protocols (MOPs) of the Common Lisp Object System (CLOS) [21, 68] constitute an open system which allows programmers to tune the language according to a user's needs. Java and C# provide library packages that permit programs to inspect their own structures. C++ supports the detection of objects' types at runtime through the Run-Time Type Information (RTTI) mechanism. Smalltalk-80 provides a rich reflective interface allowing programs to compile classes and methods on the fly, change the class of instances and the inheritance hierarchy of a class.

The subject matter of this thesis is the investigation of the reflective capabilities of mainstream languages through the reflection models of Java, and the study of coupling and reuse attributes of reflective systems based on the language's reflection models. We concentrate on the following features:

1. Reflective code can be implemented in a generic manner, thus enhancing the level of code reuse.

2. In reflective systems, runtime object interactions are not necessarily or entirely dictated by static class relationships, but are also determined by runtime mechanisms provided by the language. Binding mechanisms which assign meta-objects their application objects allow for change of structure and/or behaviour of the application objects at runtime. Runtime flexibility reduces coupling and enhances reuse.

3. Reflective systems inherently support the principle of separation of concerns; meta-objects at higher levels can be assigned some tasks on behalf of the objects they represent, i.e., base objects residing at lower levels of the reflective tower. Applying the principle of separation of concerns promotes system modularity.

4. Reflection allows programmers to take control over program implementation. As systems evolve, they must be adapted to changes in the environment or intended use. Reflection supports the principle of open implementation [69] which allows system customisations as add-on without modifying default behaviour.

Some authors differentiate between a reflective system and a meta-system. Cointe [36], for example, represents a meta-system as a system consisting of two parts in which the object system representing part of the real world is explicitly separated from the meta representation part. A reflective system, on the other hand, is represented as one entity. This difference is irrelevant to the work presented in this thesis and we shall only refer to reflective systems henceforth.

Although reflective techniques and mechanisms play a major role in constructing generic software systems such as debuggers, networking facilities and frameworks etc., reflection has not been, and still is not, part of common programming practice. The reasons for this can be stated as:

1. Reflection is not an explicit part of the OO core paradigm.

2. In the 1980s and in the 1990s, many industrial systems were developed in Smalltalk, C++ and Java. With Smalltalk-80, the meta-class capabilities of Smalltalk-72 were restricted and meta-programming was discouraged [24]. In C++, meta-programming is restricted to the *template* mechanism. No explicit representation of program structure is permitted besides the class code representing the problem domain. The RTTI mechanism of C++ can be considered to be only a weak reflective mechanism which allows programs to detect the type of their objects at runtime. Java supports introspection without allowing full structural reflection. A program can inspect and reason about its structure at runtime, but the program is not permitted to change this structure. In Chapter 3, we show that Java's dynamic proxies, first introduced in the Java Developing Environment (JDK) version 1.3, provide the basis for defining a behavioural reflection model [53, 57].

3. Reflective systems could be considered unstable and complex. An end-user could potentially change the system and there is relative difficulty of understanding and learning the coherence and connection between the different levels (meta-levels and base level) comprising a reflective system.

4. Reflection has been confined to particular research areas such as Artificial Intelligence (AI). Many languages supporting reflection, e.g., LISP and its derivatives, are partly developed for research interests and play only a minor role in developing industrial systems.

In this thesis we argue that combining the secure programming paradigm supported by popular languages (strong typing, compile-time checking, etc.) with the reflective possibilities provided by the same languages allow for the construction of robust and flexible systems. Such systems, we claim, are qualitatively better than pure static systems in terms of coupling and the degree of code reuse. This claim is based on the features listed in the previous paragraphs and will be pursued in the course of the thesis.

We choose Java as a representative of mainstream programming languages and study the language's reflective models. The reasons for choosing Java are among others: Java's multiple platform portability, support of reflection (though limited) and the richness of the language component library as well as Application Programming Interfaces (APIs). The language library covers a wide range of application domains such as distributed programming including Web and Internet applications as well as multi-threading. Moreover, Java is supported by a large programming community and is, at present, a popular and widely used programming language.

The Thesis comprises two main components. Firstly, it explores the reflective capabilities of Java as a representative of mainstream OO programming languages. Secondly, it employs the software metrics approach to assess quantifiably the impact of employing reflection in constructing OO systems. The latter component serves as a tool for demonstrating the viability of the former. The use of metrics in the second half of the Thesis strengthens our claims about the benefits of employing reflection made in the first half of the Thesis.

1.2 Thesis objectives and contribution

The objectives of this thesis, which at the same time represent its contribution, are to show the advantages of employing reflection as part of an OO programming paradigm. In the remainder of this thesis, we:

1) explore the reflective capabilities of Java as a representative of mainstream languages and show that, in addition to the inextensible structural reflection model, there is a behavioural reflection model [53, 57],

2) investigate the software qualities of reflective systems, their coupling and reuse qualities by proposing appropriate metrics. We define appropriate metrics and demonstrate that use of reflective techniques quantitatively lessens coupling and provide greater reuse opportunities than OO systems using standard programming techniques [54, 55]

3) validate the proposed metrics theoretically against sets of formal criteria. For our empirical validation, we present case studies and use a measuring tool to collect dynamic coupling data automatically [56].

We believe that as reflective techniques become more accessible to the programming community, reflection will be used more intensely in implementing industrial applications than before. Our goal is to find appropriate programming practices using reflection and, ultimately, to show that the reflective programming model supports relatively low coupling and high code reuse.

1.3 Thesis outline

The thesis is organised as follows. In the next chapter related work is discussed. Here, the results of the research work presented in the thesis are compared with the work of other researchers in the fields of reflection models and software metrics. In Chapter 3, the reflection capabilities of Java are reviewed and a behavioural reflection model based on dynamic proxies presented. The question of whether the proposed model supports *open implementation design* is addressed. In Chapter 4, we show that the proposed reflection model is applicable to distributed environments. Examples of distributed applications based on RMI, CORBA and Java Servlets technologies are presented showing that the model, as in the single application case, supports generic coding and open implementation. Chapters 5 and 6 deal with the issue of measuring software attributes of coupling and code reuse of systems built on the proposed reflection model. The proposed metrics serve as a means of evaluating the reflective programming paradigm and confirm our hypothesis about their coupling and reuse qualities. In Chapter 5, a dynamic coupling metric is defined and its theoretical validation against two sets of formal criteria discussed. For the metric's empirical validation, we developed a measuring tool and used it in a case study to collect coupling data at runtime. In Chapter 6, we propose OO code reuse metrics, validate these theoretically and apply them to a case study where we compare the code reuse level of two functionally identical systems having different architectural characteristics, namely, reflective and non-reflective. Finally, in Chapter 7, we draw conclusions from our work and discuss future research directions.

Chapter 2

Related Work

The contribution of this thesis referred to in the introduction can be split into two parts. The first involves proposing a behavioural reflection model in Java and showing that this model supports the open implementation design principle and is applicable to distributed environments as well as to single Java Virtual Machine (JVM) applications. The second part involves investigating the software qualities of coupling and reuse of reflective systems based on the proposed model. A dynamic coupling metric is defined and validated theoretically. For the empirical validation, a tool is developed and used in a case study to show that coupling data can be collected at runtime. OO code reuse metrics suitable for assessing reuse percentages of entire systems were also defined and validated. The purpose of this chapter is to relate our work to the work of other researchers in the fields of reflection models and software metrics and contrast with that of others.

As for reflection models, Maes [76] introduced a reflection model in the context of the 3-KRS language where each object was assigned a meta-object representing its structure and the way it handled messages. In our model, an application object can be assigned zero or more meta-objects representing the object's behavioural state. Our meta-objects provide no structural representation of their referents residing at the base level.

The nature of meta-objects in Maes' model is not specified and 3-KRS is not a class-based. Ferber [44] observed that there are two types of reflection models representing solutions to the problem of adapting the 3-KRS reflection model to class-based languages. The two solutions are based on identifying the meta-object:

i. with the receivers' class (first model), or

ii. with an instance of a specific class called META-OBJECT (second model).

In our model [53, 57], there are no meta-classes allowing for a dynamic creation of instance classes at runtime. We do not alter the underlying object model of Java where classes are compile-time entities representing the template behaviour of their instances (objects) created at runtime by invoking the method `new()`. Our meta-objects are not classes but instances of type `java.lang.reflect.-InvocationHandler`. They are similar to META-OBJECT instances, in that they include operational and control information about their referents, leaving the structural information (i.e. the description of the internal structure of the receiver) to the receiver's class. The receiver class is a `java.lang.reflect.Proxy` which acquires the type of the application object upon its creation. In Java, the structural information can be obtained through the introspective API described in the next chapter.

The proposed reflection model in this work does not represent an extension of the Java language, as for example in the case of AspectJ, [4] or OpenJava [34, 95]. The two languages are prominent examples used in Aspect-Oriented Programming (AOP) [70] and meta-programming, respectively. Language extensions make use of an existing language environment (compiler, tools, class-hierarchy etc.) to provide new programming features not available in the original language. Although both language extensions add to the power of Java in terms of structure and mechanisms, extra effort is needed to allow for a flexible structural and behavioural customisation of application objects at runtime. In our model, although restricted to behavioural reflection, meta-objects can be created and assigned to application objects at runtime.

In our model, application objects are encapsulated by proxies which delegate control to meta-objects. In a similar fashion, Pascoe [86] used the exception handling mechanism of Smalltalk-80 to define *encapsulator* objects and trap messages sent to the encapsulated (base) objects. In Smalltalk-80, exception handling works as follows: when an object receives a message for a method it has not defined, the message `#doesNotUnderstand()` is sent to the receiver with the original message and its arguments. The protocol of encapsulators (including that of their predecessors) does not overlap with the protocol of base objects. Consequently, messages are trapped by the `#doesNotUnderstand()` method object which can be inspected and redefined to customise the behaviour of the encapsulated objects.

As in the case of Smalltalk-80, the exception handling mechanism of CLOS can be used to encapsulate base objects with the aim of defining meta-objects to control the base objects. In CLOS, *generic functions* are used for polymorphic operations called messages in traditional OO languages. The exception handling mechanism works similarly to that of Smalltalk-80 and can be used to customise objects' behaviour. However, there is an important difference: in CLOS, there is no single receiver of a message. Object interactions are based on generic functions. The methods to be performed are determined by the types (and possibly the identities) of all their parameters. This so called *multiple dispatch* mechanism is in contrast to the *single dispatch* mechanism supported by most of OO programming languages including Smalltalk. If a generic function is called and no methods are applicable, the function `NO-APPLICABLE-METHOD()` is invoked with parameters including the generic function object for which no applicable method was found and the list of arguments to that generic function. The generic function `NO-APPLICABLE-METHOD()` is not intended to be called by programmers. However, programmers may write methods for it by which objects' behaviour can be modified.

Some languages provide mechanisms that support customisation of objects' behaviour. For example, CLOS and early dialects of OO Lisp such as Flavors [101] and Lisp Object-Oriented Programming System (LOOPS) [23] provide a mechanism that allow modification of object behaviour using *auxiliary methods* together with method qualifiers such as *:before, :after and :around.* The behaviour of an object can be modified by combining the primary method of its generic function with auxiliary methods.

CLOS, through the MOP interface, constitutes an open system which allows for modification of language abstraction and its implementation [67, 68]. Customising the underlying programming language instead of customising applications directly is not addressed in this thesis and is left for future research.

There is a close connection between reflective systems or meta-level architectures and systems based on the AOP approach. A reflective system can be characterised by its separation of meta-objects from the objects they represent, i.e., base objects. AOP separates functional modules of an application (known also as application core concerns) from tasks which do not directly participate in a system's primary function (classified as non-functional features). The latter usually cross-cut the functional modules and are thus hard to modularize using traditional OO abstraction mechanisms. As in AOP, reflection can systematically separate such non-functional (cross-cutting) features from the rest of an application. We remark that separating cross-cutting aspects from an application does not necessarily reduce coupling and lead to loosely coupled and reusable aspects. However, combining Java's introspective API with the support of the principle of separation of concerns provided by AspectJ allow for the implementation of generic reusable aspects [52]. The same applies to implementing generic hyperslice definitions in Hyper/J [94], as shown by Hassoun and Constantinides in [51].

In [19, 20], Biggerstaff discusses the reuse problem of scaling component libraries such as APIs and foundation class libraries. Extending reuse libraries in both component sizes and feature variations with acceptable performance leads to the combinatorial explosion problem or the scaling problem. The key problem of reuse libraries is that they are based on static components with all corresponding activity, such as composition, refinement and specialization being localized to predetermined areas of a target program. The solution to combinatorial explosion of the component library, according to Biggerstaff, is to build all feature variations into highly general components and leave component generation and composition decisions until runtime. The open implementation approach supported by our reflection model provides a solution to the scaling problem of Java libraries. The question of how our model could be used to counter potential scaling problems associated with Java libraries is not addressed, since the issues lie outside the scope of this thesis.

In this thesis, we address some of the outstanding issues related to dynamic coupling metrics. In the context of validating dynamic coupling metrics, other researchers have used different techniques for collecting metric data at runtime. Arisholm [13] uses Smalltalk features specific to the IBM-VisualWorks programming environment to collect coupling data at runtime by intercepting all the object messages sent and received. In [14], Arisholm et al. define a set of dynamic coupling measures. To collect dynamic coupling data, the authors use a tool in which the JVM loads a library of routines that are called whenever specified internal events occur. The tool remains separate from the program under study (the sample) and interacts with it through the JVM at runtime. In both papers, the sample program remains separate from the process code collecting data. Our measuring tool used in the context of empirical validation of the dynamic coupling metric (DCM) is developed in a generic way independently of the sampled program. We do not modify the source code to collect coupling data at runtime [55, 56]. However, our approach is different from Arisholm et al. in that it does not interfere in the JVM processing at runtime. We also do not use vendor-specific features of the Java language. The specification of events and the merging of source code with the program sample for data collection are realized at compile-time. In our approach, knowledge of the sample program is necessary, because the default DCM values depend on the class structure.

Mitchell and Power define a number runtime metrics for coupling and cohesion [81]. They use the Java Platform Debug Architecture (JPDA), available as part of the Sun Microsystems Java 2 Standard Edition [7], to obtain a runtime profile of the Java program under consideration. Their approach is restricted to Java and requires extra tools for program samples written in other languages. In contrast, our approach is applicable to a range of different languages. The AOP techniques used for developing the data measuring tool support different languages. AOP language extensions for C++, Smalltalk-80 and prototypes for other languages are also available [3].

The research described in this thesis relates to reuse and the measurement of code reuse in systems where reflective techniques are employed. A number of metrics in the past have tried to capture the extent of software reuse. The OO metrics of Poulin [87] include a Reused Source Instructions (RSI) metric as well as various other development project based reuse metrics. The metric is concerned with estimating the financial benefit of software development within "product lines". Withey [104] defines a product line as "a group of products sharing a common managed set of features that satisfy the specific needs of a selected market". Czarnecki and Eisenecker [38] observe that the concept of a product line is based on a marketing strategy and not on the technical similarities between member products. The area of software product lines is of relevance to our work, since reuse has a strong link with establishing generic traits in software that can be used in different software artefacts with only minor variation [40]. However, in our research, we are not concerned with estimating the financial benefit, but only with the impact of using reflective techniques on code reuse.

The most commonly referenced OO metrics are those of Chidamber and Kemerer [35] who proposed two inheritance-related metrics: Depth of Inheritance Tree (DIT) and Number of Children (NOC). Both were intended to give the designer an impression of expected reuse in an application through specialisation. Chidamber and Kemerer also proposed the Coupling Between Objects (CBO) metric as an indicator of reuse potential. However, we do not view CBO as an indicator of direct reuse and together all three measures can only be used for individual classes. They are not therefore suitable to quantify the code reuse of an entire program. In addition, the metrics are design metrics; that is, they are useful to assess the quality of software at the design level but are not adequate for code reuse purposes.

Kang and Bieman investigated the impact of the shapes of inheritance trees on code reuse and proposed corresponding reuse measures [63]. The authors defined a set of metrics which classified forests of inheritance trees into a set of five shape classes. The investigation provided useful guidance on the appropriate shape of an inheritance hierarchy for supporting code reuse. However, inheritance is not the only mechanism supporting code reuse. Aggregation, parametric polymorphism and reflection (if supported by the implementation language) also achieve the same aim and support a motivation for this research namely the development of appropriate metrics to model these OO features.

Chapter 3

Reflection in Java

In this chapter, we investigate the reflective programming model by introducing the notion of reflective object-oriented system. The underlying structure of such a system is a reflective tower consisting of several levels. Objects at higher levels are termed meta-objects and those residing in the base level are called base objects. Further, we consider Java's reflection model, review the introspective structural reflection interface and define a behavioural reflection model using dynamic proxies. Invocation handlers of proxies are interpreted as meta-objects. The resulting reflective tower is two-level where proxy objects (including invocation handlers) reside in the meta-level and application objects populate the base level. Meta-objects can be used to customise the behaviour of objects they represent (their referents). We give two examples showing the scope of applicability of our model and discuss the model's open implementation design feature.

The chapter builds on the related work described in Chapter 2 in relation to reflection models. It contributes to the overall thesis by providing the theoretical foundation of the reflection model which we will be using in the following chapters. In the first section, the concepts of reflection and reflective systems are introduced. Section 3.2 provides an introduction to proxies. Proxy's roles in single-application systems as well as in distributed systems including Web applications are discussed and the difference between standard and dynamic proxies stressed. Section 3.3 deals with Java's reflection model. Here, the structural reflection capabilities are reviewed and a behavioural reflection model based on dynamic proxies is proposed. Section 3.3.4 compares our approach to behavioural reflection to that of building an MOP interface for customising the behaviour of CLOS as the underlying programming language. In Section 3.4, we consider the design principle of *open implementation* and address the question of whether our model supports open architectures, i.e., systems that follow the open implementation design principles. Two examples about customising applications' behaviour are presented showing that the proposed reflection model supports open architectures. In Section 3.5, the impact of applying the proposed model on systems' attributes, such as coupling and reuse is discussed and similarities between our approach and aspect-oriented programming paradigm are identified. We end the chapter with concluding remarks in Section 3.6.

3.1 Reflection

Smith [91] defines reflection as "a process's integral ability to represent, operate on, otherwise deal with itself in the same way that it represents, operates on and deals with its primary subject matter". In [22], Bobrow et al. define reflection as "the ability of a program to manipulate as data something representing the state of the program during its own execution. There are two aspects of such manipulation: introspection and intercession. Introspection is the ability for a program to observe and therefore reason about its own state. Intercession is the ability for a program to modify its own execution state or alter its own interpretation or meaning. Both aspects require a mechanism for encoding execution state as data; providing such an encoding is called reification". Maes [76] defines a reflective system as a system having itself as application domain and that is causally connected with this domain.

For example, in CLOS, functions with a class as argument such as `class--direct-superclasses()`, `class-direct-slots()` and `class-direct-methods()` allow a program to extract and inspect, respectively, a list of the immediate superclasses, a list of non-inherited slots[1] and a list of non-inherited methods. Java provides an introspective interface which allow a program to get the same type of information about each of its runtime objects. We discuss Java's introspective interface as part of the language structural reflection model in Section 3.3.1.

On the other hand, CLOS functions such as `add-method()` and `remove-method()` can be used to change the structure of a generic function[2] by adding and removing corresponding methods defined for certain classes. Another example is the standard generic function `make-instance()` which can be used to create new *class meta-objects* by specifying their class meta-object, names, superclasses and slots [68]. Unlike CLOS, where structural changes at runtime are allowed, Java's structural reflection model is restricted to introspection and does not allow programs to create new classes, change classes or hierarchies at runtime. However, behavioural change is permitted and in Section 3.3.2 we discuss how it can be implemented in Java.

In OO systems, the structural entities are classes and objects. These entities communicate by sending and receiving messages. In reflective OO systems,

[1]In CLOS, slots are synonyms for fields of a class instance, which in some OO languages are called instance variables or attributes. For more details, refer to standard books on CLOS, for example, [64].

[2]A generic function is a synonym for a message. It consists of a set of zero or more methods and provides class-specific operations.

certain structural and behavioural aspects of the base level are represented or reified as meta-objects. Depending on the reflection model, meta-objects can either be objects, i.e., normal class instances, or meta-classes. In the latter case, meta-objects provide representations defining the structure of their classes as well as handling of object messages [44]. We observe that the label meta-class is rather arbitrary and that the role of the object entity in the reflection model is more relevant. In [68], for example, Kiczales et al. refrain from calling higher classes in the CLOS hierarchy meta-classes in spite of the fact that higher classes provide representations of application objects at the base level(see Section 3.3.4 for more details). Various models of reflection are discussed in Section 3.3.

In [76, 77], Maes introduced a reflection model based on representation of objects. Objects, in conventional object-oriented programming (OOP), are representations of real world entities whereas meta-objects are representations of objects. The role of meta-objects is to observe and modify the objects they represent. Meta-computation is often performed by intercepting normal computation carried out by normal objects. Meta-objects trap actions of their referents, perform meta-computation and thereby customise or substitute referents' actions. In Maes's model, meta-objects themselves are objects and can also be represented by meta-objects. The model leads to an infinite repetition of meta-objects similar to the infinite tower of 3-Lisp [90]. Objects populating the base level are called base objects and those residing at the higher levels termed meta-objects. Base objects provide a representation of the application domain and thus perform computation on entities of this domain. Meta-objects perform computation on objects residing in the lower level. The code dealing with meta-objects is called a meta-program. The program on which reflective computation is performed is called the base program or simply the application.

Compared to static systems exhibiting the same interface, reflective systems show a lower level of object coupling and, consequently, they are more reusable and potentially easier to maintain [54, 55].

3.2 Proxies

We use proxies to develop a behavioural reflection model. This section serves as an introduction to proxies and their use in software.

A proxy is a program that provides a communication bridge allowing different applications to engage one another and exchange data. The idea of a proxy can be utilized in many different ways; it can be used to protect applications from each other as a firewall or it can cache exchanged data as a caching proxy. Equally, a proxy can be used as a translator between applications with different communication strategies, for example, Internet and legacy systems.

In OOP, proxies are entities that act as intermediaries between client objects and target objects. The role of a proxy as a communication bridge between applications has many uses in software. From a design perspective, a proxy is a design pattern [47]; it describes a general solution to common design problems that occur repeatedly in different contexts. In distributed systems, including Web applications, a proxy is a service that sits between application servers and clients. This service receives requests from clients and makes requests to servers on behalf of the clients. Figure 3.1 shows a typical situation for two classes, `Client` and `Target`, where clients request service from target objects to fulfil their tasks. The interface `ITarget` defines the list of services a target object can provide. Here, only class names are shown and Unified Modelling Language (UML) notation is used [10].

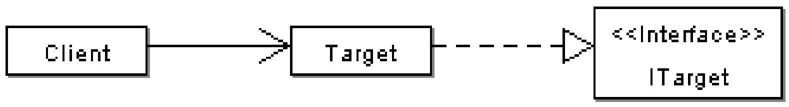

Figure 3.1: Clients request services from Target objects to fulfil their tasks

During the software life-cycle, proxies are useful as a means by which software can accommodate new changes. For common problems like tracing, adapting classes, and in general, adding new features to an existing system, a proxy provides an adequate solution. With a proxy, the code of existing classes need not be modified; it is desirable to avoid changing existing code to reduce potential testing and maintenance costs. Figure 3.2 depicts the modified class structure representing the proxy solution. The `Proxy` class implements the same interface and uses a `Target` object as a delegate. The proxy can manipulate clients' requests before dispatching control over to its delegate. If adapting a new class to replace `Target` is required, the proxy retains the client interface by implementing `ITarget` and delegates the client request to the object of the new class.

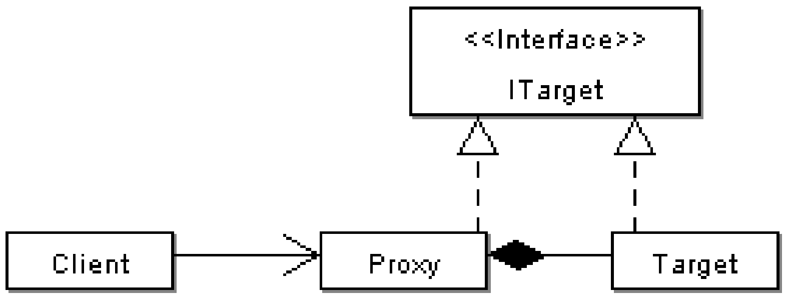

Figure 3.2: A proxy can manipulate clients' requests before dispatching control

In a networking context, a proxy can be utilized to support security, to optimize performance or to act as a translator to support the embedding of legacy systems into the Internet. As firewalls, proxies can be used for building a security layer to prevent unauthorized access to corporate Web services. Caching proxies are employed to reduce the network load as well as the server load and HTTP proxies can be used to translate clients' HTTP-requests into the native language of legacy systems and return results back as HTML pages. Proxies can also be used to implement lazy instantiation, i.e., to defer the costs of creating and initialising an object until it is actually needed.

In addition to normal proxies, Java supports dynamic proxies. A Dynamic Proxy API has been introduced in Java 2 Standard Edition (J2SE) version 1.3. Normal proxy classes are usually available as byte-codes after having been compiled from the corresponding Java source files. The byte-codes are loaded into the JVM before proxy objects are instantiated. The byte-codes of Java's dynamic proxies, on the other hand, are generated at runtime.

Figure 3.3 shows the key classes of our example involving dynamic proxies. Here, key methods are shown using a part of the UML class notation. `ProxyFactory` creates a proxy object and returns it back to the client by invoking the static method `newProxyInstance()` of the `Proxy` class. Method invocation on the created proxy instance dispatches control to the `invoke()` method of the `AssociatedHandler` object associated with the proxy. The binding between proxy and target objects is realised at runtime through parameter passing. The factory method `createProxy()` takes an object of type `java.lang.Object` as a parameter. At runtime, and with the support of polymorphism, an object of type `ITarget` can be passed. The target object is then passed further upon creating an `AssociatedHandler` object.

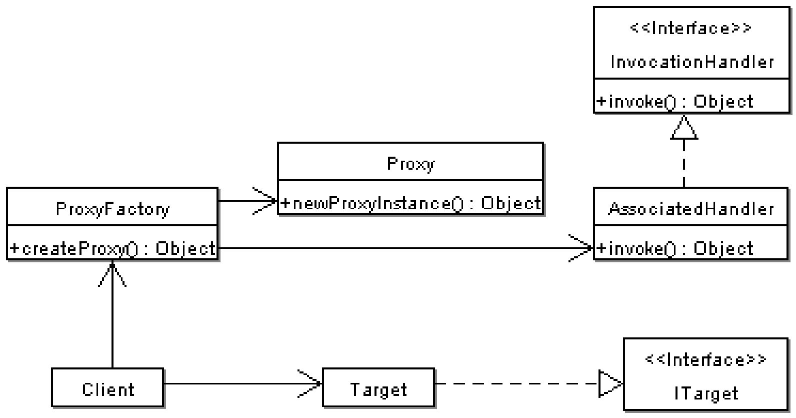

Figure 3.3: Key classes of dynamic proxies' dispatch mechanism

There are advantages to using dynamic proxies instead of normal proxies. Firstly, from an OO design point of view, the class structure of Figure 3.3 exhibits less coupling between Proxy classes and `Target` classes when compared to the static case. There is no direct relationship between these two class groups; the coupling takes place dynamically at the object level. In fact, the type of the created dynamic proxy, i.e., the array of interfaces it implements, is set at runtime with the creation of the proxy. Secondly, the dispatching mechanism of dynamic proxies allows, using polymorphism and reflection, for implementing generic behaviour independent of the targets' types. As a strongly typed OO language, Java supports subtype polymorphism (often referred to simply as polymorphism). In [38], subtype polymorphism is defined as the mechanism according to which variables of a given type can also hold objects of its subtype. Johnson and Foote [62] refer to a polymorphic procedure as an object-oriented mechanism supporting reuse. Essentially, the `invoke()` methods of the proxies' associated handlers can be seen as polymorphic procedures.

While the use of proxies is generally advantageous, there are limitations to employing proxies in a networking environment. HTTP Proxies must be properly installed before communication with legacy target systems can be established. Security constraints of firewalls could hamper the development within the corporate network. Benefits of caching proxies are constrained by several factors such as cache replacement policy, minimum document size cached and disk area allocated for caching.

3.3 Structural and Behavioural Reflection Models

In Section 3.1, we referred to two aspects of reflection, namely, introspection and intercession and gave some examples. In this section, we review the reflection models of Java in some detail and propose a behavioural reflection model based on Java's dynamic proxies.

At the class level, the reflection kernel of Java is composed of two classes `Object` and `Class`, both of which belong to the `java.lang` package. `Object` represents the ultimate superclass of the language's inheritance hierarchy and `Class` is a direct subclass of `Object`. At the object level, instances of both classes can be derived from each other by invoking appropriate methods. An `Object` instance can be created by invoking the method `newInstance()` on a `Class` object and a `Class` instance can be obtained by invoking the method `getClass()` on its object. In addition to `Object` and `Class`, Java provides a reflection package `java.lang.reflect` including classes, which support structural reflection as well as behavioural reflection.

3.3.1 Java's introspective structural reflection

As noted in Section 3.1, introspection is the ability of a program to obtain information about its own structure including its classes, its hierarchy and the instantiation relationships of its objects. Java's introspective API, which constitutes the structural reflection model, is supported by `Class` and associated classes like `Method`, `Field` and `Constructor` of the `java.lang.reflect` package.

Upon the loading of its class by the JVM, an object acquires a unique instance of `Class` representing its meta-object. By invoking the method `getClass()`, the object's class is reified allowing inspection of the structure of the system application. Since `Object` lies at the root of every inheritance hierarchy, the class of any application object can be reified and used to expose the object's structure. A complete description of the type of a base object may be obtained by invoking the various methods of `Class` on the corresponding meta-object. The methods allow us to obtain representations of the class' constructors, methods, fields, superclass and interfaces. These representations in turn provide methods that allow their structure, e.g., the types of fields and the parameter types of methods, to be discovered. The process can be continued recursively to cover the entire class hierarchy of the object under consideration.

`Class` is defined as *final* and has no public constructors, meaning that it can be neither extended by inheritance nor instantiated. Instead, `Class` objects are constructed automatically by the JVM as classes are loaded and by calls to the `defineClass()` method in the class loader. Similarly, the classes `Method`, `Field` and `Constructor` are also final and provide no public constructors. This shows that Java's introspection is limited to obtaining information about objects' structures and that the (structural) reflective model supported by these classes is fixed by the Java system and hence cannot be changed. Accordingly, the class of an object can be reified at runtime and the object's structure can be uncovered. However, an application can neither change the behaviour of its objects nor their structure.

3.3.2 A behavioural reflection model based on proxies

A behavioural reflection model allows programs to intervene in the current execution of a program in order to execute some reflective code analysing or modifying the course of events [39]. There are different implementations of the model depending on the programming language. In the following, we show that Java supports behavioural reflection through dynamic proxies, which include the `Proxy` class and the `InvocationHandler` interface.

Ferber [44] sets three conditions for defining an OO reflective architecture:

1. what is the nature of meta-objects and/or meta-communications, and what is their structure and behaviour?
2. how is the handling of messages and lookup of methods described at the meta-level related to basic message passing?
3. when does the system use meta-object and/or meta-communication?

In [53], the author of this thesis with his supervisors proposed a two-level OO reflective model based on Java's dynamic proxies classes and dispatching mechanism. In Figure 3.4, proxy objects at the meta-level include invocation handlers associated with proxies created at runtime. The proposal was based on the following propositions:

1'. invocation handlers of dynamic proxies are meta-objects,

2'. message handling makes use of the dispatching mechanism provided by dynamic proxies. Messages are handled at the base level if their receivers are base objects. They are dispatched to meta-objects and handled at the meta-level if sent to proxies representing application objects at the base level. Eventually, meta-objects pass control to their referents,

3'. events that trigger the meta-level are method calls on proxy objects.

Upon instantiating a proxy, a base object is associated with the meta-object thus reified. The proxy object acquires the type(s) of the base object without implementing its interfaces explicitly.

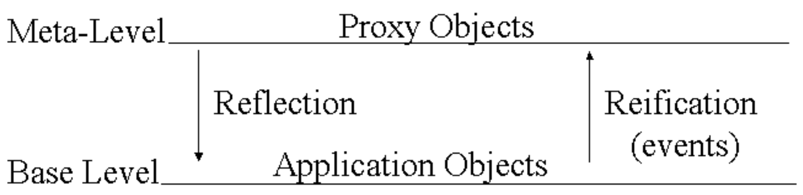

Figure 3.4: A two-level reflective architecture with proxies

3.3.3 Meta-objects as a means of reflection

Our behavioural reflection model complements the structural reflection model of Java. Both models are based on the meta-object approach, initiated by Maes [76, 77]. In our model, a meta-object can have many or no referents, and, conversely, a base object can be represented by an arbitrary number of meta-objects.

Ferber [44] identifies three different models of reflection in the domain of OO languages.

i. Classes as meta-objects: In this model, the class of object is usually considered as its meta-object.

ii. Meta-Objects as instances of the class Meta-Object: In this model, meta-objects are defined as instances of a class called `META-OBJECT` instead of being identified with the receiver's class, i.e., with the class of the object receiving a message (case i.). The meta-objects differ from the class of the receiver and include only operational and control information.

iii. Reflection as a reification of communications: In this model, meta-objects are defined as communication objects, which can react to sending messages.

Our reflection model has similarities with the second model (ii.). This model is a variation of the meta-object approach of Maes, as observed by Ferber in [44]. Our meta-objects are instances of classes of type `InvocationHandler`. These classes are different from the classes of potential referents. They can be made to implement generic code independent of the types of the referents at the base level. This code is meant to customise the behaviour of base objects and does not include any data describing the internal structure of base objects. Further, as with Ferber's second model (ii.), our model is concerned only with the issue of how messages are intercepted and methods are applied, i.e., with behavioural reflection, leaving structural issues to Java's structural reflection model.

In the case where one base object is represented by a collection of meta-objects, the former can be associated with the meta-object collection in two different ways:

1. attaching the same base object to several meta-objects, i.e., using the object's reference as a parameter in the binding methods to all handlers,

2. first attaching the base object to one meta-object, then attaching the binding reference obtained from the first binding to a second meta-object and so on up to any number of subsequent handlers.

The first case results in different and independent reifications and requires different method invocations on the same object. The second case is more complex and results in a chaining effect triggered by a method invocation on the object. It starts with the handler to which the object instance is first attached and continues at the meta-level over all the handlers in the sequence.

In the first case, the program flow depends on the order of the method invocations triggering the reification processes within the application program code. In the second case, the program flow depends on the order in the binding sequence. The reification process is triggered by one method and the flow is completely controlled by the meta-level.

To explain the second case, consider the combination of a generic tracing handler (`TracingHandler`) with a bean active property handler (`ActiveProperty-Handler`). The tracing handler intercepts all method invocations of any object being attached to it such that it gives method related information before and after the method is called. The active property handler transforms base objects' attributes into active JavaBeans properties characterized by the firing of an event when the state of the object changes. Further details on implementing generic tracing and active properties transformation code are presented in Section 3.4. The chaining relationship at the meta-level is, in general, not *commutative*, i.e., the relationship [`TracingHandler`, `ActivePropertyHandler`] has different semantics when compared to its reverse and depends on the handler implementations. Intercepting a set method by an `ActivePropertyHandler` involves calling get methods before and after invoking the method itself. As a result, with the first case, only explicit set method calls at the base level will be traced and the listener reaction will be accounted for. In the second case, however, in addition, the get method calls within the `ActivePropertyHandler` implementation will be traced.

3.3.4 Comparison with CLOS MOPs

In CLOS, customising the default behaviour of the underlying language as a mechanism for modifying application behaviour is realised in three steps. These steps constitute a procedure for extending the language by building a MOP interface [68]:

1. Use subclassing (or inheritance) to derive *specialised meta-object classes* from *standard meta-object classes*, for example:

   ```
   (defclass derived-class(standard-class) ...)
   ```

2. Modify *standard methods* which are generic functions applied to standard meta-object classes (and which define the language default behaviour) by defining corresponding methods specialised to meta-object classes defined in the first step. The *specialised methods* constitute behavioural customisation and can be added using method qualifiers.

3. Create *specialised class meta-objects* by instantiating specialised meta-object classes defined in the first step. The specialised class meta-objects inherit the behaviour of the standard class `standard-class` through inheritance but they also show additional behaviour as defined in the second step. Instances of the specialised class meta-object exhibit the customised behaviour when the modified methods are called on them.

Our approach, based on the reflection model proposed in the previous section, does not modify the underlying language. Instead, it provides (generic) implementations for customising application behaviour in a form similar to library routines. In comparison with the MOP approach of CLOS, we note that:

1. We use inheritance and derive classes from `java.lang.reflect.InvocationHandler` for implementing behaviour customisation. Objects of these classes are termed meta-objects.

2. Standard language methods are not modified. Bahaviour customisation is implemented in the `invoke()` method of `InvocationHandler`.

3. We use the proxy creation method to assign application objects to meta-objects. Creating a proxy represents a reification of customising the behaviour of the base object being assigned. By invoking an object's methods on the proxy, customisations of object behaviour are reflected back to the base level.

Kiczales et al. [68] label all instances of the specialised class meta-objects representing program domain as objects whereas all other entities are labelled as meta-objects. Meta-objects are distributed over a multilevel hierarchy with `t` class at the root, followed by `standard-object`. The next meta-level hosts the *standard meta-object classes* and consists of `standard-class`, `standard--generic-function` and `standard-method`. User defined meta-objects are termed *specialised meta-object classes* and their instances are called *class meta-objects*. Instances of the latter provide representation of the program domain and are termed objects.

Throughout this thesis, we retain the terminology used in popular OO languages where classes are strictly compile-time entities providing template behaviour of their instances, which are created at runtime. In OOP, classes can be divided into system classes (class hierarchy and runtime support) and user defined classes. Unlike Kiczales et al., we do not associate any meta attributes with the root class `java.lang.Object`; although the class is part of the *program domain* and not of the application domain. The same applies to classes of the Java package libraries or other user defined classes that may be used to represent the program domain.

We need, however, to extend the popular OO terminology used in Java and introduce the concept of meta-object. In the previous section, we have seen that Java's runtime system supports the concept of meta-objects. In Java's structural reflection model, meta-objects are runtime entities of type `java.lang.Class` which can be reified and used to expose the structure of application objects. Our model adds another type of meta-object which can be used to customise the behaviour of application objects. These meta-objects are runtime entities of type `java.lang.reflect.InvocationHandler` and can be reified using dynamic proxies. Instance of classes representing the problem domain are termed base objects.

3.4 Open Implementation

In computer science, the principle of abstraction plays a fundamental role in overcoming the complexity inherent in large software systems. Abstraction is used as a means by which uninteresting details can be removed while keeping the desired essence. The traditional way of applying abstraction follows the principle of information hiding according to which implementation details are separated from and hidden behind a module's interface. The interface represents an abstraction barrier separating clients from implementation. Data provided at the abstraction barrier is the only means by which clients can manipulate the abstraction; this strategy of information hiding is also known as black-box abstraction [74, 85].

The concept of information hiding was first introduced by Parnas [85]. He argued that in implementing a system, difficult design decisions or decisions which are likely to change subsequently should be assigned to modules and not to subroutines. Design decisions transcend execution time and therefore design modules need not correspond to processing steps. Each module should be designed so as to hide the internal details of its processing activities and modules communicate only through well-defined interfaces. According to Meyer [80], information hiding emphasizes separation of a module's function from its implementation. Meyer defines information hiding as a selected subset of the module's properties that can be made available to the authors of client modules. A principle related to information hiding is that of encapsulation, which is part of the OO paradigm and supported by OO languages. Encapsulation is sometimes used as a synonym for information hiding, but the two are different. Encapsulation is closely related to the concept of Abstract Data Type (ADT) which bundles data with methods that operate on that data. Often data abstraction as implemented in ADT is wrongly interpreted to mean that data is somehow hidden. Encapsulating data does not necessarily imply hiding data, for we can, for example, define a class where data members are directly accessible.

A module is conceived as a programming unit which handles one aspect of the problem being solved. In OO, a module is identified more closely with a class [58]. The principle of information hiding is not restricted to OO class abstraction, however. Prior to the emergence of popular OO programming languages, Niklaus Wirth developed Modula-2 [103] as a language where separation between interface and implementation is supported explicitly. In Modula-2, implementation details are hidden in an "Implementation Module" behind an interface defined as "Definition Module". In Ada [2], the programming units used to hide implementation details are the "Packages".

Despite the benefits of applying black-box abstractions such portability and localisation of change, there are cases where it is advantageous to open implementations and allow clients to change them. The open implementation approach [65, 66] calls for a new and practical interpretation of the black-box abstraction allowing clients some control over the selection of the implementation strategy while still hiding many true implementation details. That is, in cases where default behaviour is inappropriate, clients can customise an implementation according to their needs without changing its default behaviour. An open architecture is a two-level reflective architecture in which the default implementation resides at the base level whereas the meta-level is optional and can be used if the default implementation is inadequate to meet client's requirements.

Our reflection model supports the principle of open implementation. The goal of open implementation is to give clients some control over the implementation strategy of the modules while retaining the advantages of the traditional closed implementation modules [69]. Our goal is to allow clients to modify the behaviour of their applications by providing them with reusable code and a meta-interface to access this code.

3.4.1 Open implementation guidelines

Kiczales et al. [69] proposed five design guidelines for implementing open architectures. In the terminology of the authors, "a *module* represents a work assignment and an *interface* is the set of assumptions a client programmer using the module may make about its behavior". The guidelines are:

1. Open implementation module interfaces should support a clear separation between client code that uses the module's functionality (use code) and client code that controls the module's implementation strategy (ISC code)
2. Open implementation module interfaces should be designed to make the ISC code optional, make the ISC code easy to disable, and support alternative ISC codes for one piece of use code.
3. Open implementation module interfaces should be designed to allow the scope of influence of ISC code to be controlled in a way that is both natural and sufficiently fine-grained.
4. Open implementation module interfaces should be designed to pass only essential implementation strategy information.
5. When there is a simple interface that can describe strategies that will satisfy a significant fraction of clients, but it is impractical to accommodate all important strategies in that interface, then the interfaces should be layered.

To analyse our reflection model according to the guidelines just presented, we need to identify some expressions in the list with the terms used in this thesis. An open implementation module in the work of Kiczales et al. corresponds to a programming entity implementing our behavioural reflection model. The implementation strategy (ISC code) corresponds to the code used to reify meta-objects by creating proxies and to assign meta-objects their referents. Moreover, the "use code" refered to in the list corresponds to the meta-code implementing behaviour customisation. Default implementation is provided by the application (at the base level).

In reflective systems implementing our model, application code used to reify meta-objects and trigger the meta-level is separated from customisation code implemented at the meta-level. Thus the first guideline is satisfied.

The meta-code can be activated according to a certain procedure of assigning application objects to meta-objects and invoking application methods on the created proxy representing the first objects. Since this procedure is optional and default implementation should work without the meta-program, the second guideline is satisfied.

We assume that the meta-program is implemented in a generic manner independently of potential client applications. In this case, the scope of influence is purely dynamic and affects only the behaviour of base objects assigned to meta-objects at runtime. Thus the third guideline is satisfied.

In discussing guidelines 4 and 5, Kiczales et al. introduce the concept of *ISC code subject matter* to express explicitly what the code is about. Recall that "ISC code" corresponds to the code controlling the invocation of the meta-program. The subjects that can be handled by this program depend on the reflective model and its expressive power. The model is limited to behavioural reflection; it does not, however, restrict the subjects to be handled. The appropriateness of implementing the model depends on the requirements of potential client application.

It is possible to provide client applications with a set of alternatives corresponding to different subjects. To activate the meta-program, base objects are passed to the meta-level together with additional parameters potentially needed for execution of the desired behaviour. In this way, guidelines 4 and 5 are satisfied.

Therefore, the reflection model proposed in Section 3.3 provides means for realising the open implementation design principle according to the guidelines of Kiczales et al.

3.4.2 Customising applications' behaviour

In our reflection model, base applications provide default behaviour and remain executable without being bound to the meta-interface. Base applications correspond to the default implementation of the clients' modules. The meta-interface provides access to customise the default implementation. If the meta-program is implemented in a generic manner, we can use the meta-code to customise *arbitrary* applications without changing their default behaviour.

In the following examples, we note that the base level application remains executable without being attached to the meta-level. The meta-level program is optional and is supposed to customise particular aspects of the application to better meet new requirements.

We consider two examples showing how the proposed behavioural reflective model supports open implementation for applications running on a single JVM.

3.4.2.1 Tracing

Consider a tracing meta-object, which intercepts all method invocations of the application objects it represents such that method related information before and after the method is called is accessible. The tracing code implemented at the meta-level using the proxies can be made available to any application as part of a debugging and testing framework, without the need to change the application code. The tracing strategy can be modified without affecting potential client applications; applications can use the code without the need to change the behaviour or the structure of any of their objects.

```java
import java.lang.reflect.*;                                                //1

public class ProxyFactory {
  public Object createProxy(Object base_obj) throws Throwable {
    Class c=base_obj.getClass();
    ClassLoader cl=c.getClassLoader();
    Class[] infs=c.getInterfaces();
    Object proxy=Proxy.newProxyInstance(cl, infs, new TracingHandler(base_obj));
    return proxy;
  }
}                                                                          //11
```

Listing 3.1: ProxyFactory for creating proxy objects

Creating a proxy constitutes the first step in the process of customising application behaviour. Listing 3.1 shows the class `ProxyFactory` responsible for creating proxies. The base object is passed as a parameter to the invocation handler (here `TracingHandler`). The returned proxy object acquires the types (interfaces) of the base-object. Invoking base methods on the proxy triggers the reification process in which the behaviour of the base-object can be modified. The reified meta-object of type `TracingHandler` inspects the structure of the base object and uses the collected data to pass debugging information about its referent.

We note that `ProxyFactory` does not represent a creational design pattern as the Factory patterns of [47]. With `createProxy(Object base_obj)`, base objects can register themselves and be attached to meta-objects at the higher level. The returned proxy object can be used to represent base objects. `ProxyFactory` depends on the type of the invocation handler and the proxy classes of `java.lang.reflect` package but is independent of application. The same applies to the invocation handler class implementing the customisation code.

```
// base level object representing a Model
IPoint apoint=new Point();
// base level listener representing a View
Demo listener= new Demo();
// binding base objects to a proxy invocation handler
ProxyFactory fac=new ProxyFactory();
IPoint wrap=(IPoint)fac.createProxy(apoint, listener);
// triggering reification by setting x
wrap.setx(<arbitrary values>);
```

Listing 3.2: Example Runtime binding and triggering reification

3.4.2.2 Active JavaBeans Properties

Consider the problem of transforming the attributes of some objects into active JavaBeans properties. Active JavaBeans properties are characterized by the firing of an event when the state of the object changes; for example, by invoking set methods. We can make a simple property active by binding its changes to anonymous event listeners. JavaBeans support active properties through a new event type called the `PropertyChangeEvent`, the listener interface `Property-ChangeListener` and through a support class called `PropertyChangeSupport`. The JavaBeans mechanism of active properties can be used for implementing the Model-View-Controller (MVC) pattern often employed in developing GUIs [71]. The View object is supposed to be informed every time the state(s) of its Model object(s) is (are) changed. Views are registered as listeners to changes of active properties in corresponding Models. The firing mechanism for an event which changes a model's state and thereby informs interested listeners can be implemented using dynamic proxies. The proxy implementation has several advantages when compared with the classical approach (using inheritance and aggregation) of realising the MVC pattern for constructing GUIs, namely:

1. the separation of event-firing and informing listeners from basic classes (Views and Models),
2. the provision of a generic implementation of the event-triggering mechanism.

Using inheritance and aggregation for implementing the MVC pattern requires adding event-firing code in all the set methods of all Model classes holding the data. This implies dependency of the GUI implementation on the corresponding JavaBeans classes, stronger coupling and potentially extra costs of testing and maintenance. Listing 3.2 shows how a `Point` object with type `IPoint` (representing a Model) can be attached to a meta-object of type `ActivePropertyHandler` (see Listing 3.4). A listener object of an arbitrary type `Demo` (representing a View) is assigned. The implementation of the binding method `createProxy()`

follows the same pattern of Listing 3.1 with the meta-object type being `ActivePropertyHandler`.

In addition to being an `InvocationHandler`, `ActivePropertyHandler` also implements the interface `IPropertyChange` (Listing 3.3) to allow for the addition and removal of listeners and for firing a property change event after intercepting the set methods.

```
import java.beans.PropertyChangeListener;

public interface IPropertyChange extends Serializable {
   public void add(PropertyChangeListener lis);
   public void remove(PropertyChangeListener lis);
   public void fire(String property, Object oldVal, Object newVal);
}
```

Listing 3.3: Example Runtime binding and triggering reification

Intercepting any change of state of a base level object of any type is implemented in Listing 3.4 as follows. Before invoking the set method on the base-object, the meta-objects gets the old value of the base object's property field. It then invokes the set method, gets the new value and fires the event. Any other method is delegated to the base-object. We note that the base level object attached to `ActivePropertyHandler` is of type `java.lang.Object` and that we use polymorphism and reflection to provide a generic implementation of the JavaBeans active properties' mechanism. `ProxyFactory` and `ActivePropertyHandler` are implemented independently of base objects and can thus be used as library classes providing support for implementing GUIs of arbitrary applications. Java's dynamic proxies mechanism provides the means of invoking the reflective code at runtime.

3.5 Discussion

A proxy is a program that provides a communication bridge allowing different applications (clients and targets) to engage and exchange data. Dynamic proxies are proxies with additional mechanisms supporting behavioural reflection and open implementation. There are advantages to using dynamic proxies instead of normal proxies, particularly in relation to program design and consequently program maintenance. Firstly, from an OO Design (OOD) point of view, dynamic proxy classes were shown earlier in this chapter to exhibit less coupling to target classes in comparison with normal proxies. There is no direct relationship between these two class groups. The coupling takes place dynamically at the object level. In fact, the type of a dynamic proxy is set to that of the target object upon its creation at runtime. The type of a normal proxy, on the other hand, is set at compile-time. Secondly, the dispatching mechanism of

```
import java.lang.reflect.*;
import java.beans.*;

public class ActivePropertyHandler implements InvocationHandler, IPropertyChange {
   private Object base_obj;
   private Object listener;
   public ActivePropertyHandler(Object base_obj, Object lis) {
       this.base_obj=base_obj;
       this.listener=lis;
       (this).add((PropertyChangeListener)listener);
  }
  public Object invoke(Object proxy, Method meth, Object[] args) throws Throwable {
    Object result = null;
    if (meth.getName().startsWith("set")!=-1) {
       String property=meth.getName().substring("set".length());
       Object oldVal=null; Object newVal=null;
       Method[] meths=base_obj.getClass().getDeclaredMethods();
       Method getMethod=null; // get-Method thru introspection
       for (int i=0; i<meths.length; i++) {
          // obtain oldVal through getMethod;
       }
       try {
           result=meth.invoke(delegate, args);
           newVal=getMethod.invoke(delegate, null);
       } catch (InvocationTargetException e) { ... }
       (this).firePropertyChange(property, oldVal, newVal);
    }
    else {
       //no set-method, dispatch control to base object
    }
    return result;
  }
  // use a PropertyChangeSupport to manipulate listeners
  PropertyChangeSupportsupport pcs=new PropertyChangeSupport(this);
  public void add(PropertyChangeListener lis) {pcs.addPropertyChangeListener(lis);}
  public void remove(PropertyChangeListener lis) {
    pcs.removePropertyChangeListener(lis);
  }
  public void fire(String property, Object oldVal, Object newVal) {
    pcs.firePropertyChange(property, oldVal, newVal);
  }
}
```

Listing 3.4: Generic implementation of JavaBeans mechanism

dynamic proxies allows, using polymorphism and reflection, for implementing generic behaviour independent of the target's type.

The flexibility gained by employing the behavioural reflection model is not without cost. The extra overhead imposed by reflection might cause performance problems. Moreover, since the proposed reflection model allows each application object to acquire an arbitrary number of meta-objects, the number of meta-objects is virtually infinite. It is therefore necessary to manage the meta-objects in such a way that they can be cached, shared and created only when needed, i.e., in a *lazy* way. The problem of setting a caching policy including sharing and lazy creation to improve performance is left for future research.

The proposed reflection model based on dynamic proxies has parallels with the AOP approach [3, 70], since it inherently exhibits a degree of separation of concerns. In AOP, a systems constitutes a collection of concerns which are classified as core concerns and system-level concerns called aspects. The first group deals with functional requirements and can be implemented as loosely coupled programming units using techniques and constructs supported by current programming languages (including OO languages). The latter tend to *cross-cut* core programming units. However, separation of concerns in AOP does not necessarily mean weak (or loose) coupling; there is nothing that prevents aspects implemented in AspectJ (the most widely used language) [4] from explicitly referring to classes of the core concerns. In addition, core concerns and aspects are combined at compile-time. Nevertheless, AspectJ allows structural change of application classes and provides a powerful join-points model which allows grouping of pointcuts. Pointcuts are execution points at which aspect code is inserted (weaved) into the application at compile-time.

Inheritance is recognized as an object-oriented mechanism that facilitates software reuse. Empirical studies have shown that, as OO systems develop in time, inheritance trees remain relatively shallow indicating that software developers are aware of the conflict between the advantages and disadvantages of inheritance [17]. Advantages of inheritance are mainly reuse and disadvantages are increased coupling and complexity. In [31], Cargill recommends that developers should reduce inheritance and hence reduce coupling. Our reflection model can be employed to implement reusable Java code avoiding the problems of strong coupling associated with inheritance and aggregation.

OOD is concerned with developing a model mapping the problem domain into classes and objects to implement the system requirements. Design activities and artefacts serve to document system structures and to provide a smooth transition in the implementation phase. There are no prepared recipes for building high quality software which is reusable, extendable, adaptable and reliable; but there are design principles, guidelines and concepts. Applying these principles and making use of the concepts enhance the quality of software. For example, following the idea of Design by Contract [79] enhances the reliability of the system. From the perspective of viewing the relationship between a class and its clients as a formal agreement, the correctness of software elements can be addressed [80]. Also, applying the design principle of open implementation makes software systems more adaptive to changes. The main features of open architectures are less coupling between components/objects and separation of concerns.

3.6 Conclusions

In this chapter, we have provided an analysis of the reflective abilities of the Java language. The proposed reflection model established with the help of proxies provides a means of customising applications' behaviour and of adapting to new requirements without the need for modification. The reflection model supports application customisation in a way similar to the open implementation approach. Application objects provide a default representation, which acts upon and deals with some part of the real world (problem domain). The base representation is closed, meaning that it remains executable without being merged with the meta-representation. The latter representation is independent of applications; it is defined solely in terms of the service it provides.

The proposed behavioural reflection model alters neither the underlying object model of the Java language nor the JVM. In the context of OO programming languages, an object model refers to a collection of concepts used to describe objects in the language. In the Java object model, classes represent higher abstractions of application domains and objects are created as class instances at runtime. As with the object model of most OO languages, Java's object model is said to be a *2-level* system, where classes are not first-class objects, i.e., not instances of higher entities often referred to as meta-classes, and classes cannot be created by constructors and cannot receive messages. This is in contrast to *3-level* systems where classes are instances of meta-classes and can be manipulated at runtime, for example, CLOS. `java.lang.Object` continues to be the root of the inheritance hierarchy of any application and the mechanism of associating instances of `java.lang.Class` with application objects upon loading corresponding classes remains the same. `java.lang.Class` and classes of `java.lang.reflect` package represent an API allowing introspective structural reflection.

Java's dynamic proxies provide the means for writing meta-programs enabling the interception of base level operations such as method invocations, field access and the transfer of control to the proxy object at the meta-level. They allow the meta-level to replace operations in the base level and support dynamic binding of proxy objects to the base level. Proxies have added new features to the reflective power of the Java language beyond introspection. However, not all proxy technologies support meta-programming; Microsoft Component Object Model (COM) proxies, for example, do not allow reflective techniques.

Although dynamic proxies are mainly used to implement J2EE architectures, where application servers are set as part of an enterprise solution, we use proxies as a means of writing reusable code in an open architecture. Our approach does not have any restrictions on the architecture and can be extended to a distributed environment. That is, the base level and the meta-level need not be part of the same environment and may be made to communicate over the Remote Method Invocation (RMI) protocol, the Internet Inter-ORB Protocol (IIOP) of the Common Object Request Broker Architecture (CORBA), the HyperText Transfer Protocol (HTTP) or a combination of these protocols.

Dynamic proxies support open implementation. They are implemented in a generic manner and are thus made reusable allowing customisation of application behaviour after being coupled to the application through well-defined factory methods. A base level object may be attached to several proxies, each of which controls one aspect of its behavior.

We do not address the impact of customizing system behaviour at runtime on system correctness and leave this question for future investigation. We also do not discuss the option of considering the Java language as an application domain and customise its library classes instead of customising Java applications. We leave this option for future research.

Chapter 4

Applications of the Reflection Model

In Chapter 3, we proposed a behavioural reflection model with the aim of constructing open architectures showing less coupling and a higher degree of reuse. The model was applied to non-distributed applications. The purpose of this chapter is to demonstrate the applicability of the reflection model in distributed environments and its impact on design patterns. By showing the applicability of the model, we will proceed in the next chapters to investigate the coupling and reuse properties of systems employing the model using techniques of measurement theory.

In this chapter, we extend the model to include distributed systems using different networking technologies provided by the Java 2 platform and discuss the prospect of implementing reusable code on the server side. Reusable components, represented as classes or logically connected groups of classes, are implemented independently of, and separated from, their application context. As such, coupling between reusable components and applications is reduced and as a result the level of reusability is increased. The composition of precompiled reusable components with application objects, implemented at the base level on the client side, is realized at runtime by passing application objects as parameters. Decoupling is achieved by disregarding the meta-object to which base objects are attached and by invoking the methods directly on base objects. Parameter passing varies according to the distributed technology used. In all cases, however, the objects must be *serializable* to conform to Java's mechanism of sending object data between different virtual machines running on different computers on the network. In RMI, the objects are passed as parameters of a method invoked on a remote object. In CORBA, the objects are passed over the Object Request Broker (ORB) as parameters of a service method invoked on the server object implementing the method. With Java's Servlets, the base objects are sent as parameters of an HTTP-request method and the client receives the result as an HTTP-response. On the server side, Java's Servlets API is used to implement specialized Servlets controlling access to the reusable code.

In this chapter, we also address the question of adapting design patterns of Gamma et al. [47] to the reflection model supported by dynamic proxies. Although many patterns make use of inheritance, Gamma et al. propose programming to interfaces and favour object composition over class inheritance as guiding principles for pattern implementation. The proposed reflection model provides a mechanism based on flexible runtime object composition. As a result, applying the reflection model is expected to lessen coupling and increase code reuse of (at least some) pattern implementations.

The chapter is organised as follows. Section 4.1 extends the reflection model to distributed applications. In Section 4.2, different networking technologies are considered including RMI, CORBA and Java Servlets. First, we consider the RMI protocol and use the corresponding technology (Java's RMI API) available as part of the evolving Java development environment since JDK1.1.x. Next, we look at a CORBA implementation using a fully-functional ORB that is available in every deployment of the Java 2 Platform since JDK1.2.x. Finally, Section 4.2.3 deals with the HTTP protocol and builds a Java Servlet implementation using the corresponding API available as part of Java 2 Platform, Enterprise Edition (J2EE). In our discussion of the different implementations, we omit issues related to network security, access control, data encryption or Internet-firewalls, which we consider outside the scope of our research[1]. In Section 4.3, the question of adapting design patterns to the proposed reflection model is addressed. In Section 4.4, we discuss the different reflective implementation approaches for constructing distributed systems and assess their merits. The impact of applying our reflection model on design patterns in terms of less coupling and a higher degree of code reuse is presented. Section 4.5 ends the chapter with some concluding remarks.

4.1 Open architecture in a distributed system

The model of Figure 3.4 in Section 3.3.2 provides a suitable OO framework to implement code separated from, and independent of, client applications. It also allows the composition of components with application code at runtime in a natural way. From a distributed system perspective, however, it represents a single application running on a single JVM. The proxy code must be downloaded and installed on the client's machine (i.e., where the application code runs) to permit the construction of an open architecture and consequently to allow the customization of the default implementation.

In a distributed environment, the code must be accessible by several clients

[1]Although network security and related issues are important features of distributed systems, they are orthogonal to software attributes such as coupling and code reuse as well as to design aspects such as open implementation.

running on different machines, each with its own JVM. In such a case, the link between base and meta-level must be modified in a way which allows several applications at the base level to customize the behaviour of their objects concurrently using meta-objects running on a different machine. The resulting client/server architecture requires the specification of a communication protocol and consequently the application of the technology that allows the implementation of such a protocol. There are several alternatives, the most common of which are the Java Remote Method Protocol (JRMP), better known as RMI protocol [8], the Internet Inter Object request broker Protocol (IIOP) associated with the CORBA [1, 5] architecture and the HyperText Transfer Protocol (HTTP) on the Web [6, 7].

The most natural extension of the open architecture of Figure 3.4 is a client-/server architecture with several clients at the base level connected to one (or more) server application(s) at the meta-level over a TCP/IP network (see Figure 4.1).

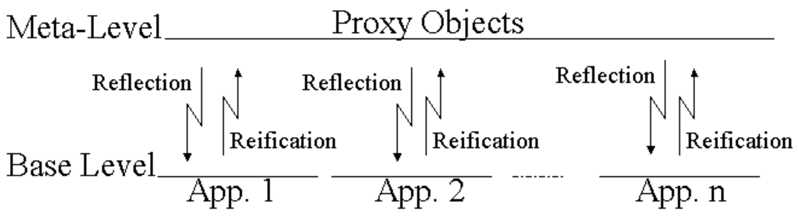

Figure 4.1: A two-level reflective model with remote proxies

Using JRMP as a communication protocol, the communication between clients denoted by App. 1, App. 2, ... , App. n, at the base level and the server application(s) at the meta-level, realized on top of the TCP/IP protocol, extends over the three layers of the RMI architecture. The RMI system provides the client with a stub (a proxy) representing the remote object to which the application object can be bound to customize its behaviour. The remote method call on the stub is forwarded to a *RemoteRef* object residing in the remote reference layer; this layer defines and supports the invocation semantics of RMI. The connection between the JVM of the client and that of the server is realised at the transport layer with the support of the TCP/IP protocol.

With the IIOP protocol, the communication between client and server is realized in a similar fashion. Each object reference on the client side, App. 1, App. 2, ... , App. n, points to a stub function. The stub uses the ORB to connect to the server machine that runs the server object. After establishing the connection, the stub sends the object reference and parameters to the skeleton code linked to the destination object's implementation. The skeleton code transforms the call and parameters into the required implementation-specific format and calls the object. Any results or exceptions are returned along the same path via the ORB. The ORB represents the underlying transport layer.

Servlets follow a request/response model in which a client sends an HTTP request message to a server and the server responds by sending back a reply message. Servlets are normally used to support application processing assigned to a middle tier that acts as a Web-server in so called three-tier client/server systems, passing and controlling the requests and responses between a light-weight client and a data source. The clients, App. 1, App. 2, ... , App. n pass their requests to the servlets on the server side utilizing the HTTP protocol and receive, in response, a reference to remote objects.

We note that by distributing the application over several machines and assigning the meta-level its own JVMs on which the server process(es) runs, we do not modify the semantics of either the meta-level or the base level objects. That is, in the distributed architecture, the model of reflection provided by Java's dynamic proxies persists. The proxies' invocation handler are still interpreted as meta-objects, to which application objects at the base level, distributed over different clients' processes and memory environments, can be attached to customize their behaviour. Moreover, the causal relationship between base and meta-level remains the same. Distribution adds a new dimension to the existing structure without affecting the other dimension representing the base-meta property of objects. The reification process and its triggering by method calls at the base level remains the same and with RMI, CORBA and Java's servlets technologies, the details of how this process develops over the network remain hidden from the user. The question we need to address is, what modifications are required in order to attach an application client object to a remote reusable meta-object running on a different machine?

4.2 Distributed Applications

We now address the question of constructing a distributed system by implementing the model of Figure 4.1. The resulting system supports code reuse using the networking technologies available in the Java-2 Platform. We start with the RMI, followed by CORBA and finish with the servlet implementation of our model. We deal with each of the networking approaches separately and in each case we consider clients as simple applications that communicate with their respective servers over the corresponding protocol. With the support of the networking technologies provided the by Java 2 Platform, it is possible to distribute the objects of Figure 4.1 over different JVMs by further dividing them into client and server objects. Base objects with their interfaces represent client applications, whereas proxies with associated handlers and factories define the server application.

4.2.1 RMI-based system

To begin with, we note that in the single application case, `ProxyFactory` (Listing 3.1) or its representative plays the role of interface between clients and the customisation service provided by the meta-objects. With this observation, we propose to employ the following steps for transforming a single application system into a distributed RMI-based system:

1. Define a remote service interface with a creation method `createProxy()`.
2. Adapt the code connecting the base and the meta-level by defining a wrapper class around `ProxyFactory`. This wrapper is a remote object that implements the remote interface of step 1 and delegates the binding of base objects to its aggregate, the `ProxyFactory`.
3. Make base objects used as parameters or as return values of a remote method call serializable.
4. Define an HTTP service class to allow dynamic class loading using `RMI-ClassLoader`.

For the first step, defining a remote service interface is part of any RMI implementation in order that the client can communicate its queries to the server through the layers of the RMI system architecture. In our case, the remote interface offers a method that allows client objects at the base level to bind themselves to meta-objects on the server side. Listing 4.1 shows a pseudo implementation of `IProxyFactory`, where the first `Object` parameter refers to the client base object to be assigned to a meta-object and consecutive dots mean possible extensions, depending on the application. Remote service interfaces such as `IProxyFactory` must be visible to both client and server at compile time.

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface IProxyFactory extends Remote {
   public Object createProxy(Object o) throws RemoteException;
   public Object createProxy(Object o, ...) throws RemoteException;
   ...
}
```

Listing 4.1: The communication interface between clients and server(s)

For the second step, Listing 4.2 depicts a pseudo implementation of `ProxyFactoryWrapper`. It implements the remote interface `IProxyFactory` and thus defines a remote object. It also extends `UnicastRemoteObject` allowing the creation of a simple remote object that supports unicast (point-to-point) remote communication and uses RMI's default socket-based transport for communication. Although this is not the only way for clients to connect to a remote service implementation, it is compatible with clients' JVMs that run on JDK1.1.x. Notice that `ProxyFactoryWrapper` uses `ProxyFactory` to create the proxy and attaches an application object to it.

```
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;

public class ProxyFactoryWrapper extends UnicastRemoteObject
                                         implements IProxyFactory {
  private ProxyFactory bpf=new ProxyFactory();
  public Object createProxy(Object obj) throws RemoteException {
     // delegate creation of a proxy to bpf and return the object
  }
  ...
  public ProxyFactoryWrapper() throws RemoteException {super();}
}
```

Listing 4.2: Delegating binding of a base object to ProxyFacory

For the third step, client objects to be attached to meta-objects on the server side are passed as parameters through `ProxyFactoryWrapper` methods. These and all other objects that need to be exchanged between client and server as parameters or as return types must be serialized, i.e., implement the `java.io.Serializable` interface. Serialization is necessary to conform to the RMI mechanism of sending object data over the network wire. The objects are then de-serialized in the remote JVM and made ready for use by the application that needs them.

For the fourth and final step, RMI supports remote dynamic class loading through the `RMIClassLoader`. The property `-Djava.rmi.server.codebase=-<URL-value>` can be used to specify a Uniform Resource Locator (URL) value identifying the resources on the network. This allows the RMI system to dynamically load the required classes. We may specify the path by setting `-Djava.-rmi.server.codebase=file://<class path>` or an HTTP (or FTP) server by setting `-Djava.rmi.server.codebase=http://<resource name>`. In the latter case, active servers on the client as well as on the server side are necessary to aid the delivery of class files between JVMs on both sides. If a serialized object is passed from one JVM to another, the receiving machine needs to load the class file of that object. In passing the object, the RMI system embeds a URL specifying where on the network this class file can be found and asks the HTTP server to deliver the file on request.

The class diagram of Figure 4.2 shows the key classes needed to implement the distributed system, according to the transformation algorithm described previously. Notice the remote layer consisting of `ProxyFactoryWrapper`, `IProxyFactory`, `java.rmi.server.UnicastRemoteObject` and `java.rmi.Remote` are inserted between the meta-level and the base level classes of Figure 3.3.

To explain how to implement the algorithm described by steps one to four, we consider the single application system example on "Active JavaBeans Properties" discussed in Section 3.4.2 of the previous chapter. We concentrate on the features related to transforming this application into a distributed RMI-based system. In the distributed system, the reusable code becomes part of the server implementation and consists of three units. The first is the interface `IPropertyChangeSupport`, to allow the registering and removal of listener objects as well as firing a `PropertyChangeEvent` to notify the listeners once the object's state has been modified. The second is `ActivePropertyHandler`, to provide the reusable code by implementing the interfaces `InvocationHandler` and `IPropertyChangeSupport`. The third is the `ProxyFactory`, to deliver a reference to a proxy object.

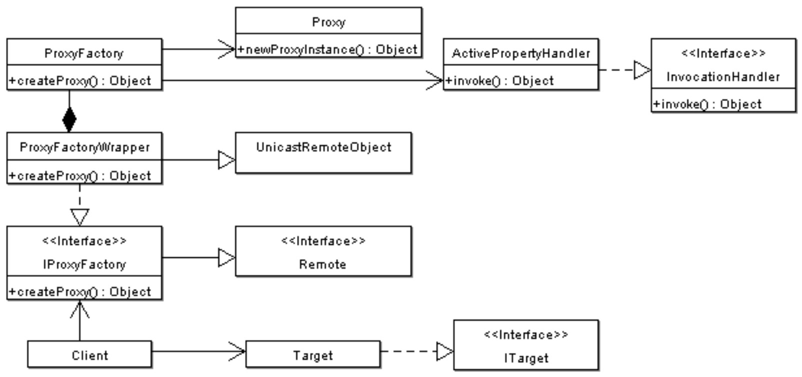

Figure 4.2: Class diagram in the distributed RMI environment

Distributing the system requires a server process on the server-side. Listing 4.3 shows the class `ProxyServiceRegistration` with the `main()` method to start the server. A security manager to control access to the resources from downloaded code to run within the server virtual machine is installed (line 6). The security policy can be set using the `-Djava.security.policy property`. Next, `ProxyFactoryWrapper` (See Listing 4.5) is instantiated and bound to an arbitrary name (lines 9 and 10). The client process looks up the remote object by using this name. By registering the service, the server process is completed and waits for client requests running on different JVMs on the network.

We now turn to the communication interface and define, as required in the first step, the remote interface as in Listing 4.4. This interface must be known to both the client and server.

As required in the second step of the transformation algorithm, we employ the Adapter pattern of Gamma et al. [47] and wrap the `ProxyFactory` in a

```
import java.rmi.*;                                                      //1
import java.util.*;                                                     //2
                                                                        //3
public class ProxyServiceRegistration {                                 //4
   public static void main(String[] args) {                             //5
       System.setSecurityManager(new RMISecurityManager());             //6
       try {                                                            //7
           System.out.println("Registering ProxyService ...");          //8
           IProxyFactory bps = new ProxyFactoryWrapper();               //9
           Naming.rebind("ProxyService", bps);                          //10
           System.out.println("Server is  ready.");                     //11
       } catch (Exception e) { ... }                                    //12
   }                                                                    //13
}                                                                       //14
```

Listing 4.3: The RMI-server process

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface IProxyFactory extends Remote {
   public Object createProxy(Object o, Object lis) throws RemoteException;
}
```

Listing 4.4: The remote interface defining the service protocol

remote object `ProxyFactoryWrapper` as in Listing 4.5.

Having established the generic service on the server side and a generic communication interface, all of which reside in the meta-level, we now turn to the client side and consider a simple application at the base level involving a `Point` object and a client object as listener. The attributes of the `Point` object are supposed to be customized to become active JavaBeans properties. The `Point` class implements a serializable interface `IPoint`, which defines get and set methods for the attributes. Listing 4.6 shows an example of a client listener class (implements `PropertyChangeListener`, lines 8 - 10). The client request begins by installing a security manager (line 13). The security manager is necessary, because, as with the server, the RMI-system could be downloading code to the client. In this example, the `ProxyFactoryWrapper`'s stub is downloaded. After installing a security manager, the base object is created (line 14) and the client constructs a name used to look up an `IProxyFactory` remote object (lines 16 and 17). The value of the first command line argument, `args[0]`, is the name of the remote host on which the `IProxyFactory` object runs. `args[1]` specifies the port that the remote object uses to accept requests. The client uses the `Naming.lookup()` method to look up the remote object by name in the remote host's registry (line 18). With the help of the remote object, the proxy object is created (line 20). In creating the proxy, the base object is attached by passing

```
import java.lang.reflect.*;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;

public class ProxyFactoryWrapper extends UnicastRemoteObject
                                                implements IProxyFactory {
   private ProxyFactory bpf=new ProxyFactory();
   public Object createProxy(Object o, Object lis) throws RemoteException {
       Object proxy=null;
       try {
           proxy = bpf.createProxy(o, listener);
       } catch(Exception e) { ... };
       return proxy;
   }
   public ProxyFactoryWrapper() throws RemoteException { super();}
}
```

Listing 4.5: Remote object wrapper for the Factory class

it as parameter, together with a listener object. Invoking a set method on the proxy causes the corresponding object attribute to acquire a new value and an event of type `PropertyChangeEvent` to be fired informing the listener of the state change. In this way, the attributes of the base object behave as active properties of JavaBeans.

As in the server case, the client defines a security policy and may use a Web-server to download the classes needed by the RMI-system on its side dynamically. The client may, however, provide a URL path expression to indicate where the classes will be made available. In our example, these classes are `Point`, `IPoint` and `ProxyClient`.

4.2.2 CORBA implementation

In CORBA, client and server applications interact over the ORB. Using the ORB, the client conveys a request for a method invocation to the server and the server sends results back to the client along the same path.

We follow a similar procedure to the RMI case where we define an *IDL*-interface and implement it on the server side using the functionality of `Proxy-Factory`. Moreover, we use the IDL programming model of the Java-2 Platform, known as Java IDL, consisting of both the Java CORBA ORB and the *idlj* compiler. The compiler maps the Object Management Group's (OMG) Interface Definition Language (IDL) to Java bindings [9]. The model enables distributed Web-enabled Java applications to transparently invoke operations on remote network services using the industry standard IDL and the IIOP protocol [1].

Following the IDL programming model, we define remote interfaces using the IDL, then compile the interfaces using the idlj compiler. The compiler generates the Java version of the interface, as well as the class code files for the

```
import java.beans.*;
import java.lang.reflect.Proxy;
import java.rmi.*;
import java.io.Serializable;

public class ProxyClient implements PropertyChangeListener, Serializable {

   public void propertyChange(PropertyChangeEvent e) {
       // implement the reaction of the listener
   }

   public static void main(String [] args) throws Throwable {
       System.setSecurityManager(new RMISecurityManager());
       IPoint point=new Point();
       try {
           String service="ProxyService";
           String url="rmi://"+args[0]+":"+args[1]+"/"+service;
           IProxyFactory bpf=(IProxyFactory)Naming.lookup(url);
           ProxyClient lis=new ProxyClient();
           IPoint wrap=(IPoint)bpf.createProxy(point, lis);
           // Do something with the proxy object thus obtained
       } catch (Exception e) { ... }
   }
}
```

Listing 4.6: A PropertyChangeListener as a client process class

stubs and skeletons that enable our applications to hook into the ORB. Applying idlj on the IDL interfaces is similar to invoking the *rmic* tool of RMI system on remote object implementations, i.e., `ProxyFactoryWrapper` in the previous section. Because the services provided by the server are reusable pieces of code, they are generic, implement general behaviour and are independent of applications' specifications. The IDL interfaces, which nominate the services implemented on the server side and are passed to clients over the ORB, reflect the genericity of these services by using generic parameters and return types unrelated to any client application. IDL provides appropriate types, e.g., *any*, that allow us to express the genericity of the services. An *any*-type is mapped into Java `org.omg.CORBA.Any` and an object of such type represents a pair consisting of an `org.omg.CORBA.TypeCode` and a value. We note that `org.omg.CORBA.Any` provides operations that allow insertion and extraction of the `TypeCode` and the value contained in the object.

Listing 4.7 shows an example of a generic IDL interface in pseudo-code, where consecutive dots denote further functions. The first in-parameter, `obj`, of type `any` represents the application object to be attached to the proxy invocation handler. The handler, implemented at the server side, provides reusable functionality suitable to customize the behaviour of the object. The second parameter is of type `any` and is added arbitrarily. The function's return value is also of type `any`. The module `ProxyObjects` is mapped into a Java package with the same name, where the stubs and skeletons needed for the communication over the ORB are placed.

```
module ProxyObjects {
  interface ProxyFactoryWrapper {
    any createProxy(in any obj, in any listener);
    ...
  };
};
```

Listing 4.7: IDL to support reuse of method invocations over the ORB

```
import ProxyObjects.*;                                                  //1
import org.omg.CORBA.*;                                                 //2
import java.lang.reflect.*;                                             //3
                                                                        //4
public class ProxyFactoryWrapperImpl extends ProxyFactoryWrapperPOA {   //5
   private ProxyFactory bpf;                                            //6
   public ProxyFactoryWrapperImpl (ProxyFactory bpf) {                  //7
     this.bpf=bpf;                                                      //8
   }                                                                    //9
                                                                        //10
   public Any createProxy(Any obj, Any lis) throws SystemException {    //11
     java.lang.Object oo=(java.lang.Object)obj.extract_Value();         //12
     java.lang.Object ol=(java.lang.Object)listener.extract_Value();    //13
     Any anyThing=_orb().create_any();                                  //14
       try {                                                            //15
           // ActivePropertyHandler defines general behaviour           //16
           ActivePropertyHandler aph=bpf.createProxy(oo, ol);           //17
           anyThing.insert_Value(aph);                                  //18
       } catch (Exception e) { ... }                                    //19
     return anyThing;                                                   //20
   }                                                                    //21
   // ...                                                               //22
}                                                                       //23
```

Listing 4.8: Implementation of the IDL interface ProxyFactoryWrapper

A Java sample implementation of the IDL interface `ProxyFactoryWrapper` is shown in Listing 4.8. The class extends the `ProxyFactoryWrapperPOA` (line 5) and provides a method for each operation in the interface. An object of type `ProxyFactory` is used as an aggregate (line 6) and initialized in the constructor (lines 7 - 9). In the method `createProxy()`, the objects' content (values) of the `org.omg.CORBA.Any` parameters are first extracted as `java.lang.Objects` (lines 12 and 13). The return object of type `org.omg.CORBA.Any` is created using the instance of the ORB currently associated with the skeleton (line 14). The extracted objects are used as parameters of the method delegated to the aggregate to obtain a proxy object as a return value (line 17). The proxy object is inserted into the return object (line 18) before the latter is returned (line 20). Delegating the call to `ProxyFactory` is similar to the way we implemented the RMI remote object `ProxyFactoryWrapper` in Listing 4.5 in the previous section.

The skeleton `ProxyFactoryWrapperPOA` is generated by the idlj compiler and acquires a name of the form `<InterfaceName>POA` as part of the IDL/Java mapping specification. All the skeleton classes generated by applying the idlj-compiler allow our server application to connect to the ORB runtime system and provide marshalling and un-marshalling routines. There is an alternative to this inheritance implementation style of associating object implementation classes with a skeleton class, called *delegation* or the Tie-method. With the Tie-method, `FactoryWrapperImpl` does not extend `ProxyFactoryWrapperPOA` but it implements `ProxyFactoryWrapperOperations` and delegates method invocations to the skeleton class `ProxyFactoryWrapperPOATie`. Generating the required classes to implement the Tie-Method is achieved by using appropriate options when applying the idlj tool.

The Portable Object Adapter (POA) is a component of the ORB structure responsible for looking up and potentially activating the implementation for executing operations.

Listing 4.9 shows a typical implementation of the server application class. In the `main()` method, the class initializes the ORB (line 8), creates the implementation object `theProxyImpl` (lines 12 and 13), and makes it available to the clients by using the CORBA's naming service `NameService` (lines 19 and 20). The implementation object is bound to an arbitrary name (line 21), through which it can be identified by clients. Before giving the object a name, it must be first transformed into a CORBA object reference through the POA (line 15). The object adaptor is obtained by referencing root `POA`, which is always available from the ORB, and the result is narrowed down to a `POA` type (line 10). The object adaptor is activated through the `POAManager` to allow for dispatching incoming requests (line 23). Finally, `run()` is called on the ORB object to allow the main thread to wait and listen for incoming requests (line 24).

Listing 4.10 shows the client implementation class. As in the server case, the client application first initializes the ORB and then uses CORBA's naming service to resolve the object reference (lines 9 - 15). The reference is of type `org.-omg.CORBA.Object` and is narrowed down to the appropriate Java interface type with the help of `ProxyFactoryWrapperHelper` (line 16). `ProxyFactoryWrapper` interface is generated by applying the idlj tool to the .idl-file of Listing 4.7. The tool maps an IDL-interface into a Java interface with the same name and same set of methods. Having obtained the object reference with the appropriate type, we can invoke methods on the object, whose implementations are defined on the server side, as for example, `createProxy(Any, Any)`. To invoke this method, we use the ORB to create two `org.omg.CORBA.Any` objects and insert into these the

```
import org.omg.CORBA.*;                                                 //1
import org.omg.PortableServer.*;                                        //2
import org.omg.CosNaming.*;                                             //3
                                                                        //4
public class ProxyServer {                                              //5
   public static void main(String[] args) {                             //6
     try {                                                              //7
       ORB orb=ORB.init( args, null );   //init orb                     //8
       //init object adapter                                            //9
       POA poa=POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
       // create a ProxyFactoryWrapperImpl object                        //11
       ProxyFactory pf=new ProxyFactory();                               //12
       ProxyFactoryWrapperImpl theProxyImpl=new ProxyFactoryWrapperImpl(pf);
       // export the object reference                                    //14
       org.omg.CORBA.Object o=poa.servant_to_reference(theProxyImpl);    //15
       // print stringiefied object reference                            //16
       System.out.println( orb.object_to_string(o) );                    //17
                                                                         //18
       org.omg.CORBA.Object oRef=orb.resolve_initial_references("NameService");
       NamingContextExt nc=NamingContextExtHelper.narrow(oRef);          //20
       nc.bind( nc.to_name("anyx"), o);                                  //21
       // wait for requests                                              //22
       poa.the_POAManager().activate();                                  //23
       orb.run();                                                        //24
       } catch(SystemException e) { e.printStackTrace();                 //25
       } catch(UserException u)   { u.printStackTrace(); }               //26
   }                                                                     //27
}                                                                        //28
```

Listing 4.9: The CORBA server application

appropriate serializable parameter base objects as values using `insert_Value()` (lines 19 - 30). The result of the method invocation can be extracted from the method's return `Any`-type, using `extract_Value()` and type-casting (lines 32 and 33). In our example, the result is of type `java.lang.reflect.InvocationHandler` and is implemented on the server by `ActivePropertyHandler` thus providing reusable behaviour. Having obtained the result, the client can create a proxy to allow for base object customizations, as in the RMI case.

```java
import java.lang.reflect.*;
import org.omg.CORBA.*;
import org.omg.CosNaming.*;

public class ProxyClient implements java.io.Serializable {

   public static void main(String args[]) {
     try {
       ORB orb = ORB.init( args, null ); // initialize the ORB
       org.omg.CORBA.Object obj = null;  // get object reference

       org.omg.CORBA.Object oRef=orb.resolve_initial_references("NameService");
       NamingContextExt nc=NamingContextExtHelper.narrow(oRef);
       if(nc==null) throw new RuntimeException("narrow failed");
       obj = nc.resolve( nc.to_name("anyx"));
       ProxyFactoryWrapper pfw=ProxyFactoryWrapperHelper.narrow(obj);
       if(pfw==null) System.exit(1);

       // appl object parameters of types IPoint and ProxyClient
       IPoint pt=new Point();
       ProxyClient listener=new ProxyClient();

       // insert paramters into Anys to use in method invocation
       Any anyPoint = orb.create_any();
       anyPoint.insert_Value(pt);
       Any anyListener = orb.create_any();
       anyListener.insert_Value(listener);

       // invoke the operation
       Any anyProxy=pfw.createProxy(anyPoint, anyListener);
       // Extract the result of method invocation
       ActivePropertyHandler ih=
                      (ActivePropertyHandler)anyProxy.extract_Value();
       // Do something with this result like creating a proxy
       // ...
     } catch(UserException u) { ... }
       catch(SystemException ex) { ... }
   }
}
```

Listing 4.10: The CORBA client application

4.2.3 Java's Servlets implementation

Compared to Sockets, the classes `URL` and `URLConnection` of the `java.net` package provide a relatively high-level mechanism and system-independent network communication for accessing resources on the Internet. These classes allow a client application to locate and connect to servlets on the server side. The method `getConnection()` of Listing 4.11 (lines 14 - 21) shows an example of a client using `URL` and `URLConnection` objects to establish a connection to a Web-server. The servlet's location on the Internet is passed as a URL parameter string (lines 7 and 15). After establishing the connection, the client can utilize the `URLConnection` object reference to request services from the server by passing required parameters as serialized objects and obtain the result by de-serializing the received data.

On the server side, the Web-server acts as a servlet container supporting Java's servlets technology and hosts the reusable resources needed to customize the behaviour of application objects on the client side. The servlets act as controllers allowing access to these resources.

To explain the Servlet implementation supporting code reuse, we present the same example considered in the RMI and CORBA-based systems.

Listing 4.11 shows the pseudo-code for an application client utilizing `URL` and `URLConnection` objects (lines 34 and 14 - 21) to request a reference to a proxy object implementing a reusable general behaviour on the server side. The client forwards a request by serializing the request's parameters (lines 22 - 26) as two elements of a vector (lines 35 - 39). The result is de-serialized (lines 27 - 32) and cast to `IPoint` (line 40). As in the previous RMI and CORBA cases, invoking a set method on the returned proxy object (line 41) sets the object attribute to a new value and fires an event of type `PropertyChangeEvent` informing the listener `sclient` of the state change. In this way, the attributes of the base object `ipoint` behave as active properties of JavaBeans.

The servlet referred to in Listing 4.11 is specified in the URL string expression "`http://<server:port/servlet location>`" and runs on a Web-server. In addition, the server also hosts the classes implementing the reusable code of catching and firing `PropertyChangeEvents`. Listing 4.12 shows an implementation of an `HttpServlet`. The servlet processes HTTP requests (i.e., *GET*, *POST*, *PUT*, *DELETE*, *HEAD*) by a client by overriding the corresponding methods. By default, an `HttpServlet` handles an unspecified request as a GET-request and dispatches the processing to the `doGet()` method. We choose to delegate the processing to the `doPost()` method because there is no limit on the data size to be processed (lines 27 - 29). The method uses the `HttpServletRequest` parameter to de-serialize the data (lines 20 and 7 - 12) and obtain the object parameters needed to call the reusable code (lines 21 - 23). The proxy reference is then serialized and sent back to the client using the `HttpServletResponse` parameter of the method (lines 24 and 13 - 17).

By de-serializing the parameters, the server must have access to client classes to call the reusable code. Mechanisms, such as dynamic class-loading as provided by the RMI system or provision of a pool of classes accessible to the server are needed to support the HTTP communication and allow the server to load the client object parameters.

```
import java.io.*;
import java.net.*;
import java.util.Vector;
import java.beans.*;

public class ServletClient implements Serializable, PropertyChangeListener {
   private static final String servlet="http://<resource address>";
   private static Vector ov=new Vector();

   public void propertyChange(PropertyChangeEvent e) {
      // implement the reaction of the listener
   }

   private static URLConnection getConnection() throws IOException {
       URL u = new URL(servlet);
       URLConnection con = u.openConnection();
       // set boolean flags for input, output, cache etc ..
       // request property key & value: Content-type & octet-stream
       con.setRequestProperty("Content-type","application/octet-stream");
       return con;
   }
   private static void sendObject(URLConnection con, Object obj)
                                                      throws IOException {
       ObjectOutputStream out=new ObjectOutputStream(con.getOutputStream());
       if (obj != null) out.writeObject(obj);
       out.close();
   }
   private static Object receiveObject(URLConnection con) throws Exception {
       ObjectInputStream in=new ObjectInputStream(con.getInputStream());
       Object obj = in.readObject();
       in.close();
       return obj ;
   }
   public static void main( String[] args ) throws Exception {
       URLConnection con = getConnection();
       IPoint ipoint=new Point(0, 0);
       PropertyChangeListener sclient=new ServletClient();
       ov.addElement(ipoint);
       ov.addElement(sclient);
       sendObject(con, ov);
       IPoint proxy=(IPoint)receiveObject(con);
       proxy.setx(<arbitrary value>);
   }
}
```

Listing 4.11: Client uses URL objects to communicate with Web-server

```
import java.io.* ;                                                    //1
import javax.servlet.* ;                                              //2
import javax.servlet.http.* ;                                         //3
import java.util.Vector;                                              //4
                                                                      //5
public class ActivePropertyServlet extends HttpServlet {              //6
   private Object receiveObject(HttpServletRequest req) throws Exception {
       ObjectInputStream in=new ObjectInputStream(req.getInputStream()); //8
       Object obj = in.readObject();                                  //9
       in.close();                                                    //10
       return obj ;                                                   //11
   }                                                                  //12
   private void sendObject(HttpServletResponse resp , Object obj)     //13
                                                          throws Exception {
       ObjectOutputStream out=new ObjectOutputStream(resp.getOutputStream());
       out.writeObject(obj);                                          //15
       out.close();                                                   //16
   }                                                                  //17
   public void doPost(HttpServletRequest req , HttpServletResponse resp) {
       try {                                                          //19
           Vector iv=(Vector)receiveObject(req);                      //20
           Object o1=iv.elementAt(0);                                 //21
           Object o2=iv.elementAt(1);                                 //22
           Object proxy=new ProxyFactory().createProxy(o1, o2);       //23
           sendObject(resp , proxy);                                  //24
       } catch (Exception e) { ... }                                  //25
   }                                                                  //26
   public void doGet(HttpServletRequest req, HttpServletResponse resp) { //27
       doPost(req, resp);                                             //28
   }                                                                  //29
}                                                                     //30
```

Listing 4.12: Controller servlet provides clients with ActivePropertyHandlers

4.3 Design Patterns

Design patterns describe general solutions to common design problems. Some design problems appear repeatedly in different contexts; the main motivation behind patterns is to provide abstract solutions applicable in all contexts. In a particular context, however, application-dependent information must be provided. A pattern is described by its name, intent, the contexts where the pattern occurs, the forces or trade-offs resulting from applying the pattern and the solution provided.

Research on design patterns in software was inspired by the work of C. Alexander on the problems of urban architecture [11, 12]. In software, design patterns are used in different application domains and at different levels of software development including architectural design and implementation [29, 37, 46, 47, 84, 98]. The application domains include design of programming languages' libraries (for example, Java's GUI packages), middleware (for example, CORBA) and concurrent systems, to name a few. Gamma et al. provide a systematic approach and a catalogue of twenty-three design patterns [47] in the domain of program design.

In the subsequent analysis, the patterns of Gamma et al. are analysed from the perspective of the reflection model proposed in Chapter 3. Applying the model makes the implementation of some patterns qualitatively simpler. We show that for most patterns, the separation between generic control structure of the pattern and application dependent data helps to reduce the coupling among pattern's participants. This becomes clear from the class diagrams expressed in UML notation. For some other patterns, however, employing the reflection model shows no genuine advantages when compared with the standard (non-reflective) implementation of the patterns. This is due to the limitations of the reflection model and/or to the structure and/or application of the pattern itself. The model does not allow structural changes at runtime. The application domain of some patterns, e.g., compiler construction of the Interpreter pattern, makes the employment of the reflection model inappropriate. In such cases, separating the participants of the pattern into base and meta-objects becomes inadequate. For some patterns, the Java language provides appropriate abstractions, e.g., Iterator and Prototype.

To adapt patterns to our behavioural reflection model, we proceed as follows. We start by analysing the pattern by separating its constituents (participants including control structure) into two parts: the application dependent part and the generic part. If the separation provides a better solution in terms of less coupling and a higher level of code reuse, we provide an alternative implementation using the model. In certain cases, we discuss some pattern implementations in detail with the aim of showing how the procedure works. Code extracts as well as the class diagrams before and after adapting the pattern to the reflection model are shown. To keep UML diagrams simple, we only show the relevant constituents of a structure and give more detail only as needed. The idea is to highlight the resulting class structure of each of the patterns after adaptation to the reflection model.

Gamma et al. classify the patterns in different families according to the pattern's purpose. As a result, three pattern categories emerge: behavioural, structural and creational patterns. Subsequently, we follow this classification and discuss each pattern category in a separate section.

4.3.1 Behavioural Patterns

4.3.1.1 The Template Method Pattern

The Template Method Pattern defines the skeleton of an algorithm and defers some steps to sub-classes without changing the algorithm's structure. The pattern underpins object orientation where inheritance is used to extend and/or modify existing behaviour.

The Template Method pattern serves as a good example of how a dynamic proxy dispatching mechanism can be used as a substitute for inheritance. The "abstract" part of the Template Method algorithm is implemented by super-class(es) and invoked by looking up the methods along the inheritance path. Using our reflection model, this part is moved to the meta-level and invoked through the dispatching mechanism supported by dynamic proxies.

Listing 4.13 and 4.14 show extracts of an implementation of the Template pattern using dynamic proxies. The algorithm consists of three different steps defined at the meta-level (Listing 4.13, line 22). Some of the algorithm's methods are defined at the base level; `BaseImpl` implements the second step and part of the third step (Listing 4.14, lines 6 - 10). An implementation using inheritance replaces the proxy classes (`TemplateMethodHandler` and `ProxyFactory`) with super-classes (to `BaseImpl`) implementing the methods of the algorithm, namely, `firstStep()` and the parts of the `thirdStep()` not implemented in `BaseImpl`.

4.3.1.2 The Strategy Pattern

The Strategy pattern makes a family of encapsulated and related algorithms interchangeable by separating the algorithms from the context in which they are used. Strategy is an example of how our reflection model simplifies the implementation of the pattern. Figure 4.3 depicts the class structure of the Strategy pattern in UML notation. We note the dependency of the context represented by `StrategyContext` on the algorithms represented by `Strategy`.

Different applications provide different families of related algorithms; in other words, strategies are application dependent. In the Strategy pattern, the role of a context object is to control the execution of algorithms and these can be altered without any reference to the context. It follows that strategies can be

```
import java.lang.reflect.*;                                            //1
                                                                       //2
public class ProxyFactory {                                            //3
   public Object createProxy(Object base_obj) throws Throwable {       //4
       // creates and returns a proxy using Proxy.newProxyInstance(...)   //..
       // The handler is of type TemplateMethodHandler
   }
}                                                                      //12
class TemplateMethodHandler implements InvocationHandler {
   private Object base_obj;
    public TemplateMethodHandler(Object base_obj) { this.base_obj=base_obj; }
    public Object invoke(Object x, Method m, Object[] args) throws Throwable {
       try { this.perform();  } catch (Exception e) { ... }
       return null;
    }
                                                                       //20
public void perform() {                                                //21
   firstStep(); ((ITemplateMethod)base_obj).secondStep(); thirdStep();    //22
}                                                                      //23
private void firstStep() { System.out.println( "Metaobject.. firstStep" ); }
private void thirdStep() {                                             //25
   thirdStep_1();
   ((ITemplateMethod)base_obj).thirdStep_2();
   thirdStep_3();
}
private void thirdStep_1() {System.out.println("Metaobject.. thirdStep_1" ); }
private void thirdStep_3() {System.out.println("Metaobject.. thirdStep_3"); }
}                                                                      //32
```

Listing 4.13: An implementation of the Template pattern: the meta-program

```
interface ITemplateMethod {                                            //1
   public void secondStep();                                           //2
   public void thirdStep_2();                                          //3
   public void findSolution();                                         //4
}                                                                      //5
class BaseImpl implements ITemplateMethod {                            //6
   public void secondStep() { System.out.println ("BaseImpl.. secondStep" ); }
   public void thirdStep_2() { System.out.println( "BaseImpl..thirdStep_2" );}
   public void findSolution() { //default impl. }
}                                                                      //10
public class TemplateMethodDemoDynamic {
   public static void main( String[] args) throws Throwable {
       ProxyFactory pfac=new ProxyFactory();
       ITemplateMethod algo=(ITemplateMethod)pfac.createProxy(new BaseImpl());
       algo.findSolution();
   }
}
```

Listing 4.14: An implementation of the Template pattern: the client part

assigned to the base level, because they are application dependent, and contexts can be lifted to the meta-level, because their role is general and independent of algorithms.

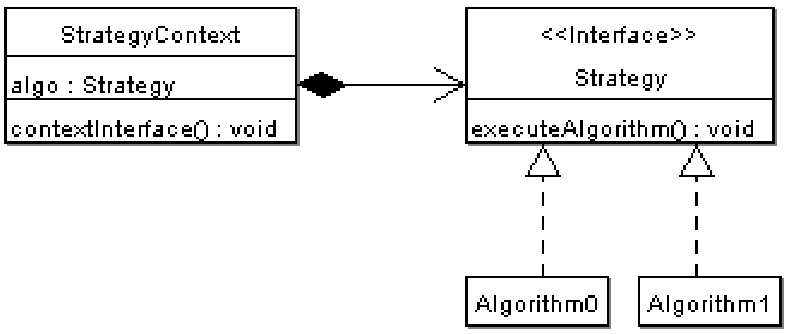

Figure 4.3: Class structure of the Strategy pattern

Figure 4.4 shows the class structure of the Strategy pattern corresponding to an alternative implementation using proxies. The context class is replaced by two classes: `ProxyFactory` and `StrategyHandler`; the latter implements `InvocationHandler` of the `java.lang.reflect` package and corresponds to the meta-object. There is no direct association between application strategy classes at the base level and classes representing context objects at the meta-level. The binding of strategies to context objects is established first at runtime with the help of a `ProxyFactory`. We note that by changing the mechanisms of associating strategies to their contexts, the requirements of the Strategy pattern are preserved, i.e., encapsulating the family of algorithms and making it changeable independent of the context. In addition, contexts are made generic, thus reusable and independent of the algorithms.

With the reflection model supported by the proxies, separating algorithms from contexts is implemented as a direct consequence of the separation between base- and meta-level. Method interception of meta-objects suffices and no manual changes to the static class structure are needed. Binding strategies to context objects is realized at runtime; there is otherwise no coupling between algorithms and their contexts. This is not the case in the static implementation of Figure 4.3; a context object contains a strategy so that method invocations to execute the various algorithms can be delegated.

An extract of an implementation using proxy objects is depicted in Listing 4.15. The class `StrategyHandler` makes use of the dispatching mechanism provided by dynamic proxies to invoke the `executeAlgorithm()` methods on base objects of type `Strategy`.

The class diagrams of Figures 4.3 and 4.4 show that proxy implementation of the pattern exhibits less coupling when compared with direct implementa-

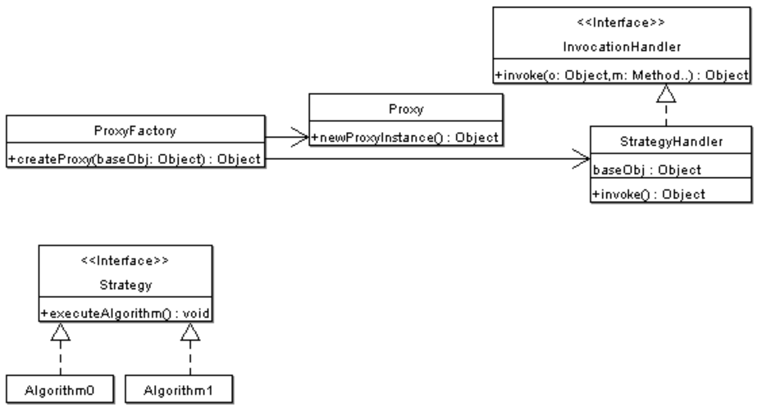
<<Interface>>
InvocationHandler
+invoke(o: Object,m: Method..) : Object
Proxy
+newProxyInstance() : Object
ProxyFactory
+createProxy(baseObj: Object) : Object
StrategyHandler
baseObj : Object
+invoke() : Object
<<Interface>>
Strategy
+executeAlgorithm() : void
Algorithm0
Algorithm1

Figure 4.4: Adapting the Strategy pattern to the reflection model

tion. In the latter, a context is replaced by a meta-object to which control is dispatched by invoking strategy methods on corresponding proxy objects. A context is explicitly separated from the algorithms and it is also independent from them. Furthermore, it is generic and can be reused irrespective of how the algorithms and their methods are implemented.

```
import java.lang.reflect.*;

public class ProxyFactory {
   public Object createProxy(Object algo) throws Throwable {
       // The invocation handler is a StrategyHandler
   }
}
class StrategyHandler implements InvocationHandler {
   private Object delegate;
   public ContextStrategy (Object delegate) {
       this.delegate=delegate;
   }
   public Object invoke(Object p, Method m, Object[] args) throws Throwable {
       Object result = null;
       try { result = method.invoke(delegate, args); }
       catch (InvocationTargetException e) { ... }
       return result;
   }
}
```

Listing 4.15: The meta-level part of the Strategy pattern implementation

4.3.1.3 The State Pattern

The State pattern allows an object to alter its behaviour when its internal state changes. The State pattern has similarities with the Strategy pattern; both are examples of composition and delegation. The difference lies in the intent; a state object encapsulates a state-dependent behaviour and possibly a state transition rule while a strategy object encapsulates an algorithm.

Separating states from their contexts follows the same procedure applied to the Strategy pattern in the previous section. State objects reside in the base level and contexts are meta-objects controlling state transitions. A State transition scheme is defined by the application at the base level. The context object at the meta-level identifies current states including the initial state and invokes the state method on these objects.

In a traditional implementation of the State pattern, the meta-program is replaced by a context class (or classes). The context class maintains a state object and using polymorphism delegates state-specific requests to this object. Unlike the proxy implementation where the context is decoupled from states and coupling becomes active only when requested at runtime, the coupling between contexts and state objects in a traditional implementation holds during the entire lifetime of the context object.

4.3.1.4 The Iterator Pattern

The Iterator pattern provides a way of accessing the elements of an aggregate object sequentially without exposing its underlying representation. Java provides a number of iterator classes in the `java.util` package as part of the language collections framework. These classes support *polymorphic iteration* [47] by providing traverse and access operations independent from the aggregate internal structure. Java Iterators separate the generic control structure of the pattern from specifics of their clients. Consequently, the objectives of applying the reflection model on this pattern are met by the Java library class implementations.

4.3.1.5 The Visitor Pattern

The Visitor pattern separates object structures from operations acting on those objects and thus allows the addition of operations to classes without changing those classes. To achieve this, Visitor uses a double-dispatch technique where dispatching control to a method depends on the name of the request and the types of two receivers, i.e., the object and its visitor.

The Visitor pattern could be considered as a substitute for the double-dispatch mechanism unsupported by mainstream OO languages such as C++, Java and Smalltalk. As observed by Gamma et al. [47], languages that support multiple dispatching (for example CLOS) lessen the need for this pattern.

The class diagram of the Visitor pattern is depicted in Figure 4.5 and Listing 4.16 shows an extract of corresponding implementation. We note the dependency between objects and their visitor; the objects' `accept()` method takes

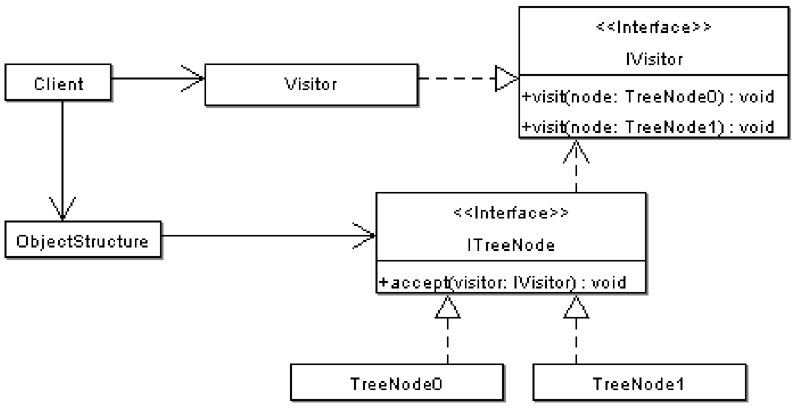
Client
Visitor
<<Interface>>
IVisitor
+visit(node: TreeNode0) : void
+visit(node: TreeNode1) : void
ObjectStructure
<<Interface>>
ITreeNode
+accept(visitor: IVisitor) : void
TreeNode0
TreeNode1

Figure 4.5: Class structure of the Visitor pattern

```
interface ITreeNode { public void accept(IVisitor v); }

class TreeNode_0 implements ITreeNode {
   public void accept(IVisitor v) { v.visit(this); }
   public String visit_treeNode_0() { return "visiting TreeNode_0"; }
}
class TreeNode_1 implements ITreeNode {  // as TreeNode_0  }

interface IVisitor {
   public void visit(TreeNode_0 tn_0);
   public void visit(TreeNode_1 tn_1);
}
class Visitor implements IVisitor {
   public void visit(TreeNode_0 n_0) {
       System.out.println(n_0.visit_treeNode_0());
   }
   public void visit(TreeNode_1 n_1) {
       System.out.println(n_1.visit_treeNode_1());
   }
}

public class VisitorPattern {
  // define a tree of nodes and traverse it applying the visit
}
```

Listing 4.16: An implementation of the Visitor pattern

a IVisitor type as parameter and a visitor operation (visit() method) has a parameter of type ITreeNode. There are two different ITreeNode class types, each of which implements the accept() operation, thus allowing the visitor to act upon its objects. The Visitor class provides implementations of visiting operations for each ITreeNode object type. In running the pattern, the nodes can be traversed in such a way that each object is operated upon by invoking the accept() method.

We can implement the double-dispatch technique by using the dispatch mechanism provided by dynamic proxies. Node objects are separated from the visitor by assigning them to the base level; proxy objects residing at the meta-level take the role of visitors. Once an accept() method is invoked on an ITreeNode object, control is dispatched to the meta-level where reflective code implemented by the meta-object is executed. Figure 4.6 shows the class structure of the Visitor pattern according to this view in UML notation. Note that the classes of proxy objects are completely decoupled from application classes at the base level.

```
import java.lang.reflect.*;

public class ProxyFactory {
   public Object createProxy(Object tnode) throws Throwable {
       // creates and returns a proxy. Handler is of type VisitorHandler
   }
}
class VisitorHandler implements InvocationHandler {
   private Object tnode;
   public VisitorProxy (Object tnode) { this.tnode=tnode; }
   public Object invoke(Object p, Method m, Object[] args) throws Throwable {
       Object result = null;
       try {
           Method[] meths=tnode.getClass().getDeclaredMethods();
           for (int i=0; i < meths.length; i++) {
               if (meths[i].getName().startsWith("visit")) {
                   result=meths[i].invoke(tnode, null);
               }
           }
       } catch (InvocationTargetException e) { ... }
       return result;
   }
}
```

Listing 4.17: An implementation of the Visitor pattern at the meta-level

Figure 4.6 suggests that, at the base level, the method accept() is made independent of the visitor. As a result, ITreeNode classes provide a default (an empty) implementation of the method. The accept() method is invoked on proxy objects triggering the visitor's operation implemented at the meta-level. The result is the same as in the static case. The differences lie in the implementation of the visitor object as well as in its activation.

Listing 4.17 shows an extract from the corresponding Java implementation of the pattern. Instances of class `VisitorHandler` represents visitor objects which operate on base objects by intercepting `accept()` method invocations on them. Reflection is used to invoke the visit methods on base objects. In executing the meta-code, meta-objects assume that objects at the base level provide visiting methods whose names start with "visit".

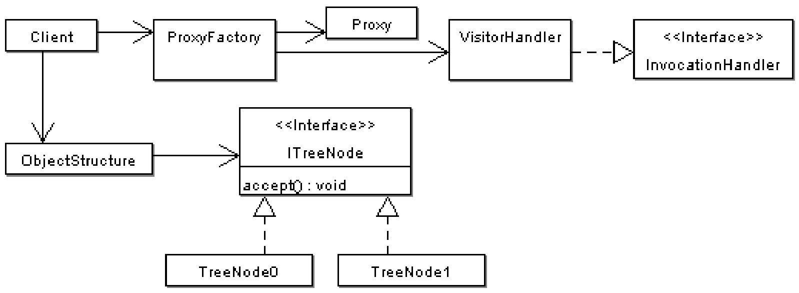

Figure 4.6: Adapting the Visitor pattern to the reflection model

4.3.1.6 The Observer Pattern

The Observer pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. This is equivalent to the MVC pattern where `Views` are dependent on their `Models` and must be informed every time data held by models has changed.

Java provides a number of abstractions that allow direct implementation of the pattern. There is, on the one hand, the class tuple `(Observable, Observer)` in the `java.util` package. `Observable` objects keep track of interested observers. When a change of state of an observable object takes place, all observers are notified and updated automatically. On the other hand, there are the JavaBeans classes `PropertyChangeEvent`, `PropertyChangeListener` and `Property-ChangeSupport`, which support active JavaBeans properties. In Section 3.4.2.2 of the previous chapter, we implemented the Observer pattern using JavaBeans classes. By adapting this implementation to our reflective model, we provided a generic coding of the event-trigger mechanism independent of the application object (the observable). The generic implementation shows the power of our reflection model in supporting code reuse and less coupling.

4.3.1.7 The Command Pattern

The Command pattern encapsulates a request as an object. It decouples clients from objects having the knowledge to perform the requested operations.

Adapting the Command pattern to the proxies' reflective model is straightforward, due to the fact that the model supports decoupling of objects and classes. Clients can be identified with base objects and commands as meta-objects. The meta-objects are assisted by other objects having the knowledge to perform the required operations. Clients' requests are dispatched to the invocation handlers (meta-objects), which pass on the request to appropriate objects for manipulation.

Listing 4.18 shows an example of how two commands are implemented (`LightOnCommand` and `LightOffCommand`) together with a `CommandHandler` controlling the execution of these commands. Switching lights on or off depends on clients' input dispatched to the handler through a proxy object. Listing 4.19 is an example of a client using a proxy to pass its requests to a command controller residing at the meta-level. The coupling between client objects and commands takes places at runtime and is temporal; once the commands are executed the coupling ceases to exist.

The advantages of implementing Command using dynamic proxies are less coupling at the class (and object) level and promotion of modularity as a result of the separation of clients and commands.

4.3.1.8 The Chain of Responsibility Pattern

The Chain of Responsibility pattern avoids coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Receiving objects are chained and the request is passed along the chain until an object handles it.

As for the Command pattern, Chain of Responsibility promotes looser coupling between classes. It forwards requests along a chain of classes whereas the Command pattern forwards a request to a specific class. Adapting the pattern to the reflection model follows the same procedure as that applied to the Command pattern. Request senders are interpreted as base objects and the chain of handlers reside at the meta-level. The proxy's invocation handler receives clients' requests and implements the chaining operation with the support of receiver objects.

4.3.1.9 The Mediator Pattern

The Mediator pattern defines an object that encapsulates how a set of objects interact. The pattern promotes loose coupling by keeping objects from referring to each other explicitly.

We can minimise coupling further by applying our reflective model to Mediator. The role of the mediator object can be implemented at the meta-level. The meta-object keeps a list of all (base) objects and keeps them from referring to each other explicitly. The meta-object implements the interaction strategy;

```
import java.lang.reflect.* ;

public class ProxyFactory {
   public Object createProxy(Object base_obj) throws Throwable {
       // creates a proxy. The invocation handler is CommandHandler
   }
}
class CommandHandler implements InvocationHandler {
   private Object swit;
   private Light light;
   public CommandHandler(Object swit) {
       this.swit=swit;
       light=new Light();
   }
   public Object invoke(Object px, Method m, Object[] args) throws Throwable {
       Object result = null;
       try {
           if (method.getName().equals("switchOnCommand")) {
               new LightOnCommand(light).execute();
           }
           else if (method.getName().equals("switchOffCommand"))  {
               new LightOffCommand(light).execute();
           }
           else result=method.invoke(swit, args);
       } catch (InvocationTargetException e) { ... }
       return result;
   }
}
interface ICommand { public abstract void execute(); }
class Light {
   public void turnOn() { System.out.println("Light is on "); }
   public void turnOff() { System.out.println("Light is off"); }
}
class LightOnCommand implements ICommand {
   private Light myLight;
   public LightOnCommand (Light light) { myLight=light; }
   public void execute() { myLight.turnOn(); }
}
class LightOffCommand implements ICommand {
   private Light myLight;
   public LightOffCommand (Light light) { myLight=light; }
   public void execute() { myLight.turnOff(); }
}
```

Listing 4.18: CommandHandler controls execution of commands

```
interface IRequest {
   public void switchOnCommand ();
   public void switchOffCommand ();
}
class Switch implements IRequest {
   public void switchOnCommand() { }
   public void switchOffCommand() { }
}
public class CommandDemo {
   public static void main(String[] args) throws Throwable {
       ProxyFactory pf=new ProxyFactory();
       IRequest ireq=(IRequest)pf.createProxy(new Switch());
       // use proxy and associated handler to switch on/off
   }
}
```

Listing 4.19: Commands are dispatched to the meta-level using a proxy object

it provides necessary operations to ensure that the objects work properly together. The objects are instantiated at the application level and passed to the same meta-object upon creation of corresponding proxies. Method invocations are trapped by the meta-object where they are introspected and handled according to the interaction strategy.

Applying the reflection model promotes loose coupling of Mediator further; the set of objects whose interactions are to be controlled by the Mediator but are passed at runtime and have no reference to the Mediator (of type InvocationHandler) at the class level. In a non-reflective implementation of the pattern, there is an additional coupling resulting from the dependency of interacting objects on the Mediator at compile-time.

4.3.1.10 The Interpreter Pattern

The Interpreter pattern defines a representation for the grammar of a language along with an interpreter that uses the representation to interpret sentences in the language. Implementing the Interpreter pattern requires routine compiler techniques including parsing the language expressions, interpreting the resulting syntax tree into actions and then executing the actions to be exercised. Error checking is also required to handle grammatical statements with incorrect syntax.

The Interpreter pattern is useful in cases where a *simple* compiler is needed to handle a particular set of expressions of an arbitrary application. The compiler construction process consists of several consecutive steps (scanning, parsing, code generation etc) normally executed in a specified order. Inserting proxies in the compilation process as a result of applying our reflection model is not beneficial, because it complicates the process with no apparent gains.

4.3.2 Structural Patterns

4.3.2.1 The Adapter Pattern

```
import java.lang.reflect.*;

public class ProxyFactory {
  public Object createProxy(Object base_obj) throws Throwable {
    // creates & return a proxy. Handler is a StackHandler
  }
}
class StackHandler implements InvocationHandler {
  private Object stack;
  private java.util.Vector adaptee;
  public StackHandler(Object stack) {
    this.stack=stack;
    adaptee=new java.util.Vector();
  }
  public Object invoke(Object pxy, Method m, Object[] args) throws Throwable {
    Object result = null;
    try {
       if (method.getName().equals("push")) adaptee.addElement(args[0]);
       else if (method.getName().equals("pop")) {
          Object le = adaptee.lastElement();
          adaptee.removeElementAt(adaptee.size() - 1);
          result=le;
       }
       else if (method.getName().equals("top")) result=adaptee.lastElement();
       else result=method.invoke(stack, args);
    } catch (InvocationTargetException e) { ... }
    return result;
  }
}
```

Listing 4.20: A Vector object as adaptee

The Adapter pattern converts the interface of one class into that of another. The intention of the pattern is to convert the interface of a class to match the requirements of a client class.

There are two ways to implement Adapter: through inheritance and through composition [47]. The two approaches are termed class adapter and object adapter, respectively. The non-conforming class is referred to as the adaptee and the new class as the adapter. In the case of inheritance, the adapter is defined as a sub-class of the adaptee and the methods required to match the desired interface are added to it. In the case of composition, the adaptee is included inside the adapter and acts as a delegate to method calls invoked on adapter objects.

Adapters are intermediary entities between adaptees and client objects. Proxy objects can play this intermediary role of adapters. The difference to the class and object adapter implementations is that a dynamic proxy (as adapter) acquires the target interface at runtime. It is independent of this interface and uses reflection to identify method calls. The proxy depends on the adaptee to implement the required functionality, as in the class and object adapter implementation. Furthermore, the proxy implementation does not substitute the original target class with the adapter class, but keeps this class and represents its objects at runtime.

Listing 4.20 shows an example of how our model can be used to implement the Adapter pattern in such a way that a proxy object plays the role of an adapter. In this extract, the handler uses a `java.util.Vector` object (the adaptee) as delegate to implement the required functionality. Listing 4.21 shows an example of how a target class interface `BaseStack` can be converted into a `Vector` interface by passing the client request to the meta-object using a proxy. We note that the only change required to adapt the default implementation represented by `BaseStack` is to divert the method invocation to the proxy object.

```
import java.util.*;
interface IStack {
   public void push(Object v);
   public Object pop( );
   public Object top( );
   public boolean empty( );
}
class BaseStack implements IStack {
   public void push(Object v) { // default impl }
   public Object pop( ) { // default impl  }
   public Object top( ) {  // default impl }
   public boolean empty( ) { return true; }
}
public class AdapterDemo {
   public static void main(String [] args) throws Throwable {
       ProxyFactory fac=new ProxyFactory();
       IStack stack=(IStack)fac.createProxy(new BaseStack());
       // stack allows adaptation of default impl using a Vector as adaptee
   }
}
```

Listing 4.21: Adapting default implementation of a stack using a proxy

4.3.2.2 The Decorator Pattern

The Decorator pattern attaches additional responsibilities to an object dynamically. It provides a flexible alternative to sub-classing for extending functionality. Decorator and Adapter are wrapper patterns. While Decorator wraps and adds new responsibilities to a core object without changing its interface, Adapter changes the adaptee's interface.

Decorator can be easily adapted to the proposed reflection model. Core objects residing at the base level can be decorated by attaching them to meta-objects. Meta-objects extend the functionalities of their referents before delegating control back to them.

4.3.2.3 The Memento Pattern

The Memento pattern stores the current state of an object in a memento object. The object's state can be later restored without exposing the internal representation of the object to the outside world.

We can adapt the Memento pattern to the proposed reflection model by letting meta-objects implement the undo and save operations invoked on their referents. Here, we assume that the internal states of base objects need to be stored by mementos. When the undo operation is invoked, a meta-object asks the base object to retrieve stored data from its memento. When the save operation is invoked, a meta-object allows its referent to create a new memento object and store data in the newly created object. The class relationships between base objects and their mementos remain the same as in the non-reflective case.

Listing 4.22 shows an extract of an implementation of the undo and save operations using our proxy-based reflection model. The meta-object (of type `MementoHandler`) holds references to the base object (of type <`Object_Type`>) and its memento (of type <`Memento_Type`>) passed as parameters upon creation (lines 16 - 19). When a `save()` or an `undo()` method is invoked at the base level, control is dispatched to the meta-object. In case of `save()`, a memento object is created and the base object fields are set (lines 22 - 25). In case of `undo()`, the fields are retrieved by invoking `setMemento((<Memento_Type>)mem)` and the base object is returned with the retrieved values (lines 26 - 30).

The advantage of implementing Memento using dynamic proxies is the separation of application objects from objects manipulating the states of these objects, thus increasing the modularity level. This separation is particularly useful when save and undo are part of typical database transactions. In this case, meta-objects can act as buffers holding application objects' states and as initiators of database actions, thus separating object state manipulations from object data.

4.3.2.4 The Bridge Pattern

The Bridge pattern is meant to decouple an abstraction from its implementation so that the two can vary independently.

The Bridge pattern supports the principle of abstraction in software engineering, according to which implementation details are separated from and hidden behind an application's interface. The proposed reflection model sup-

```
import java.lang.reflect.*;

public class ProxyFactory {
  public Object createProxy(Object base_obj, Object mem) throws Throwable {
       // ...
  }
}
class MementoHandler implements InvocationHandler {
  private Object base_obj;
  private Object mem;
  public MementoHandler(Object base_obj, Object mem) {
    this.base_obj=base_obj;
    this.mem=mem;
  }
  public Object invoke(Object pxy, Method m, Object[] args) throws Throwable {
    try {
      if (method.getName().equals("save")) {
        mem=(<Memento_Type>)(((<Object_Type>)base_obj).createMemento());
        // set object's fields
      }
      else if (method.getName().equals("undo")) {
         if (mem==null) throw new Exception("No Mememnto object ...");
            ((<Object_Type>)base_obj).setMemento((<Memento_Type>)mem);
            return ((<Object_Type>)base_obj);
         }
      } catch (InvocationTargetException e) { ... }
      return null;
   }
}
```

Listing 4.22: Adapting default implementation of a stack using a proxy

ports the principle of abstraction. However, the model allows clients to modify application behaviour through a meta-interface. The modification is carried out without modifying the default implementation. Adapting the Bridge pattern to the proposed reflection model is useful only if the new feature of the model, namely, open implementation, can be used effectively and the benefits gained outweigh the complexity of adding this new feature.

4.3.2.5 The Composite Pattern

The Composite pattern composes objects into tree structures to represent whole-part hierarchies and allows individual objects (leaves) and composites (nodes) to be treated uniformly. The pattern is used to describe a recursive composition in such a way that simple and compound objects are treated equally.

Figure 4.7 shows the class structure of the Composite pattern. The programming units `IComponent` and `Component` represent abstractions providing the basic functionality of a recursive structure and allow a uniform treatment of simple and compound objects. The `Leaf` class represents simple objects and

```
interface IComponent {
   public void specify();
   public void add(IComponent c) throws Exception;
   public void remove() throws Exception;
   public IComponent getFirst();
   public IComponent getRest();
}
class Component implements IComponent {
   // provides a default impl of IComponent. Throws exceptions
   // for add/remove and returns nulls for getFirst and getLast
}
class Leaf extends Component {
   public String name;
   public int id;
   public Leaf(String name, int id) { this.name=name; this.id=id; }
   public void specify() { // specify a leaf ... }
}
class Composite extends Component {
   // implements stack structure recursively
   IComponent first; IComponent rest;
   Composite cons; // will be first defined by calling add
   public Composite() { first=null; rest=null; }
   public Composite(IComponent f, IComponent r) {
     this.first=f; this.rest=r;
   }
   public void specify() {
     IComponent en=cons.getFirst();
     IComponent re=cons.getRest();
     while (en!=null && re!=null) {
        en.specify(); en=(re.getFirst()); re =re.getRest();
     }
   }
   public void add(IComponent o) {
     if (cons==null) cons=new Composite();
     cons=new Composite(o, cons);
   }
   public void remove () { // for simplicity removes 1st object
     cons.first=(cons.getRest()).getFirst();
     cons.rest =(cons.getRest()).getRest();
   }
   public IComponent getFirst() { return this.first; }
   public IComponent getRest()  { return this.rest; }
}
```

Listing 4.23: An implementation of Composite without meta-programming

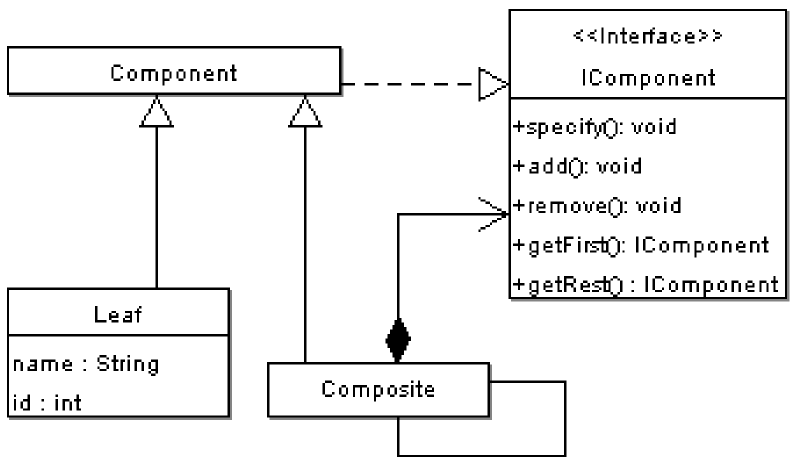

Figure 4.7: Class structure of the Composite pattern

hence inherits the default behaviour defined by its superclass, `Component`. By default, a `Component` cannot add to, remove from, or navigate through the object structure. Class `Leaf` defines individual object attributes and how objects specify themselves. Class `Composite` represents compound objects and defines the functionality needed to manipulate and navigate through the objects' structure in the form of a stack. The stack is constructed recursively through the `add()` method. The recursive structure of Composite is represented by a self-association as shown in Figure 4.7. Specifying a composite object requires traversing its structure and, at each entry, calling the corresponding `specify()` method. The `remove()` method removes the first object from the list. Listing 4.23 shows an extract of a Java implementation of the pattern without using meta-level facilities provided by the proxies.

To adapt the static structure of the Composite pattern to the reflective two-level architecture, we need to identify the generic aspects of the pattern and separate them from those that are application specific. Simple objects with their attributes and specifications are application dependent. Recursive structures are abstract data types and are inherently generic. Following this view, simple and compound objects can be treated uniformly and classified as base and meta-objects, respectively. Figure 4.8 shows the class structure of the Composite pattern according to this classification. Note that `CompositeHandler` refers to itself reflecting the recursive structure of the stack. As in previous cases, the class diagram shows no coupling between classes of the two levels.

The base level implementation of the Composite pattern can be deduced from Figure 4.8. The abstraction, `IComponent`, which defines the recursive structure and represents simple and compound components, remains the same as in

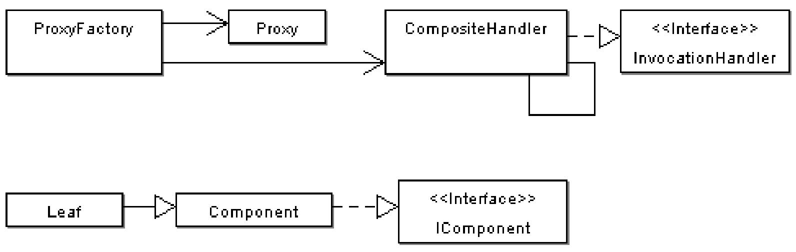

Figure 4.8: Adapting the Composite pattern to the reflection model

the static case. Similarly, class `Component` and class `Leaf` also remain the same as before. To run the pattern application, a meta-object is reified upon instantiating a proxy using the factory method of `ProxyFactory`. The result is identical to that of the static case. The difference is that, in the dynamic case, manipulating a compound object is realized at the meta-level in a generic manner without reference to the base class types, `IComponent`, `Component` and `Leaf`.

Listing 4.24 shows an extract from the Composite pattern implementation at the meta-level. The class `CompositeHandler` represents compound objects (lines 14 - 35). As with the `Composite` class in the static case, it implements all the methods of the component interface. The major difference is that `CompositeHandler` makes no reference to base object types. To manipulate the recursive structure of a compound object, the methods of adding or removing simple objects, navigating through the structure and specifying the entries are intercepted and delegated to the generic `invoke()` method (lines 22 - 34). With the `specify()` method, the recursive structure is traversed and at each entry the corresponding method of simple objects is called back.

The advantages of separating the generic part of the Composite pattern from the application specific part are less coupling, reuse of the meta-code, and support for separation of concerns (the base level from the meta-level). The combination of these features allows for a better understanding of design and for potentially reduced maintenance costs.

4.3.2.6 The Facade Pattern

The Facade pattern provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use. This can be used to simplify a number of complicated object interactions into a single interface.

Facade can be seen as an extended form of the Adapter pattern; while an adapter wraps one object, facade wraps a number of existing objects. As in the case of Adapter, a dynamic proxy can play the role of an intermediary between client and a set of objects. In this case, objects are attached to the facade at

```
import java.lang.reflect.*;

public class ProxyFactory {
   public Object createProxy(Object delegate) throws Throwable {
    // ... The invocation handler is a CompositeHandler
   }
}

class CompositeHandler implements InvocationHandler {
   private Object delegate;
   CompositeProxy cons;
   // define first and rest variable ...
   public ContextComposite (Object delegate) {
       this.delegate=delegate;
       cons=new ContextComposite ();
   }
   public Object invoke(Object pxy, Method m, Object[] args) throws Throwable {
       Object result = null;
       try {
         if (method.getName().equals("add")) {
           cons=new CompositeProxy(args[0], cons);
         }
         else if (method.getName().equals("specify")) { // as in static case }
         else if (method.getName().equals("remove")) { // as in static case }
         else if (method.getName().equals("getFirst")) cons.getFirst();
         else if (method.getName().equals("getRest")) cons.getRest();
       } catch (InvocationTargetException e) { ... }
       return result;
    }
   }
```

Listing 4.24: The meta-level part of the Composite pattern implementation

runtime. The advantages of using a dynamic proxy as facade are flexibility and less coupling.

4.3.2.7 The Flyweight Pattern

The Flyweight pattern uses sharing to support large numbers of fine-grained objects efficiently. The participants involved in the pattern are the `Flyweight`, the `ConcreteFlyweight`, the `UnsharedConcreteFlyweight`, the `Client` and the `FlyweightFactory`. `ConcreteFlyweight` implements the `Flyweight` interface and stores intrinsic states independent of its objects context. A `ConcreteFlyweight` object must be shareable unlike an `UnsharedConcreteFlyweight`. The `Flyweight-Factory` serves to deliver particular flyweights when requested. The factory is passed certain properties and returns the requested flyweight if it already exists; otherwise it creates and returns a new flyweight. In addition, there is the `Context` class, which acts as a repository of an extrinsic state. When creating a new object, a `Client` assigns a flyweight to the object and computes its extrinsic state.

We note that the role of `Context` together with `FlyweightFactory` is to control the flow of flyweights required by a `Client`. On the other hand, `Flyweight`, `ConcreteFlyweight`, and `UnsharedConcreteFlyweight` provide the data structure whose storage is to be efficiently manipulated.

We can employ the reflection model and separate the control classes from the structural classes. However, because of the large number of objects involved, we expect computation time to increase due to the reflection overhead caused by the proxy mechanism. In addition, employing the reflection model does not reduce the level of coupling, because the factory must know the flyweight structure (i.e., type) in order to decide whether to create a new flyweight or deliver an existing one.

4.3.2.8 The Proxy Pattern

The intent of a normal proxy is to provide a surrogate or placeholder for another object to control access to it. Dynamic proxies, on which our reflective model is based, play the same role with additional advantages of less coupling and a higher degree of code reuse. The type and the relationship of a normal proxy to target objects are fixed at compile-time. Those of a dynamic proxy are first determined at runtime. Dynamic proxies allow, using polymorphism and reflection, for implementing generic behaviour independent of the targets' type.

4.3.3 Creational Patterns

Factory patterns provide an interface for creating an object or families of related objects. Depending on runtime data, the factory decides which instance of the several possible subclasses or class hierarchies to return and returns one. Singleton ensures a class only has one instance and provides a global point to access it.

As observed by Norvig [82] and Sullivan [93], languages that allow overriding of the default creation method provide direct support for the Factory Method and Singleton patterns. Also, for languages in which classes are first-class entities, there is no need for abstract factories since classes themselves serve as factories. The behavioural reflection model supported by dynamic proxies does not allow redefinition of Java's `new()` method for object creation independently of Factory classes. Moreover, as noted in Section 3.6 of the previous chapter, the proposed reflection model does not change the underlying object model. Classes are not first-class, they are strictly compile-time entities and their instances do not understand creation messages. As a result, there are no benefits in employing proxies for implementing Factory Method, Abstract Factory and Singleton patterns.

The Builder pattern separates the construction of a complex object from its representation (type and content) so that the same construction process can create different representations. It is similar to Abstract Factory but objects are created in more than one step. As for the factory patterns, applying the proxies' reflection model has no apparent advantages when compared with the static implementation.

The Prototype pattern is used when it becomes more appropriate to create objects by cloning rather than by instantiating the class. Java's interface `Cloneable` provides an implementation of the pattern. Objects must implement this interface and override the `clone()` method of the root class `java.lang.Object` before they can be copied. The method performs a shallow copy of the object, i.e., the copy points to the same objects the copied object points to. The method copies values of the fields in the new object, not the actual objects (references) the original object points to. If a shallow copy of the data is sufficient, the Java implementation of Prototype is simple.

4.4 Discussion

Extending the reflective architecture to incorporate the client/server model is straightforward and varies only according to the networking technology used to implement the communication protocol. In the client/server model, meta-objects providing customisation services become remote objects, running on a different JVM (on the server side) as their clients at the base level.

We have presented three implementation approaches for constructing distributed systems based on dynamic proxies. Part of any analysis should be a comparison of those approaches to assess the relative merits of each. In doing so, we concentrate on the salient features of the different architectures.

RMI and CORBA are considered as two platforms specifically designed for developing OO distributed systems and as such share common features. Java Servlets, on the other hand, provide an API allowing HTTP communications between clients and servers according to the request/response model. Compared to the RMI and CORBA/IIOP protocols, HTTP is stateless and suitable for text transfer only; objects must be serialized and de-serialized before being sent and received along the communication wire. Enterprise applications require stateful communication sessions, security and transaction services. Servlets provide no naming service, no dynamic class loading and there is no security mechanism readily available for them (access security is normally managed by the servlet engine, i.e., the Web server).

RMI and CORBA provide such services and more. RMI and CORBA provide similar mechanisms for transparently accessing remote objects. The most important difference between the two technologies lies in the portability across language and operating systems platforms. RMI can only operate with Java systems and is thus tied to platforms with Java support whereas CORBA supports heterogeneous systems written in different languages and distributed over different platforms. RMI provides no support for legacy systems written, for example, in Fortran and Cobol; in CORBA, server implementations can be written in various languages and accessed by any language with an IDL mapping. Although developing distributed applications follows a similar procedure in both frameworks, CORBA requires knowledge of the IDL and IDL mapping of the implementation language. IDL language mappings are specified by the OMG and mapping tools for generating stubs and skeletons required to establish the connection between clients and servers over the ORB would be needed.

In addition to the difference in the definition of the client server interface (RMI uses Java interface construct and CORBA the IDL), there is a slight difference in their architectures. In RMI, stubs and skeletons are generated from implementation objects. In CORBA, stubs and skeletons are formed from the IDL interface. As a result, in RMI, the generated stub is a proxy for all the remote interfaces of the remote object whereas in CORBA each interface requires its own stub.

We perceive four key advantages to implementing reusable code on the server side with the help of dynamic proxies. Firstly, no changes on the client side are required and the client applications remain executable without being attached to the meta-code on the server side. Secondly, the meta-code is held separated from base applications. This is a desirable design feature since it reduces complexity and enhances the readability of the code. Thirdly, the class structure shows less coupling when compared with the corresponding static case and the object coupling between both sides (clients and servers) is flexible; one meta-object can be used to represent several base objects and vice-versa. Finally, with the help of polymorphism and reflection, customisation code is implemented independently and without any reference to particular applications. Variables of generic types such as `java.lang.Object` are used to hold application objects and reflection is used to manipulate object data at runtime.

Table 4.1 summarizes the impact of applying the proposed reflection model on design patterns in terms of prominent features. The features include the usefulness of applying the model, whether applying the model simplifies the pattern implementation, promotes loose coupling and increases the level of code reuse as well as modularity. Applying the model is not useful if it complicates the implementation without apparent benefits. The conclusions related to loose coupling and code reuse are derived intuitively from class diagrams and coding examples.

Most pattern implementations resulted from applying the model show a higher level of modularity, due to the fact that the proposed model supports the separation of application objects from their meta-representations. Recall that in OO, a module is identified more closely with a class [58]. We note that

Pattern's name	**Prominent features of applying the reflection model**
Template Method	Substitutes inheritance by object composition
Strategy	Simplifies pattern implementation
State	Similar to Strategy pattern
Iterator	Not useful. Java's abstractions are satisfactory
Visitor	Promotes loose coupling
Observer	Using JavaBeans, increases level of code reuse
Command	Increases level of modularity and promotes loose coupling
Chain of Responsibility	Similar to Command pattern
Mediator	Promotes loose coupling further
Interpreter	Not useful
Adapter	Provides a more flexible implementation
Decorator	Similar to Adapter
Memento	Supports separation of concerns
Bridge	Useful in case open implementation feature is required
Composite	Promotes loose coupling and code reuse
Facade	Promotes loose coupling and code reuse
Flyweight	Not useful, due to costs of reflection overhead
Proxy	Not applicable
Creational patterns	Not useful, because model cannot be utilized to redefine new()

Table 4.1: Features of design patterns when adapted to the reflection model

some pattern implementations do not lead to autonomous and self-contained modules independent of application specifics. For example, the Memento pattern implementation separates application objects from (meta-) objects manipulating their states; the meta-program, however, depends on the types of application objects (See Listing 4.22).

For the Template Method pattern, the reflection model substitutes inheritance with object composition. For Strategy and State, applying the model simplifies pattern implementation. For Iterator and Prototype (a creational pattern), Java provides appropriate abstractions as library classes that allow for a direct implementation. Applying the reflection model to the Observer pattern using JavaBeans classes allows for a generic implementation of event-trigger mechanism, thus increasing the level of code reuse. For Adapter and Decorator, the model provides a more flexible implementation; the adaptee is added through a proxy and the default implementation is kept as is. For some patterns, applying the model promotes loose coupling arising from the movement of some participants to the meta-level. For Interpreter and Flyweight patterns, applying the model complicates the implementation and worsens the performance. The reflection model is not useful when applied to creational patterns, because the model cannot be utilized to redefine the `new()` method.

Empirical investigations are required to support or refute the claims about coupling and reuse features exhibited by the pattern. We accept that we have not so far supported our claims with software metrics. In Chapters 5 and 6 we address this question in terms of appropriate coupling and code reuse metrics.

Case studies indicate that reflective systems exhibit less coupling and higher level of code reuse when compared with identical non-reflective systems.

4.5 Conclusions

In this chapter, the behavioural reflection model was extended to include the client/server programming model. To provide an enterprise solution with multiple JVMs connected over a TCP/IP network while retaining the semantics of both levels as is, we transform the single application into a client/server application. The transformation is realized by adapting the single application code, firstly, to the requirements of an RMI based system, secondly, to those of a CORBA IDL programming model and finally, to the requirements of request/response model supported by HTTP servlets.

As in the case of the single application model, we used the reflection model to implement reusable code. In the case of distributed systems, this code is implemented on the server side and accessed by clients via RMI, CORBA/IIOP or HTTP protocols. The question of comparing our approach to code reuse with the standard OO mechanisms such as inheritance and aggregation (composition) will be discussed in detail in Chapter 6.

Applying the reflection model to design patterns showed that some pattern implementations become simpler. For a number of patterns the implementation exhibited less coupling, a higher degree of modularity and greater reuse. However, for many patterns, in particular creational patterns, the reflection model showed no apparent benefits over the traditional implementation.

Elevating design patterns to the meta-level has many advantages. In addition to being reusable design assets, patterns can be coded as abstract reusable classes. Patterns' code is implemented separately from client applications and can be developed and maintained independently. As the pattern code is localised (at the meta-level) and not scattered over the application, design comprehension and program code are potentially improved and maintenance costs potentially reduced.

Chapter 5

Coupling Metrics

In Chapters 3 and 4, we introduced the notion of reflective systems and amended Java's introspection with a behavioural reflection model. In this chapter, we study the object coupling of reflective systems implementing the proposed model. We demonstrate how metrics applied to systems employing the model can provide quantifiable benefits in terms of reduced coupling. In terms of its contribution to the Thesis, this chapter provides a metric theoretic approach to quantifying the coupling level of systems employing the model in comparison with identical models following the classical OO approach. In Chapter 6, we apply the same approach in the context of code reuse.

One important aspect of reflection is the ability to manipulate a program's execution state at runtime. The behaviour and the structure of a reflective system changes during program execution. Here, program execution can be thought of as an ordered sequence of program steps in time and the program state at one step in the sequence is the sum over the states of all objects. A change of behaviour or structure (or both) reflects a change of coupling between objects.

Various OO coupling metrics have been proposed and used in past empirical investigations [15, 26, 35, 41, 50, 60, 73]; none of these metrics, however, take the dynamic change of objects' coupling into account. Yacoub et al. [105] define object coupling metrics. The authors, however, follow a static approach based on parsing UML design documents to calculate the measures. We believe that any measurement of coupling should include changes taking place at runtime. Arisholm [13] argues that static coupling measures are insufficient for capturing coupling aspects of systems related to the dynamic behavior of software. The author remarks that in the case of inheritance and due to polymorphism, it is not always possible to determine the actual receiver and sender objects from static code analysis. In recent work [14], Arisholm, Briand and Føyen observe that regardless of the structural attribute used to define the coupling metric, predictions of external attributes based on a static analysis of the design or code become imprecise when inheritance and polymorphism are used intensively in the systems under study. The authors argue that inheritance and polymorphism will be used more frequently to improve internal reuse in a system and facilitate maintenance, as the use of OO design and programming matures in the industry.

In reflective systems, object interactions at runtime are not entirely determined by their static class couplings. Static metrics relying on class structure relationships are inadequate when reflective mechanisms are used. To understand the coupling behaviour of reflective systems, we need to define metrics that take object interactions into account. We also need to develop appropriate measuring tools that allow us to collect coupling data at runtime.

In this chapter, we introduce a dynamic coupling metric, examine its theoretical foundations and conduct an empirical validation with the support of a measuring tool developed for collecting coupling data at runtime. Our hypothesis is that reflective systems based on separating base objects from meta-level objects show less runtime coupling when compared to equivalent static systems exhibiting the same behaviour. This hypothesis is based on the observation that in reflective systems, objects can interact by passing messages without their respective classes being directly related at the class level. In addition to inheritance, polymorphism and introspection, a reflective system is supported by runtime mechanisms such as behavioural and/or structural reflection that allow the program to manipulate its own state. This allows for a flexible object communication and less object coupling as a result. Introduction of a dynamic coupling metric allows us to demonstrate quantifiably and in a practical sense the relative coupling of classes within OO applications. We view such a metric as a useful approach serving the theoretical evaluation of reflection as well as a practical tool during application development. The proposed dynamic coupling metric applies equally to reflection-based (employing a reflection model) as well as to OO systems in general.

This chapter is organised as follows. The next section provides an introduction to software metrics and presents the underlying measurement theory. Section 5.2 defines a dynamic coupling metric as a mapping of object entities into the set of real numbers, where the attribute of interest is object coupling. In Section 5.3, we address the theoretical validation of the metric. Two sets of measurement principles are considered as evaluation criteria. The problems of applying these criteria to our dynamic metric are discussed. Section 5.4 investigates the prospect of empirical validation. A measuring tool is developed and used in a case study conducted for testing the hypothesis empirically. In Section 5.5, the proposed metric is compared with static metrics based on OO class relations such as inheritance and aggregation and we conclude the chapter in Section 5.6.

5.1 Software Metrics

Applying quantitative measures - known as software metrics - to assess the quality of software design and code has a great impact on the ease of implementation, maintainability and reliability of software [42, 59, 61]. The underlying theory behind software metrics is measurement theory. This section introduces the basic concepts of this theory and its application in the domain of software system development and maintenance. The introduction serves as the basis for the definition of coupling and code reuse metrics introduced in this and the next chapter.

Measurement theory is concerned with the connection between data and the real world. The basic concept of measurement theory is the correspondence or mapping between the real world (modelled as a set of entities with attributes) and a set of numbers or symbols - normally the set of real numbers, R. Measurements are real numbers and the attributes being measured represent features or properties of entities of the system under study. The fundamental ideas are

1) measurement data is not the same as the attributes being measured, and

2) conclusions about attributes should take into account the correspondence between measurements and attributes.

When combined with the theory of statistics, measurement theory can be used to derive meaningful inferences and make prdictions about the real world.

Fenton and Pfleeger [42] define the notion of a *measurement* as “the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules”. The authors provide another definition of measurement as “a mapping from the empirical world to the formal, relational world”. They introduce the notion of a *measure* as “the number or symbol assigned to an entity by the measurement mapping in order to characterise an attribute”. We refer to measures as metrics and introduce a *metric* as a function that defines a correspondence or a measurement mapping between system entities and the set of real numbers. An example of an entity is source code. The size of the code is represented by an attribute and one possible metric for measuring the size of code is the number of Lines Of Code (LOC). Statements can be made about sets of entities by use of comparison operators on their corresponding attribute values. Thus, if e_0 and e_1 are two source code entities and $LOC(e_0) \leq 2 * LOC(e_1)$, then we say the size of e_0 is at most double the size of e_1.

Further useful definitions include measurement scale, the corresponding set of admissible transformations and the nature of attributes [42]. A *measurement scale* refers to the entirety of measurement mapping, i.e., the metric function, the function’s domain and its range. The domain of the function is the set of system entities in the empirical or real world and the range is the set of real numbers or any set of symbols. An *admissible transformation* is a transformation of a measurement scale that preserves the relationships of the measurement. For example, source code e_0 is "shorter than" e_1 corresponds through a metric definition, LOC, to $LOC(e_0) < LOC(e_1)$. This numerical relationship and its correspondence to real world entities are not affected by an order preserving operation on the metric such as multiplying the metric values by a positive real constant. An *internal attribute* of an entity is an attribute that can be measured purely in terms of the entity itself, for example, the size of a piece of some code. An *external attribute* is one that can only be measured with respect to how the entity relates to its environment, for example, time required to write a given piece of source code.

Applying measurement theory to software implies restricting the real world or narrowing the application domain of the general theory to software systems.

5.1.1 Metric scales

There are at least five different scales of measurements. Each scale is associated with a set of permissible transformations. The scales form a partially ordered set according to the restriction imposed by the set of admissible transformations. An account of this approach can be found in [42].

1. Nominal scale: In this scale, which is the weakest of all scales, the real world consists of entities that are classified into sets with no ordering. Numerical labelling (the metric) is totally arbitrary and entities with the same label have the same attribute. A metric on this scale maps distinct entities into distinct numbers. For example, consider an OO system consisting of a finite set of classes. Any mapping which assigns a *unique* symbol to each entity in the set as a characterisation of some attribute shared by all entities, is a nominal scale metric. The values assigned are arbitrary with no order. The class of admissible transformations for a nominal scale metric is the set of all one-to-one mappings.

2. Ordinal scale: In this scale, entities are ordered according to a pairwise comparison and the corresponding metric values reflect an ordering relation defined by the attribute in the set of entities. Consider the OO system of the previous paragraph. If the attribute is the *complexity* of a class, then a metric which assigns to each class a distinct number (ϵ R) representing the class complexity, is an ordinal scale metric. We note that, the ordering induced by the metric in R must match the ordering defined by the attribute in the real world system. The class of admissible transformations for an ordinal scale metric is the set of all monotonic functions that preserve ordering.

3. Interval scale: In this scale, entities are assigned metric values such that differences between these values reflect differences between corresponding

attribute values defined by the ordering relation in the set of entities. Consider the same OO system of the previous paragraph. Let the attribute be the *level of dependency* of a class where the levels are ordered and the difference between consecutive levels is constant. A mapping which assigns to each class a real number representing its level of dependency such that difference between neighbouring entities stays the same, is a interval scale metric. In the interval scale, the origin and the unit of measurement are arbitrary. The class of admissible transformations for an interval scale metric is the set of all affine transformations of the form: $\mu(e) \rightarrow \mu'(e) = a\mu(e) + b$, where μ represents the metric function, a and b are real constants with $a > 0$.

4. Ratio scale: Here, entities are assigned metric values such that differences and ratios between these real number values reflect differences and ratios between corresponding attribute values defined by the ordering relation in the set of entities. Compared with the interval scale, only the unit of measurement is arbitrary. The class of admissible transformations for a ratio scale metric is the set of all linear transformations of the form: $\mu(e) \rightarrow \mu'(e) = a\ \mu(e)$, where μ represents the metric function and a is a strictly positive real constant.

5. Absolute scale: The strongest of scales. In this scale, entities are assigned real numbers such that all the properties of numbers reflect properties of attributes in the real world. Fenton and Pfleeger [42] observe that the only possible measurement mapping is the actual count in the entities. For example, the number of failures observed during an integration test can be measured only by counting the failures. The only admissible transformation is the identity transformation.

5.1.2 Metric validation

There are two aspects of validating a software metric, theoretical and empirical. Theoretical validation amounts to proving analytically that the metric conforms to a set of formal criteria. Weyuker's set of measurement principles [102] and the properties of Briand et al. for testing the usefulness of coupling metrics [25] are examples of criteria with which proposed metrics can be evaluated. Validating a software metric experimentally is confined to the set of assumptions relating the measure of an internal property to an externally visible attribute. Observations made about the external property of the system during its execution can be gathered and co-related with an internal property defined by the metric. The assessment of this co-relation should support or refute the assumption made about the predictive power of the metric.

5.2 A Dynamic Coupling Metric

According to the ontological model of Wand and Weber [99], itself based on Bunge's work [27, 28], two objects are coupled if either one of them can influence the history of the other. The history of an object is defined as the sequence of its states in time. Objects are representations of things; the attributes of a thing are represented as attributes of the object and permitted state transformations as the result of operations on the object. The state is defined as the set of values of the object's attributes.

A coupling measure between two arbitrary objects, P_0 and P_1, thus depends on the time during which P_0 influences the history of P_1 or vice-versa. A coupling measure also depends on the number of objects involved and on their complexity. Here, complexity is due to the underlying class relationships. An object P is the entity and the measure is the object's coupling during a time period Δt, defined as the sum over all program execution steps and the sum over all objects, O_i, coupled to P:

$$\mu(P)_{|\Delta t} = \sum_j \sum_i f_i(t_j) g_p(|O_i|)$$

Here, $i = 0, 1, 2, ...$, number of coupled objects, $\sum_i$ is the sum over the set of objects coupled to P, Δt is an ordered sequence of program execution steps $< t_0, .., t_j, .., t_n >$ and $\sum_j$ is the sum over the program execution steps; $f_i(t_j)$ assumes values 1 or 0 depending on whether coupling of the i^{th} object is active at t_j or not and $g_p(|O_i|)$ denotes the complexity measure (due to class coupling) of the i^{th} object coupled to P.

Note that the sequence of program execution steps is *not* identical to the sequence of program statements, but is the sequence of events representing object interactions during program execution. According to the OO paradigm, objects interact by passing messages such as object creation or method invocation. Depending on the underlying class relationships, a single statement such as object creation may induce a chain of events leading to a number of objects being created.

When an object P is created at some time t (i.e., due to an event during program execution. t=0 corresponds to the start of program execution.), the default value of $\mu(P)$ is determined by the (static) class structure. Henceforward, we refer to our metric as simply *DCM* (Dynamic Coupling Metric) and to the time-independent factor, $\sum_i g_p(|O_i|)$, as the contribution due to the complexity measures of the coupled objects.

In the ontological model of Wand and Weber, a system is a set of things (objects) in which every object is coupled to at least one other object in the set. At the system level, the coupling measure in a time interval Δt is the sum of all measures defined over all the objects of the system, i.e.,

$$DCM(system)_{|\Delta t} = \mu(system)_{|\Delta t} = \sum_{P}^{all\ system\ objects} \mu(P)$$

The term $g_p(|O_i|)$ represents the *static* part of the DCM and is determined by the class relation between P and O_i. The form of $g_p(|O_i|)$ depends, as observed by Briand et al. [25], on the goal of the measurement. The goal is determined by the external attribute or quality we associate with our measure. For example, Briand et al. [26] showed that an inheritance-based coupling measure is not appropriate for predicting fault-proneness of classes, whereas a non-inheritance-based measure is. Therefore, for the fault-proneness as external property, it is inappropriate to assign metric values to $g_p(|O_i|)$ such as DIT (Depth of Inheritance Tree) and NOC (Number Of Children) of Chidamber and Kemerer [35]. Furthermore, if we are interested in comparing reflective systems to non-reflective systems exhibiting the same behaviour with regard to coupling, $\sum_i g_p(|O_i|)$ can be used to express the complexity of class coupling in the two systems. To this end, it is sufficient to assign $g_p(|O_i|)$ a real number counting the number of classes which class of P uses or depends on; for example DAC (Data Abstraction Coupling) [73] or CBO (Coupling Between Objects) [35] metric values. In summary, $g_p(|O_i|)$ is determined by the goal of the measurement and, in particular cases, requires knowledge of class relationships.

The factor $f_i(t_j)$ is calculated on a step-by-step basis and requires knowledge of objects' coupling scales (transient or static. See next section on DCM scales). If a meta-object O_i is coupled to P at t_j and decoupled immediately after, the contributions to $DCM(P)$ due $\sum_i g_p(|O_i|)$ are not counted from t_{j+1} onwards. In Java, the coupling scale of objects (base and meta-objects) can be identified according to their class types. A DCM measuring program should be able to identify objects by reading data files provided at the start of the program.

A program execution state is the collection of its objects' states. If a program consists of one object, then the coupling measure is zero. On the other hand, if, at an execution step t_j, the coupling of an object O_i can be switched off, the contribution of this object to $DCM(P)$ is set to zero until the coupling is reactivated again. We follow object behaviour during a limited period of time consisting of a sequence of program execution steps and measure the coupling of the object at each step. The contribution to $DCM(P)$ due to the complexity measures of the coupled objects ($\sum_i g_p(|O_i|)$ factor) may remain constant and retain its default value (determined by the class structure) during the entire lifetime of P. This contribution may, however, vary in time in the case where the program is allowed to change its behaviour and/or structure due to reflective mechanisms provided by the implementation language. In the particular case where a system can be described solely in terms of static class relationships, $f_i(t_j) = 1 \; \forall \; t_j$, the contributions to $DCM(P)$ are determined entirely in terms of class relationships.

A number of metrics in the literature define coupling, cohesion and complexity by referring to the static class structure. Coupling metrics reflect dependencies among classes including inheritance, aggregation and usage class relationships (see Section 5.2.2 for a discussion on class relationship and object coupling). Cohesion metrics indicate whether a class represents a single abstraction or multiple abstractions. For example, the metric LCOM (Lack of Cohesion in Methods) of Chidamber and Kemerer reflect cohesion of a class by counting the number of method pairs that share instance variables against the count of those method pairs that do not share such variables. Complexity metrics give us indications about the complexity of a class, for example, by counting its Lines of Code (LOC) or Number of Methods (NOM) [59].

The most notable of *static* coupling metrics are DAC [73], CBO [35] and MPC (Message-Passing Coupling) [73, 75]. If we classify inheritance as a coupling relationship, we can add to the list the inheritance related metrics of Chidamber and Kemerer, DIT and NOC. None of these measures, refer to coupling induced at runtime between objects due to reflective mechanisms. DAC measures coupling that results from declaring classes as attributes in another class, i.e., coupling due to aggregation. DAC counts the number of abstract data types, i.e., classes, defined in a class. CBO of a class is a count of the number of other classes to which it is coupled. In the definition of CBO, inheritance is not considered as a form of coupling between classes. Still, CBO includes DAC, since it also counts the classes defined as attributes. MPC measures the dependency of a class on methods of other classes. MPC counts the number of send statements, i.e., method calls on objects of other classes from the methods of a class. DIT counts the number of ancestor classes of a class and NOC represents the number of immediate sub-classes in the class hierarchy.

5.2.1 DCM scales

We use the time dependency of DCM to define an ordinal scale between different types of couplings. Where coupling between a pair of objects (P, O_i) holds during the whole lifetime of P, we refer to the coupling as *strong* or *static*, otherwise we refer to it as *loose* or *transient*. In the case of object composition, the aggregate O_i influences the history of its container P as long as the container exists. In this case, the coupling is static. On the other hand, if P is a base object and O_i represents a set of meta-objects, the latter can influence the former only as long as the binding relationship holds. In addition, the state of the meta-object is not contained in, and consequently cannot affect, the state of the base object when the binding between both objects is switched off. In this case, the coupling is transient.

Boolean statements describing the relationships between the measures of different objects are invariant under transformations of the form: $DCM(P) \rightarrow$

$a\ DCM(P)$ with a being a positive real number. This implies that DCM is at least on the ratio scale.

5.2.2 Class dependencies and object coupling

In this section, the connection between object couplings and the underlying class couplings will be investigated. The basis of this investigation is the *instantiation* relationship which exists between classes and objects, according to the OO paradigm. We restrict ourselves to classes and objects of so-called 2-Level object model adopted by most OO languages, such as C++ and Java. Here, classes represent template behaviour of their instance and are strictly compile-time entities. This restriction does not, however, exclude Smalltalk-80 systems, since, although the language object model is 3-Level, meta-classes are anonymous and cannot be defined explicitly by the user. In practice, this means there are no meta-classes and the dominating instantiation relationship is a class-object relation.

Having defined object coupling types as elements of an ordinal scale of the DCM, we now discuss the interconnection between object coupling types and OO class relationships. The DCM metric defines two types of couplings between objects; static and transient. Static object coupling refers to couplings that persist as long as the participating objects remain accessible during program execution. Transient object coupling, on the other hand, is more flexible and can be switched on/off during program execution.

Most object interactions are determined by their class relationships. There is a connection between object coupling types as just mentioned and the underlying class relationships. Class relationships imply dependencies between corresponding objects' instances created at runtime. Independent of the class relationship inducing the coupling, dependency between objects ceases when the dependent object is destroyed; this can occur either explicitly by invoking an object's destructor, as in C++, or implicitly by invoking a garbage collector, as in Java and Smalltalk-80.

Class aggregation (has-a) and class inheritance (is-a) are two examples of class relationships that induce static coupling between objects. In the case of class aggregation, the relationship of inner classes to their containers is ignored, because it is purely a containment relationship at the class level. For our purposes, class aggregation that implies object composition (has-a relationship at the instance level) is relevant. Classes are defined independently of each other and the container class holds other classes (aggregates) as attributes. As far as our object coupling type classification is concerned, inheritance resembles class aggregation; inheritance induces static coupling between a child object and its parent.

In addition to inheritance and aggregation there are class relationships in which classes use other classes (uses-a) to implement their functionality. For example, a class `A` uses a class `B` as a method parameter or as a return type. Alternatively, `B` is needed to implement the logic of a method and is used as a local variable in a method of `A`. An instance of the class `A` does not include instances of `B` as part of its state. This is the main difference between the (uses-a) relationship and the other two class relationships. The instances of used classes do not constitute a part of the state of the instance that uses them. The uses-a class relationship thus induces transient coupling at the object level.

The coupling relationship between application objects and meta-objects of a reflective system is transient. In strongly typed object-oriented languages, subtype polymorphism may be used to implement meta-object classes in a generic manner, thus eliminating explicit compile-time class dependencies. This makes the coupling between objects at the base level and those at the meta-level a purely runtime issue. In reflective systems, base objects are bound to meta-objects at runtime and decoupled from them after the changes made at the meta-level are reflected back to the base level.

5.3 Theoretical Validation of DCM

In this section, the theoretical validation of DCM against Weyuker's set of measurement principles [102] and properties of Briand et al. for justifying coupling measures [25] is addressed. In both cases, we begin by presenting the principles in the same notation as expressed by the authors and consequently analyse the metric with respect to each entry in each set.

We recall that the DCM metric maps objects into real numbers where the numbers characterise object coupling. The sets of Weyuker and Briand et al. require the concept of concatenation between real world entities which are, in our case, runtime objects, to be defined. In the following, we define *object combination* in terms of the combination of their constituent classes. In the same manner, permutation of elements within the entities being measured, as required in Weyuker's seventh axiom, will be applied to classes whose objects constitute the real world.

Weyuker's axioms represent a formal set of criteria with which we can evaluate software metrics. Many authors have criticized the axioms for not being founded on a consistent view of complexity [42], for being inconsistent with the principles of scaling [106] and for being only necessary but not sufficient conditions for good complexity metrics [33]. However, Weyuker's set for evaluating software metrics is considered to be a well-defined set of properties which constitute a formal analytical approach and provide a language for evaluation of metrics [35, 49]. In the same way, the set of properties proposed by Briand et al. provide a formal approach for evaluating coupling metrics. The latter set, constitutes, apart from the non-negativity property of coupling metrics, a subset of Weyuker's set of principles.

5.3.1 Weyuker's set of measurement principles

The set of axioms suggested by Weyuker are intended to aid in the evaluation of complexity metrics. We use the same language employed by Weyuker to express the axioms. The language includes arithmetic and boolean operators expressing operations and relationships among real numbers used in the usual manner. In addition, there is a concatenation operator acting on a set of program entities which we denote by $\oplus$. The object of study is a mapping $||$ from the set of programs to the set of real numbers. The mapping reflects certain qualities or attributes intrinsic to the entities; for example, the complexity inherent in a program. The mapping is an abstraction of software metrics which assign real numbers to software systems entities (representing the real world). In the OO world, entities are classes, objects or an entire program and attributes of the entities include coupling, level of code reuse, fault proneness etc.

Following Weyuker [102], let S denote a set of programs and $p_0, p_1 \epsilon S$. The expression $p_0 \equiv p_1$, represents program equivalence and means that the two programs have the same output values on the same input. The expression $p_0 = p_1$ means that the two programs are syntactically identical.

- Property 1: $(\exists\, p_0 \epsilon S)(\exists\, p_1 \epsilon S)(|p_0| \neq |p_1|)$. Property 1 excludes metrics which give all program entities the the same value.
- Property 2: For a nonnegative number c, there are only a finite number of programs of that complexity.
- Property 3:$(\exists\ p_0\ \epsilon\ S)(\exists\ p_1\ \epsilon\ S)(p_0 \neq p_1\ \mathit{and}\ |p_0| = |p_1|)$. Property 3 asserts that there are distinct programs having the same complexity.
- Property 4: $(\exists\ p_0\ \epsilon\ S)(\exists\ p_1\ \epsilon\ S)(p_0 \equiv p_1\ \mathit{and}\ |p_0| \neq |p_1|)$. Property 4 suggests that a different program implementing the same functionality can have different complexity.
- Property 5: $(\forall\ p_0\ \epsilon\ S), (\forall\ p_1\ \epsilon\ S)(|p_0| \leq |p_0 \oplus p_1|\ \mathit{and}\ |p_1| \leq |p_0 \oplus p_1|)$. Property 5 requires monotonicity of the complexity metric. It states that the complexity of the sum of two parts should be at least as great as the complexity of either part.
- Property 6: $(\exists\ p_0\ \epsilon\ S)(\exists\ p_1\ \epsilon\ S)(\exists\ p_2\ \epsilon\ S)(|p_0| = |p_1|\ \mathit{and}\ |p_0 \oplus p_2| \neq |p_1 \oplus p_2|)$. Property 6 assures that the concatenations of code segment p_0 and p_1 of the same complexity, with a third code segment p_2, may result in program entities having different complexity measures. The same applies to concatenations being performed in the opposite order, i.e., the p_2 code added at the beginnings of p_0 and p_1 instead of the ends.
- Property 7: $(\exists\ p_0\ \epsilon\ S)(\exists\ p_1\ \epsilon\ S)$ such that p_1 is formed by permuting the statements of p_0 and $|p_0| \neq |p_1|$. Property 7 states that changing the order of the statements can change the complexity.

- Property 8: For programs $p_0 \,\epsilon\, S$ and $p_1 \,\epsilon\, S$, if p_1 is a renaming of p_0 then $|p_0| = |p_1|$. Property 8 states that changing variable names in a program does not change its complexity.

- Property 9: $(\exists\, p_0 \,\epsilon\, S)(\exists\, p_1 \,\epsilon\, S)(|p_0| + |p_1|) < |p_0 \oplus p_1|$. Property 9 states that the concatenation of two program entities may result in a program more complex than the sum of the complexities of its constituents.

5.3.2 Applying Weyuker's principles on *DCM*

Weyuker's set of measurement principles can be applied to OO metrics where the complexity mapping || is considered as a mapping from the set of classes or objects to R. In our case, the complexity mapping is identified with the *DCM* metric defined in Section 5.2. Here, the set of programs S refers to the domain of the metric, namely, the set of objects which constitutes the OO program under consideration. In addition, we need to adapt the concatenation operator $\oplus$. The ontological model of Wand and Weber [99], based on Bunge's work, does not provide a definition of the notion of aggregation of two or more things. Bunge's ontology, however, provides a basis for defining the combination of classes. As observed by Chidamber and Kemerer [35], from their principle of additive aggregation of two (or more) things, the combination of two object classes results in another class whose properties are the union of the properties of the component classes. Thus, the concatenation operator $\oplus$ can be defined on a basis provided by Bunge's ontology. The expression $\mu(P_0 \oplus P_1)$ refers to the *DCM* metric value of an object of a class obtained by concatenating the classes of the objects P_0 and P_1.

We assume that two classes can have a number of "identical" methods and/or fields. Such methods and attributes become redundant through a combination of their classes. In each case only one copy of a method and/or an attribute is considered. We further assume that an object of a combination class can replace the objects of the constituent classes without affecting the overall behaviour of the enclosing program. We shall use these assumptions when considering Weyuker's properties 5, 6 and 9. Each item in the following list validates our metric against the corresponding Weyuker's criterion. Further, $\mu(P) \,\epsilon\, R$ refers to the metric function DCM of an object entity P mapped into the set of real numbers.

1. Weyuker's first property states that not every object can have the same metric value. This property is satisfied by the *DCM* metric if we assume (in common with Chidamber and Kemerer) that the number of couplings of a given object at any stage of program execution is a discrete random variable characterised by some general distribution function; all such numbers are independent and identically distributed. Thus, object couplings follow a statistical distribution not apparent to an observer of the system.

This assumption is similar to that made by Chidamber and Kemerer on number of methods of a given class, number of instance variables used by a method etc. Following this assumption, object couplings (DCM values) of two different objects P_0 and P_1 are independent and identically distributed which implies that $\mu(P_0) \neq \mu(P_1)$.

2. Weyuker's second property states that a metric should assign the same value only for a finite number of objects. Since the domains of the metric constitute finite OO executable systems, each of which contains a finite set objects, this property will be met by any dynamic object metric, including DCM.

3. Weyuker's third property states that different objects might have the same metric value. Here, the objects can be instances of the same class or of different classes. DCM is independent of the object identity; it is defined in terms of number of coupled objects, their class coupling complexity ($g_p(|O_i|)$) and the nature of coupling ($f_i(t_j)$ factor) over a certain number of program execution steps. Property 3 is thus satisfied.

4. Weyuker's fourth property states that different objects exhibiting the same behaviour (their classes implementing the same functionality) might have different metric values. Here, the same argument applies as in the case of the third property. The DCM value of an object, though dependent on the functionality of its class, it is not determined by it. The contribution to the metric through the class functionality is due only to one factor in DCM, namely, $g_p(|O_i|)$. Thus, property 4 is satisfied.

5. Weyuker's fifth property states that the metric value of an object of a combination of classes should be at least as great as the DCM value of either objects of the constituent classes.

   Let P_0 and P_1 denote two objects whose classes are combined. The assumption made about class combination implies that the combined class has less methods and/or less fields but provides the same functionality. It follows that *at any execution step*, objects of the combined class have less object interactions when used instead of objects of the constituent classes. Therefore,

   $\forall\ t_j\ \epsilon\ \{set\ of\ execution\ steps\}$

$$f_i(t_j)\sum_i g_{p0\oplus p1}(|O_i|) \leq f_i(t_j)(\sum_i g_{p0}(|O_i|) + \sum_i g_{p1}(|O_i|))$$

   In this expression, two equivalent object systems are involved. The left-hand side of boolean operator ($\leq$) corresponds to a program containing objects of classes obtained by combining classes of a second system corresponding to the right-hand side.

We note that $f_i(t_j)$ factors (which express the type of object coupling) do not change in the system containing objects of combined classes; they keep their values as in the original system with constituent objects. Therefore, by summing over all t_j, we get:

$$\mu(P_0 \oplus P_1) = \mu(P_0) + \mu(P_1) - \delta \text{...................} (A5)$$

where δ is the number of object couplings reduced due to combination of the classes of P_0 and P_1. Since reductions can not be greater than the original number of object interactions, we have: $\mu(P_0) - \delta \geq 0$, and $\mu(P_1) - \delta \geq 0$. It follows that $\mu(P_0 \oplus P_1) \geq \mu(P_0)$ and $\mu(P_0 \oplus P_1) \geq \mu(P_1)$. Thus, property 5 is satisfied.

6. Weyuker's sixth property states that given two objects having equal metric values, the metric values of their concatenations with a third object can be different.

   Let P_0 and P_1 denote two different objects with $\mu(P_0) = \mu(P_1)$. Consider a third object P_2 together with two other objects whose classes are obtained by combining each of the classes of P_0 and P_1 with the P_2 class. Using formula $A5$, we have:

$$\mu(P_0 \oplus P_2) = \mu(P_0) + \mu(P_2) - \delta$$

$$\mu(P_1 \oplus P_2) = \mu(P_1) + \mu(P_2) - \eta$$

   Since δ and η are independent of the metric values of P_0 and P_1, they can be different. Therefore, the right-hand side of the preceeding formulas need not be equal. It follows that:

$$\mu(P_0 \oplus P_2) \neq \mu(P_1 \oplus P_2)$$

   Thus, property 6 is satisfied.

7. Weyuker's seventh property requires that a permutation of elements within an object should change the metric value of that object. As noted in the introduction of this section, we consider such permutations as being applied to the classes of the objects involved. Objects interact by sending messages realised as method calls. Permuting method statements may change a method's semantics and consequently the behaviour of objects concerned. As a result, DCM values of objects change according to modifications made to the objects they are coupled to at arbitrary execution steps. Thus, property 7 is satisfied.

8. According to Weyuker's eighth property, metric values should not change with the change of object names. DCM is defined independently of ob-

jects' names and depends only on the nature of their interactions and class relationships. Consequently, this property is satisfied by DCM.

9. Weyuker's ninth property states that the metric value of an object of a combination of classes is greater than the sum of the DCM values of objects of the constituent classes.

   Following the same analysis of property 5, we get by using formula $(A5)$:

   $$\mu(P_0 \oplus P_1) \leq \mu(P_0) + \mu(P_1)\text{.........................}(A9)$$

   Weyuker's property 9 is not satisfied by DCM, i.e., a combination of classes does not increase a program's object interactions measured by the metric; on the contrary, it reduces those interactions. Failing to meet the ninth property shows that DCM exhibits the same behaviour as the metrics of Chidamber and Kemerer; some of which (CBO, DIT and NOC) are classified as static coupling metrics. We conclude that, in designing software, minimizing the number of classes decreases redundancy and as a result reduces object coupling. Notice that, in the limit where all system classes are combined to form a single class, DCM is zero, i.e.,

   $$\mu(system\ of\ single\ object) = 0\text{.........................}(A9')$$

Properties 5, 6 and 9 deal with objects of class combinations. In validating DCM against these properties, we assumed that at every program execution step, the set of objects coupled to the object entity being measured (P) is reduced to a subset. Let P_0 and P_1 denote the objects of classes combined to produce the class of P. If $S_{t_j,P}$ denotes the set of objects coupled to P at some execution step t_j, S_{t_j,P_0} and S_{t_j,P_1} being the sets of objects coupled to P_0 and P_1, respectively, then our assumption reads as:

$$\forall\ t_j \ \epsilon\ \{set\ of\ execution\ steps\}\ \ S_{t_j,p} \subseteq S_{t_j,p_0} \bigcup S_{t_j,p_1}$$

We further assume that the types of object coupling expressed in terms of the step function $f_i(t_j)$ do not change as a result of class combination. This means the set of object relationships between P and $S_{t_j,P}$ over all the execution steps, denoted by R_{S_P}, is a subset of the union of corresponding sets of relationships of P_0 and P_1. If the two latter sets are denoted by $R_{S_{P_0}}$ and $R_{S_{P_1}}$, we can express our assumption as:

$$R_{S_p} \subseteq R_{S_{p_0}} \bigcup R_{S_{p_1}}$$

If these two assumption hold, DCM satisfies properties 5, 6 and invalidates property 9 *over the entire program.*

5.3.3 The coupling measures criteria of Briand et al.

Briand et al. [25] define five properties for testing the usefulness of coupling metrics. These properties represent necessary but not sufficient conditions for justifying a coupling measure and were applied for testing various class level metrics.

The notation of Briand et al. will be used to express the properties, where c denotes a class, C denotes an OO system considered as a collection of classes, $OuterR(c)$ denotes the relevant set of relationships from or to a class c, and finally, $InterR(C) = \bigcup_{c \,\epsilon\, C} OuterR(c)$ denotes the set of all class coupling. Briand et al. distinguish between used classes and the using classes. Without loss of generality, we can drop the direction of dependency and refer to $OuterR(c)$ as the set of classes coupled to c. Hereafter, *Coupling* denotes the metric whose validity is investigated.

1. Nonnegativity: The coupling of a class c or of an object-oriented system C is nonnegative.

$$Coupling(c) \geq 0 \; or \; Coupling(C) \geq 0$$

2. Null value: The coupling of a class c or of an OO system C is null if $OuterR(c)$ or $InterR(C)$ is empty.

$$OuterR(c) = \phi \Rightarrow Coupling(c) = 0 \; or$$

$$InterR(C) = \phi \Rightarrow Coupling(C) = 0$$

3. Monotonicity: Let C be an OO system and $c \,\epsilon\, C$ be a class in C. Assume that class c is modified to form a new class c' which is identical to c except that $OuterR(c) \subseteq OuterR(c')$, i.e., some relationships were added to c'. Let C' be the object-oriented system identical to C except that class c is replaced by class c'. It follows:

$$Coupling(c) \leq Coupling(c') \; or$$

$$Coupling(C) \leq Coupling(C')$$

4. Merging of classes: Let C be an OO system, and $c_0, c_1 \,\epsilon\, C$ two classes in C. Let c' be the class which is the union of c_0 and c_1. Let C' be the OO system identical to C except that classes c_0 and c_1 are replaced by c'. Then

$$Coupling(c_0) + Coupling(c_1) \geq Coupling(c') \; or$$

$$Coupling(C) \geq Coupling(C')$$

5. Merging of unconnected classes: Let C be an OO system, and $c_0, c_1 \epsilon C$ be two classes in C. Let c' be the class which is the union of c_0 and c_1. Let C' be the OO system which is identical to C except that classes c_0 and c_1 are replaced by c'. If no relationships exist between classes c_0 and c_1 in C, then:

$$Coupling(c_0) + Coupling(c_1) = Coupling(c') \;\; or$$

$$Coupling(C) = Coupling(C')$$

Applying the criteria of Briand et al. on DCM requires modification of their semantics to adapt them to the DCM domain (where entities are objects and not classes). We proceed as in the case of Weyuker's principles and consider each property separately.

i. For the nonnegativity, no term in the analytical expression of DCM is negative. The measure is always greater than zero and the nonnegativity property is satisfied.

ii. In the case of a system consisting only of one object, the set of coupled objects is empty. Hence, the factor $\sum_i g_p(|O_i|)$ is zero and DCM value is therefore null (See A9' in the previous section).

iii. The monotonicity property is identical to that of Weyuker's fifth axiom. In the previous section, we showed that DCM satisfied this property.

iv. Merging of classes property corresponds to Weyuker's ninth axiom. In analysing the DCM validation of Weyuker's ninth axiom, we have shown that for a combination of classes (c_0 and c_1), the DCM values are related as in A9, namely, $\mu(c_0 \oplus c_1) \leq \mu(c_0) + \mu(c_1)$. Here, $c_0 \oplus c_1$ expresses class merging of c_0 and c_1. Thus the property of Briand et al. with respect to class merging is satisfied.

v. The merging of unconnected classes property is a special case of the previous property where $OuterR(c_0) \bigcap OuterR(c_1) = \phi$. In this case, the classes do not share common methods or common fields. For the DCM metric, we have $\mu(c_0 \oplus c_1) = \mu(c_0) + \mu(c_1)$.

The properties of Briand et al. of class coupling metrics can be adapted to dynamic object metrics with minor changes in the semantic of merging of metric domain entities. Monotonicity and merging properties correspond to similar properties in Weyuker's set of principles. In the next section we present an empirical validation of the DCM which will allow us to demonstrate the practical application of the metric.

5.4 On the Empirical Validation of the DCM

In the context of empirical validation, we present a measuring tool and show that DCM can be measured automatically. A case study is conducted and the tool is used to collect object coupling data at runtime. The motivation behind this study is investigating the hypothesis that reflective systems show less runtime coupling when compared to equivalent static systems exhibiting the same behaviour.

5.4.1 Measuring tool

One way to collect coupling data of interacting objects is to intercept the message exchanges between them. In C++ and Java, where objects are the only runtime entities, message exchange is realized as method invocation on objects. In these languages, programming constructs (loops, conditional statements, assignments, etc.) are used to control interaction between objects but themselves take no part in it. For C++ and Java systems, the programme execution steps relevant to coupling are object instantiations and method invocations. In Smalltalk-80, language constructs themselves represent objects and applying them amounts to sending messages with particular parameters. If we neglect the meta-class instantiations carried out independently of the user by the Virtual Machine (VM) and if we consider only objects corresponding to class instantiation, then object interaction can be treated in the same way as in C++ and Java.

5.4.1.1 Tool implementation

A trivial implementation of the idea of intercepting objects' message exchanges is to modify the existing code by inserting intercepting code at appropriate places. Modifying applications implies extra costs of testing and maintenance.

There are different ways to avoid the problems of changing existing code. One alternative is to customise the behaviour of the underlying VM. Arisholm et al. [14], for example, let the JVM load a library of routines for data collection every time object interactions take place. A second alternative is use reflective capabilities of the programming language. Arisholm [13] uses specific features provided by the IBM-VisualWorks Smalltalk programming environment and lets "shadow objects" intercept all messages sent and received at runtime.

To avoid changing JVM behaviour and using incompatible language features, we developed, using AOP techniques, the interceptive code as an independent programming unit and merged it with the target programme using a weaving tool. AOP supports the principle of separation of concerns through a wide range of tools. There are AOP language extensions for Java, C++, Smalltalk-80 and prototypes for other languages are also available [3].

In AOP, intercepting code could be realised as an aspect or a concern whose code cross-cuts that of the system's core concerns. For Java programs, we used AspectJ to implement intercepting code which was then inserted at the appropriate join points, i.e., at those places in the sample programme where objects are created or exchange messages. AspectJ is an extension of Java with new language constructs such as pointcuts, advices and aspects that allow explicit implementation of the principle of separation of concerns.

Listing 5.1 provides the key features of the interceptive code implemented in AspectJ. We note that the `Interceptor` aspect makes no reference to application programs and is abstract (line 4). Abstract aspects cannot be merged and must be specialized (through the extends mechanism) before their code can be weaved into the programme code. Two abstract pointcuts with no parameters are defined corresponding to objects' creation and methods' invocation (lines 5 - 6). A third pointcut (line 7) is defined to invoke data collection advice code. Data collected throughout programme execution is manipulated using this advice (lines 57). In extending the aspect, i.e., defining concrete aspects that apply in certain contexts, the join points at which the corresponding advice code is executed must be specified. Join points are specified using pointcut designators (or events) such as calling a constructor or invocation of a method. The code to be executed when an object is created or when a method is invoked is implemented as an advice. Advices can be invoked before, instead of or after join points are reached. Aspect extension languages provide APIs allowing the invocation of the target method from within the around-advice.

A number of hash tables, lists and a variable are defined for manipulating object data and for counting the execution steps (lines 8 - 16). One key feature of this manipulation is getting the identity of an object set by the system at runtime using the method `thisJoinPoint.getThis()`. This expression is transformed into a string and used to identify the object.

Object creation is manipulated in the around-advice `myConstructor()`. There is an after-advice with the same name; its task is to increment the execution step counter. When an object is created, the `Interceptor` reads its $\sum_i g_p(|O_i|)$ value and a digit (0 for static and 1 for transient) from configuration files. The digit represented the coupling scale to which the object belongs to. Using AspectJ reflective capabilities, the object type is derived as well as its unique identity. Each newly created object is assigned a record of three integer values ($\sum_i g_p(|O_i|)$, exec-step, coupling scale) and saved in a hash table with its ID being the key. All created objects are saved together with their records for later use (line 31). Executions steps, represented as integer values, are incremented *after* every constructor and method is called. The dependency of object DCM values on the number of execution steps is realised in the `Interceptor` aspect implementation by saving the execution step value as a key together with the object identity and DCM value (hash table `exec_ht`, line 16). In addition, the class types of the created objects are also saved in a list for the final counting.

Method calls on objects are handled by the around-advice `execsTest()`. With each execution step corresponding to a message exchange, the measures of all existing objects are updated depending on the coupling scale (lines 46 - 47 and 53). Again, the reflective interface of AspectJ allow us to identify the object and by using the hash table containing the objects and their records (`objects_ht`), `Interceptor` can identify the object and update its record including its DCM value. When a message exchange results in a chain of objects' interactions, the DCM value of the leading object (initiating the series of calls) is incremented by the contributions of objects involved and needed to fulfill the leading object's task (lines 43, 48 - 49). This is achieved by keeping a list of the objects involved in the chain interaction. The list is populated by an automatic recursive call caused by calling the method `proceed()`. Recall that an around-advice traps the execution of the original method representing the join point and that the original action can be recovered by calling `proceed()` from within the advice. As in the case of object creation, the execution step counter is incremented every time an object exchanges messages with other objects.

5.4.1.2 Data Collection Process

Recall that the goal of our measurement is to compare two systems with regard to their object coupling. In this case, it is sufficient to assign for each $g_p(|O_i|)$ a value representing the number of classes which the class of P uses or depends on. The $g_p(|O_i|)$ values of all class types can be read from the system's class diagrams. They can be stored in configuration files and read during programme execution. The files can be expressed in XML or any other suitable text format.

To the outside world the two systems behave in the same way. Their designs, however, are different. This difference reflects itself in varying $g_p(|O_i|)$ values and a different internal object behaviour. It also reflects itself in the different values of $f_i(t_j)$ factors which we associate with objects upon their creation depending on type of their classes. For example, objects of type `InvocationHandler` are assigned value 0, whereas application objects are assigned the value 1.

To apply the `Interceptor` aspect in a certain context, we had to extend the aspect (i.e., make it concrete) and specify the join points. Invoking the AspectJ weaving tool inserted the corresponding advices' code at the specified points in the application programme. Listing 5.2 shows an example of a concrete aspect that allows the application of the abstract `Interceptor` aspect in an application named `TestClass` and involves objects of types `Class_1`, `Class_2`, etc. We assume that the application programme, before terminating, calls a method `collectData()` to allow for DCM data collection.

```
import org.aspectj.lang.*;                                              //1
import org.aspectj.lang.reflect.CodeSignature;                          //2
import java.util.*;                                                     //3
abstract aspect Interceptor {                                           //4
  abstract pointcut myConstructor();  // object creation event          //5
  abstract pointcut execsTest();      // object interactions            //6
  abstract pointcut dataCollection(); // for collecting data            //7
  // hash table for object records (objects IDs, (dcm, exec step, 1|0)) //8
  Hashtable objects_ht=new Hashtable();                                 //9
  // for class types(name,(total no of objects of type, total coupling val))
  Hashtable types_ht=new Hashtable();                                   //11
  Vector task_objects=new Vector(); // objs taking part in fulfilling a task
  Vector all_objects=new Vector();  // all created objects              //13
  Vector all_types=new Vector();    // all class types                  //14
  // holding data (key, value) as (exec step, (objects ID, coupling value))
  Hashtable exec_ht=new Hashtable();                                    //16
  private int exec_step=0;           // no. of execution steps          //17
  after(): myConstructor() { exec_step +=1; }                           //18
  void around(): myConstructor() {                                      //19
    Signature sig=thisJoinPoint.getStaticPart().getSignature();         //20
    String class_name=sig.getDeclaringType().getName();                 //..
    String the_object=""+thisJoinPoint.getThis();
    if (thisJoinPoint.getThis()!=null) all_objects.add(the_object);
    ...
    if (!object_found)  {
      // DCM_val_sm contains gp(|Oi|) and 0|1 separated by ";". 1 for
      // static and 0 for meta-object. DCM_val refers to Sum(gp(|Oi|))
      String DCM_val_sm=ClassRelationship.getParameter(class_name);
      // extract gp(|Oi|) value and the nature of the object.
      // define val as array of Integers:(gp(|Oi|), exec_step, Integer(0|1))
      objects_ht.put(the_object, val);                                  //31
      ...                                                               //32
    }                                                                   //33
  }                                                                     //34
  after(): execsTest() { exec_step +=1; }                               //35
  Object around(): execsTest() {                                        //36
     String the_object=""+thisJoinPoint.getThis();                      //37
     if (thisJoinPoint.getThis()!=null) task_objects.add(the_object);   //..
     Object result = proceed();
     ...
     Enumeration keys=objects_ht.keys();
     if (task_objects.size() > 1) {
       String oid=(String)task_objects.elementAt(0); // leading  object //43
       int leadObj_extra_value=0;    // due to Sum over all objects coupled
       while (keys.hasMoreElements()) {
         // if (object belongs to task set or is a static object)       //46
         // update its record by adding the default gp(|Oi|) value      //47
         // if (a task object) then update leadObj_extra_value as       //48
         leadObj_extra_value+=new Integer(<gp(|object|)>).intValue()    //49
         ...
       }
       if (task_objects.size()==1) { // no object task list             //52
        // update the coupling values of static objects                 //53
       }                                                                //54
     }
  }                                                                     //56
  void around(): dataCollection() {  // collect data ... }              //57
}                                                                       //58
```

Listing 5.1: Main features of the interceptive code

```
public aspect ConcreteInterceptor extends Interceptor {
   pointcut myConstructor():
                (within(TesClass)||within(Class_1)||..)&&execution(new(..));
   pointcut execsTest():
                  (within(TesClass)||within(Class_1)..)&&execution(* *(..));
   pointcut dataCollection():(within(TestClass))&&call(void collectData(..));
}
```

Listing 5.2: Concrete aspect for intercepting executions a concrete system

5.4.2 A case study

Consider two Java systems with different architectural characteristics providing the same functionality. Classes in the first model, henceforth referred to as the static system, are related through the known has-a and uses-a relationships and all the objects interact accordingly. In the second model, i.e., the meta-system, the class structure exhibits less coupling due to the dynamic nature of the coupling relationships. In the latter system, object interaction is more flexible and supported by polymorphism and reflective features of the Java programming language. Application objects are attached dynamically to reified meta-objects and decoupled from them after the reflection process is over, i.e., after changes made at the meta-level have been reflected back to the base level.

The structure of configuration files could be simplified and deployed as a list of pairs in the format (class name, $(\sum_i g_p(|O_i|); 0|1))$. The choice between 0 or 1 depends on the scale of object coupling (transient or static) an object can build. For systems implementing our reflection model, there are two alternatives: objects of type `InvocationHandler` including those of helper classes used at the meta-level are assigned value 0 and application objects are assigned the value 1.

Figure 5.1 depicts the class diagram of the static system in UML notation [10], showing only class names and their relationships. The configuration file corresponding to the model of Figure 5.1 contains four entries: (`A`, (2;1)), (`B`, (1;1)), (`C`, (1;1)) and (`D`, (0;1)); the configuration file was expressed as a Java properties file. In these entries, the numbers representing $\sum_i g_p(|O_i|)$ refer to the number of aggregates in each class. They correspond to the DAC (Data Abstraction Coupling) counts. We are interested in these counts, because they represent static coupling at the object level.

Table 5.1 shows the total number of objects of each of the class type in the model. The first column (pes) refers to the number of execution steps or the number of object interactions. The second, third and fourth columns show the total number of objects and the corresponding DCM value separated by commas for A, B and C class types, respectively. The fifth column shows the total number of D objects created at each stage of data collection. Since D is an

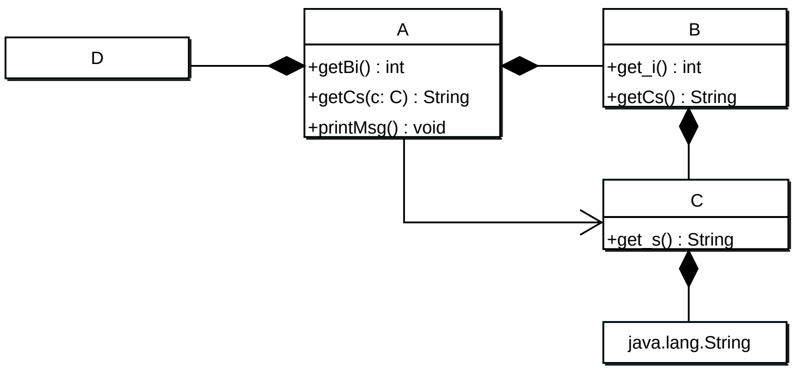

Figure 5.1: Class structure of the static system

independent object, DCM(`D`) remains zero throughout program execution and therefore not shown in the table. The objects column shows the total number of objects taking part at each stage and the DCM(system) gives the overall coupling value of all objects.

pes	`A`	μ(`A`)	`B`	μ(`B`)	`C`	μ(`C`)	`D`	objects	μ(sys)
1	0	0	0	0	0	0	0	0	0
10	1	9	1	5	1	5	2	5	19
100	2	356	2	179	2	179	3	9	714
200	2	702	3	436	5	607	4	14	1745
500	2	1902	3	1363	5	2107	4	14	5345
1000	102	12602	103	6786	105	7757	104	414	27145
2000	102	74052	253	60636	255	62407	104	714	197095
3000	202	166752	353	107586	355	110557	204	1114	384895
5000	222	194892	373	123576	375	130387	224	1194	448855
10000	222	214892	373	138576	375	155387	224	1194	508855

Table 5.1: DCM values as a function of the execution steps of the static model

Table 5.1 shows an increase in the DCM values and in the total number of objects as the pes-value increases. Increases in the individual DCM values depend on object's contribution to the interactions and is calculated according to the scheme described in the previous sections. To increase objects' interactions, new objects were created and methods invoked on them. In some cases, methods' invocations induced further invocations on other objects, resulting in a chain of objects' interactions. To produce the above results, different aspects of objects interactions were considered. These were:

- object instantiation,
- object instantiation through object composition,
- method invocations leading to a chain of objects' interactions and, finally,
- method invocation corresponding to simple client-target object interaction (no further invocations).

These aspects cover object coupling types discussed in Section 5.2.2. We note that coupling due to an is-a class relationship is considered identical to coupling due to a has-a relationship and that implicit inheritance relationships implied by the implementation language were not considered as part of our study.

The class structure of the meta-system is illustrated in Figure 5.2 where methods relevant to proxies' mechanisms of binding and method invocation are shown. It involves, in addition to the classes of the static system, classes that allow customization of objects' behaviour at runtime. Polymorphism and Java's dynamic proxies mechanism are used to implement the model. We arbitrarily chose to decouple `B` from `A` and made `B` part of `InvocationHandler` class at the meta-level. The resulting system exhibits the same functionality provided by the static model. The difference is that method invocation on `B` is realised through a proxy and the coupling between B's and A's is transient.

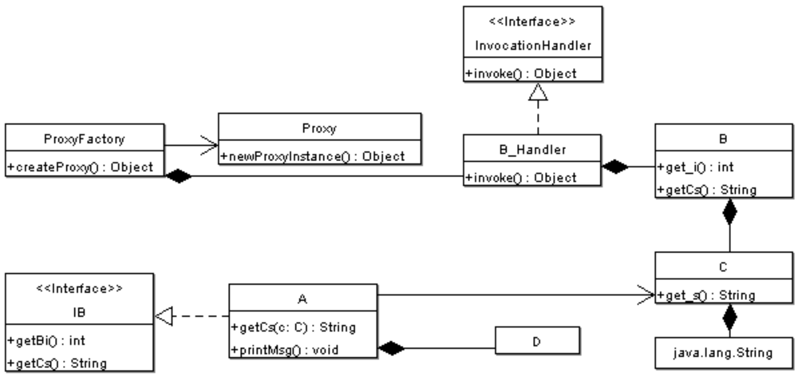

Figure 5.2: Class structure of the dynamic system

The configuration file corresponding to the model of Figure 5.2 contains two new classes when compared with the static case, i.e., `ProxyFactory` and `B_Handler`. The DCM default value of `A` changes and thus for this model we have the pairs: (`A`, (1;1)), (`B`, (1;0)), (`C`, (1;1)), (`D`, (0;1)), (`ProxyFactory`, (1;0)) and (`BC_Handler`, (1;0)). We note that the coupling scale of `B` has changed from 1 to 0.

Table 5.2 shows the DCM values and the number of objects present together with the overall DCM value of the meta-model system as they change with program execution steps. The numbers in the pes-column are slightly different from the corresponding numbers in the static case of Table 5.1. This is due to the difference in the set of objects involved and interactions needed to establish the connection between base objects (`A` and `D`) and objects at the meta-level (`ProxyFactory`, `B_Handler` and `B`). The same application code is applied in both models. In the meta case, calls on `A` and `B` instances are diverted to the meta-level. `ProxyFactory` assigned `A` objects to the meta-object of type `B_Handler`. We note that only one `ProxyFactory` object was created and used to bind the different base objects (of type `A`) and that only one meta-object was created and used to represent all these base objects. Furthermore, only one `B` object contained in `B_Handler` was created. DCM(`ProxyFactory`)=1 over all the executions steps, since this object does not take part in object interactions. The object counts of `ProxyFactory`, `B_Handler` and `B` as well as the DCM(`ProxyFactory`) value are all one (1) and will not be shown in the table.

pes	`A`	μ(`A`)	`C`	μ(`C`)	`D`	μ(`B`)	μ(`B_Ha..`)	objects	μ(sys)
1	0	0	0	0	0	0	0	0	0
15	1	6	1	6	2	2	2	7	17
105	2	182	1	92	3	3	3	9	281
201	2	360	2	267	4	5	5	11	638
508	3	1345	3	1257	5	6	6	14	1905
908	103	6795	3	1557	105	6	6	214	8365
2358	253	114195	3	3657	255	156	156	514	118165
3258	353	191145	3	5457	355	156	156	714	196915
5238	273	213135	3	11217	375	156	156	754	224665
10238	373	228135	3	26217	375	156	156	754	254665

Table 5.2: DCM values as a function of the execution steps of the meta-model

5.4.2.1 Comparison of Systems

From Table 5.1 and Table 5.2, we observe that the meta-model needs fewer objects for the same behaviour when compared with the static model. For example, in the static model (Table 5.1), the total number of objects and the DCM(system) corresponding to 10000 pes are 1194 and 508855, respectively, whereas in the meta-model the corresponding values are 754 and 254665. However, in cases where the communication between base and meta-level is intensified, the meta-system exhibits more interactions. This is the case when base level objects like `A` instances send messages (through the meta-object) to an object of type `B` asking for a `C` string. For example, the the pes values 2358, 3258, 5238 and 10238 in Table 5.2 are greater than the corresponding values in

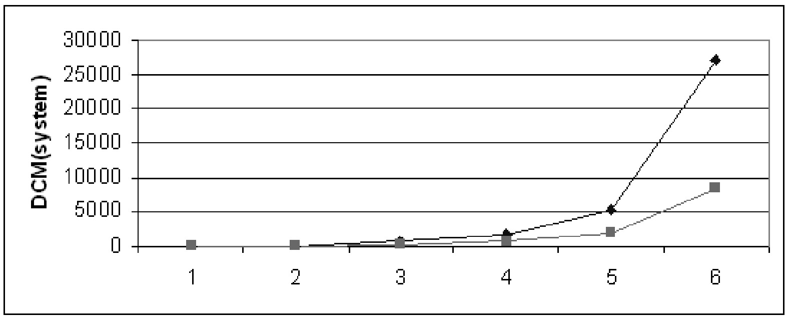

Figure 5.3: DCM(static system) vs DCM(meta-system) for 1-1000 steps

Table 5.1: 2000, 3000, 5000 and 10000. On the other hand, when the objects at base level (the `As`) do not interact intensively with the meta-level, the number of interactions in the meta-system is less than that in the static system. For example, compare the pes-value 908 with the corresponding 1000 in the static case. Here, most of the interactions took place at the base level.

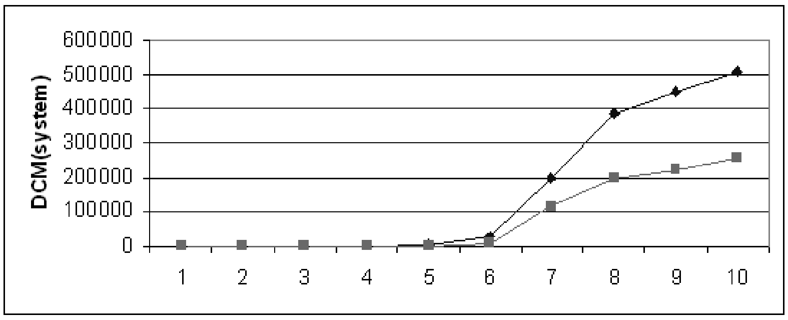

Figure 5.4: DCM(static system) vs DCM(meta-system) for 1-10000 steps

Figures 5.3 and 5.4 show graphical representations of the $DCM(system)$ values of both models as a function of the number of pes. In Figure 5.3, the lower range of execution steps (10 - 1000) is shown whereas in the second, the full range of our experiment (10 - 10000 interaction) is considered. With interactions fewer than 100, the difference in DCM values is negligible. As the number of interactions increases, DCM values of the static system grow faster than that of the meta-system. The total number of objects shows similar behaviour.

In conducting this study, the sample programs are implemented in such a way as to include all types of object coupling. We started with programs describing static systems and transformed these to programs exhibiting the same behaviour but using reflective mechanisms according to the reflection model proposed in eralier chapters. Direct interactions between `A` and `B` objects in the static case, in which the first asks the second to deliver data, is diverted to the meta-level. Here, a proxy is created and all `A` objects are assigned to a single `B_Handler` residing at the meta-level. The meta-object inspected the call and asked the contained `B` for the required data. The result is then sent back to the requesting object.

We conclude that reflective systems exhibiting the same behaviour as corresponding static systems have less coupling, fewer objects and more interactions. This is expected because of the dynamic nature of the coupling. Dynamic object coupling in reflective systems is transient. However, unlike the corresponding coupling arising from uses-a class relationships in static systems, the classes of the coupled objects are not related.

5.5 Discussion

One issue arising from the work in this chapter is the comparison of reflective techniques with classical OO techniques of inheritance and aggregation from object coupling perspective. In OOP, the two most common mechanisms for software reuse are aggregation and inheritance. In composing two classes, the container class uses aggregate services to extend or customize its own functionality. One of the known examples of reuse using the composition mechanism is the Adapter pattern of Gamma et al. [47]. The adapter object delegates method invocations to the adaptee to provide the client with the required functionality.

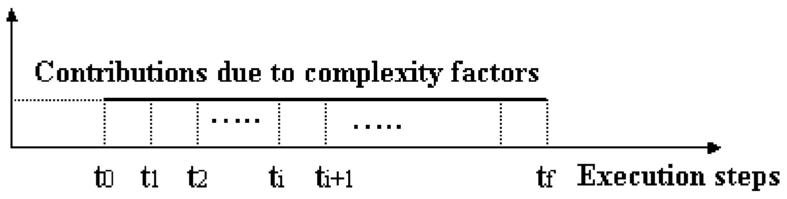

Figure 5.5: Static coupling holds a constant contribution to DCM over time

According to our classification of coupling, both composition and inheritance induce static coupling between objects. In case of composition, the aggregate is normally instantiated with the instantiation of the container object and holds as long as the container object holds. The same applies to inheritance. With the instantiation of a child object, a parent object is instantiated and the coupling is held during the entire lifetime of the child object. Figure 5.5 shows a representation of the contributions to DCM(P) corresponding to the complexity measures of objects coupled to a an object P, the $\sum_i g_p(|O_i|)$ factor. It is assumed that a hypothetical object P is coupled statically to other objects as a container (composition coupling) or as a child object (inheritance

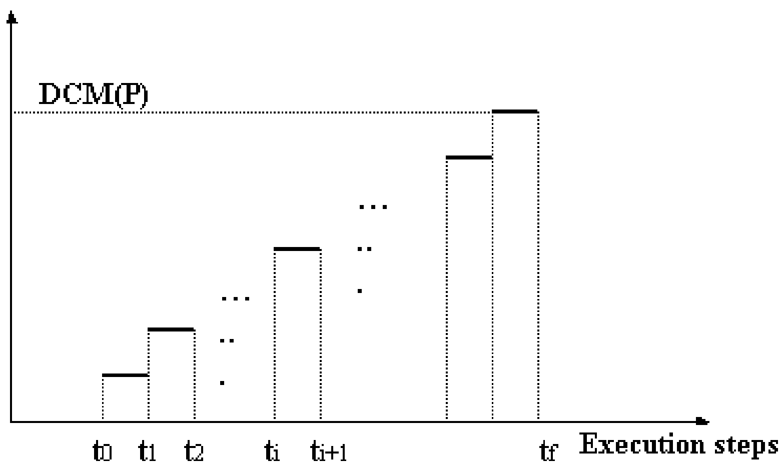

Figure 5.6: DCM increases steadily at a constant rate in case of static coupling

coupling). Here, t_0 and t_f denote programme steps at which the object was created and then destroyed, respectively. If n denotes the total number of execution steps and if P does not take part in any interaction, the coupling measure increases by a constant value at each step , and as a result, we get: $DCM(P) = n * \sum_i g_p(|O_i|)$. Figure 5.6 shows that DCM increases steadily at a constant rate in time as each execution step contributes a constant factor to the measure.

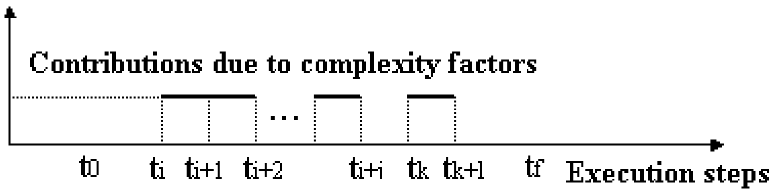

Figure 5.7: Transient coupling reduces the number of contributions to DCM

Unlike composition and inheritance, reflection provides a flexible mechanism allowing transient coupling. The mechanism underlying the relationship between application objects and meta-objects differs from that of composition and inheritance; it is flexible and allows variable coupling during program execution. As a result, we expect the contributions due the complexity measures of the coupled objects ($g_p(|O_i|)$ factor) to vary during program execution. Figure 5.7 shows a representation of these contributions to the measure of the same hypothetical (base) object of Figure 5.5 being coupled to other (meta-) objects. We consider the same set of objects as in Figure 5.5 with different coupling and assume that the system defined by this set of objects behaves in the same way as the system in Figure 5.5, i.e., provides the same interface as seen from the outside world. As before, t_0 and t_f denote program steps at which the base object is created and then destroyed, respectively. The coupling to a meta-object at the higher level takes place at a later step (t_i), after the creation of the object. The coupling is switched off at t_{i+j} and activated again at t_k and held activated until t_{k+l}. This means that contributions between t_0 and t_i, between t_{i+j+1} and t_k as well as between t_{k+1} and t_f are all set to zero.

If n denotes the same number of execution steps as in the previous case, the coupling measure of the system is: $DCM(P) = (j + l) \ * \ \sum_i g_p(|O_i|$.

Since $(j + l) < n$, it follows that static coupling measure is greater than the measure due to dynamic coupling. Figure 5.8 shows that DCM increases in time at a variable and slower rate compared to the case of static coupling.

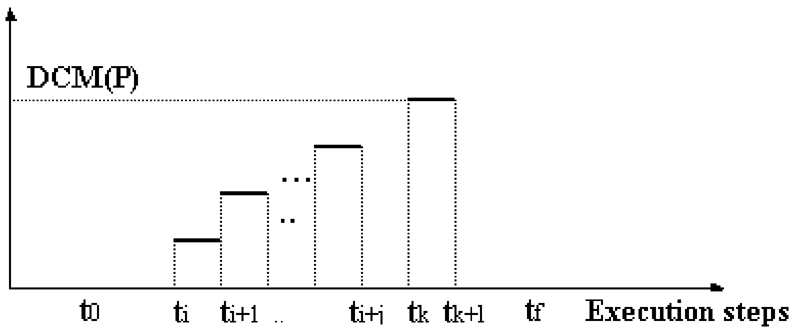

Figure 5.8: A variable and slower increase rate in case of transient coupling

We are aware of the threats to the validity of the case study carried out in this chapter. Firstly is the assumption that values from automatic data collection are the correct values. To counter this threat, significant effort, including visual checking and manual counting was done to ensure that the correct values of the DCM had been obtained. A further threat to the validity of our study is the scalability of the models considered in this work. We believe that the models can be scaled up to any number of objects. There are no logical constraints that would prevent the extension of the model to include larger numbers of objects and we feel a similar trend to that found would be shown as a result. The number of interactions, through which we compared both models, could also be reached with larger number of objects.

5.6 Conclusions

Most OO coupling measures proposed in the literature deal with coupling at the class level. Counting class attributes and method invocations does not cover the coupling features of objects at runtime. Measuring object couplings gives us insight into the system structure and allows us to compare architectural aspects of different systems relative to reuse and maintenance. In reflective systems with meta-level architectures, objects' coupling are runtime dependent and may change due to customisation of objects' behaviour or modifications to their structure.

Our DCM metric can be used to measure the coupling of particular objects and/or the entire system at runtime. It is two-dimensional: the time factor reflects the fact that it is an object coupling measure and the class complexity factor can be specified according to the goal of the measurement. The DCM can be also used to predict the runtime complexity of the system. It may also help system engineers to decide on the appropriate software components to be used in the production and maintenance phase. Classes with high object couplings need more attention and care and consequently induce higher maintenance cost; these types of classes should thus be assigned to more capable developers.

In this chapter, we presented an empirical case study of DCM. Two programming models representing different architectural design approaches while exhibiting the same behaviour were considered. In the first model, class relationships including inheritance, aggregation and association determined the type of objects' interactions at runtime. The second model adhered to a reflective architecture and showed less class coupling. In the latter model, objects' interactions were also determined by polymorphism and reflection. At each stage of the investigation, data corresponding to DCM values of individual objects, the DCM value of the system and the total number of objects involved together with the total number of objects' interactions was collected and compared. The application programs, which exhibited the same behaviour at each stage, were coded in such a way to cover all possible object interaction types.

The process of collecting DCM data for empirical validation was automated. Using the AOP approach, we developed the profiling code independently from the target programme and then merged this code with the program code. We defined concrete aspects needed to apply the interceptive code and specified the join points suitable for the application.

We found the DCM could be used to directly compare systems' coupling at runtime. We can thus foresee the use of DCM as a means of comparing runtime coupling of one system at different stages of the system's development. The approach outlined in this chapter for automating the collection of DCM data concentrates on Java source code. It is, however, directly applicable to systems coded in C++ and Smalltalk, using the aspect extensions available for these languages.

Chapter 6

Code Reuse Metrics

In Chapter 5, we applied the metric approach to assess the coupling level of reflective systems. In this chapter, we apply the same approach to assess the level of code reuse of such systems. The contribution of this chapter is to demonstrate the quantitative benefits of employing the reflection model to a key concept in object-orientation, namely, reuse.

It is widely accepted that applying reuse-supporting techniques has the potential for increasing software productivity, shortening development time and decreasing maintenance costs. Objective measurement of reuse through metrics can help us to understand such issues and realize these benefits. In this chapter, we adapt code reuse metrics proposed for procedural languages to object-oriented languages and use these metrics to assess an approach to code reuse through reflective techniques. The proposed approach consists of identifying and refactoring aspects of an application, re-implementing these aspects in a generic manner, and then composing the re-factored aspects with the rest of the application to recover original behavior.

Despite wide acceptance that reuse has the potential for increasing software productivity and enhancing software quality in terms of reliability and maintainability, reuse is rarely practiced as a standard part of the software development process, according to Krueger [72], and the quantifiable benefits of applying code reuse rarely shown, as argued by Chen et al. [32]. Definitions of reuse also vary. Krueger [72] defines software reuse as the process of creating software systems from existing software assets, rather than building software systems from scratch. Tracz [96], makes the distinction between reuse and salvaging; software salvaging is using software not originally designed for reuse. In turn, Prieto-Diaz [89] defines reuse as the use of existing software components to construct new systems. The software assets or components appearing in these definitions may include different aspects of software beyond code components. Many authors [16, 18, 45, 72, 78, 88, 97, 100] have also extended the scope of reuse to encompass, in addition to component reuse (a synonym for code reuse), all the resources used and produced during the development process.

These include concepts, tools, analysis, design, specifications, documentation, people and domain knowledge.

The OO paradigm supports code reuse through inheritance, polymorphism and aggregation and different OO languages provide extra mechanisms such as reflection to support code reuse. For example, as we have seen in the previous chapters, Java supports reuse through subtype polymorphism, introspection and generic implementation of customisation code. C++ supports reuse through parametric polymorphism known as the template mechanism.

In the previous chapters, we have shown that reflective techniques can be used to reduce coupling; such an approach was applied to both a distributed and single JVM environment [53, 57]. In this chapter, we define OO code reuse metrics as means to quantify the code reuse due to reflective techniques. We adopt refactoring principles due to Fowler and Opdyke [43, 83] and extend the code reuse metrics proposed in Chen et al. [32] to OO systems. We then propose an approach which identifies and refactors certain aspects of applications, re-implements these aspects in a generic manner (thus making them reusable) and later composes those aspects with the rest of the application to recover the original behaviour. A case study based on the Model-View Controller (MVC) pattern is used to demonstrate the benefits of this approach (a quantifiable improvement in reuse) and a reduction in application code size.

This chapter is organized as follows. In the next section, a model of software systems and code reuse measures proposed by Chen et al. are reviewed and the question of extending and adapting both procedural-based model and measures to OO systems is addressed. In Section 6.2, the code reuse metrics are evaluated against Weyuker's set of measurement principles. Section 6.3 investigates the issue of empirical validation of the proposed metrics. A case study is presented to affirm or refute the hypothesis on the high level of code reuse of reflective systems. The metrics are used to analyse code reuse data. In Section 6.4, the proposed metrics are compared with other OO reuse metrics in the context. The question of automating the process of gathering reuse data is also discussed. We end the chapter with some concluding remarks in Section 6.5.

6.1 Object-Oriented Code Reuse Measures

In this section we review the model of software systems and the reuse metrics applied to procedural-based systems of Chen et al. and adapt both to OO systems. Our choice of Chen's metrics as a basis for OO reuse metrics (and disregard for other proposed metrics) stems from the fact that firstly, Chen's metrics are system-wide (as opposed to class-based) metrics and hence give us an advantage over the majority of current OO reuse metrics. Secondly, they allow us to incorporate the essential features of OO systems without major changes to their original format. Finally, they provide an intuitive and quantifiable basis for measuring reuse and improvements in reuse when considering concrete application examples.

6.1.1 Adapting Chen's software model and measures

According to Chen et al., a software system consists of two parts: an application part and a reusable part. Formally, a system S consists of a set E of basic code entities (e). Entities may directly or indirectly depend on each other and direct dependencies among entities are expressed in terms of a set D of 2-tuples (e_1, e_2), meaning e_1 is directly dependent on e_2. It is assumed that there exists a subset R (root entities) of E such that each execution of S starts from R. In this way, S is modelled as a directed graph (R, E, D) with root set R, node set E and edge set D.

In OO systems, the basic code entities are classes, and dependencies correspond to class relationships such as inheritance, aggregation, association, etc. The subset R corresponds to object classes having a `main()` method or equivalent programme starting entity. Inheritance hierarchies represent dependencies and indirect reference. However, part of inherited code may never be used in certain application contexts and should not be considered as part of the set of entities, E. Two conditions are required of E:

- E is complete, i.e., each part of the software system is covered by some class entity in E.
- Each class entity is reachable from R.

We associate with each class entity (e) its Non-Commented Source Lines ($NCSL$) as its weight, weight(e), and define the size of S as:

$$Size(S) = \sum_{e \, \epsilon \, E} weight(e)$$

where the $\sum_{e \, \epsilon \, E}$ denotes the sum over all the class entities of the system. We note that, because of the reachability condition imposed on S, we consider reachable classes in inheritance hierarchies to include those that can be reached through polymorphism at runtime.

Having defined an OO program model, we now formalise the reusable part of S in terms of code reuse measures. Let P (subset of E) denote the set of all reusable class entities. We define $Refer(S, P)$ to be the set of entities of P that S directly refers to, and $Closure(S, P)$ to be the set of entities of P that S directly or indirectly refers to. Note that the two sets ($Refer(S, P)$ and $Closure(S, P)$) are identical in the case where the dependency graph of P is shallow, i.e., where reusable classes are independent entities. We note that both $Refer(S, P)$ and $Closure(S, P)$ are sets and hence do not allow for double-counting of elements. The reuse metric of a system S with respect to a set of reusable entities that are reachable from R is defined as:

$$ReusePart(S, P) = \sum_{e \,\epsilon\, Closure(S,P)} weight(e)$$

where $\sum_{e \,\epsilon\, Closure(S,P)}$ denotes the sum over all the reusable class entities including reachable classes in the inheritance hierarchies of directly and indirectly referenced entities. The percentage of reused code of P to the size of S is defined as:

$$AppReusePerc(S, P) = ReusePart(S, P)/Size(S)$$

In the same way, we define the measure for the percentage of the repository of reusable classes (represented by the set P) that is reused by S as:

$$RepReusePerc(S, P) = ReusePart(S, P)/Size(P)$$

In addition to the previous metrics, which are adaptations of Chen's et al. metrics, we define the measure of the application code as a complement to the reuse measure as:

$$AppPart(S) = \sum_{e \,\epsilon\, E-P} weight(e)$$

where $\sum_{e \,\epsilon\, E-P}$ denotes the sum over all the application class entities, i.e., those system entities that are considered not reusable (in E but not in P). The definitions of E, R and $Refer(S, P)$ follow the same pattern as that of procedural languages proposed by Chen et al. For the $Closure(S, P)$ set, however, we need to identify the entities that S indirectly refers to in the inheritance hierarchies of P. Although inheritance implies dependency, not all the entities in a hierarchy are relevant to code reuse since some entities will not be used in certain contexts. We need to consider only those entities in an inheritance tree that are reachable. For OO systems, the condition that each class entity is reachable from R can be implemented using (subtype) polymorphism. In strongly typed languages such as C++ and Java, the polymorphic reusable code can be used only by objects whose types belong to the hierarchy and thus identifiable at compile time. In Smalltalk, the same principle applies; identifying the reachable classes is, however, more complex and requires runtime analysis.

6.1.2 An illustrative example

To explain the OO reuse measures and related definitions, consider an application S reusing a set of entities P as shown in Figure 6.1. S consists of four class entities $\{e_1, e_2, e_3, e_4\}$ and P of the entities $\{e_5, e_6, e_7, e_8\}$ where e_7 is shown to be part of a hierarchy tree. Figure 6.1 represents a class diagram and the arrows denote class dependencies expressing class relationships such as inheritance, aggregation or association. The set E consists of all the classes $\{e_1..e_8\}$ plus the inheritance hierarchies lying behind these classes and the root set $R = \{e_1\}$. The set of entities of P, directly referred to by S is:

$$Refer(S, P) = \{e_5\}$$

The set of entities of P that S directly and indirectly refers is the *Closure* set. The *Closure* set includes all the classes that are directly and indirectly referred to, i.e., $\{e_5, e_6, e_7, e_8\}$ in addition to those classes in the underlying class hierarchies that are reachable through polymorphism (denoted by <reachable classes>):

$$Closure(S, P) = \{e_5, e_6, e_7, e_8, < reachable\ classes >\}$$

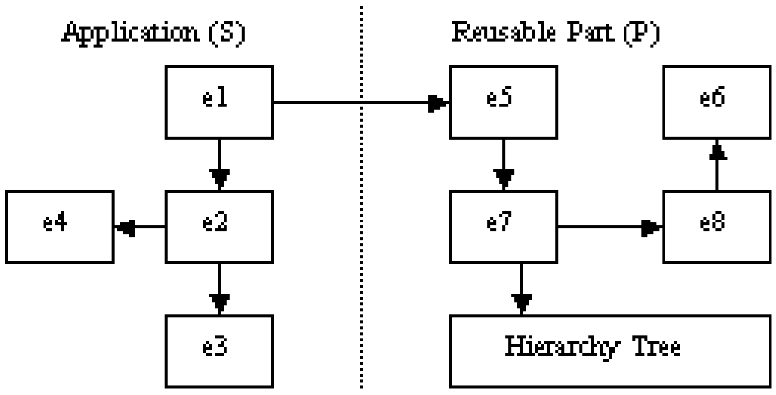

Figure 6.1: Application S reusing set P

6.2 Theoretical Validation of the Metrics

In this section, we evaluate our code reuse metrics *AppReusePerc*(S, P) and *RepReusePerc*(S, P) against Weyuker's formal set of measurement principles discussed in the previous chapter. We restrict ourselves to these two metrics, because all other measures, such as, *Size*, *ReusePart* and *AppPart* are introduced as auxiliary measures for defining the reuse percentages. We follow the same procedure applied to validating the coupling metric DCM in Section 5.3.

Recall that the metrics were defined in Section 6.1.1 in terms of the percentages of the sizes of S and P. A system S has been defined as a set of class entities E including a set of reusable entities P and modelled as a directed graph.

In the following, we need to introduce the notion of *system combination* as the union of sets of the constituent systems which involves class concatenation. In combining systems, equal system elements (classes) are made redundant and considered only once whereas corresponding classes are combined to form a single entity. As in the case of evaluating the DCM metric in Section 5.3.2 of the previous chapter, we assume that a concatenation of two class entities results in redundant methods and/or fields where only one copy of a method and/or a field is considered. We further assume that a combination class can replace the constituent classes without affecting the overall behaviour of the enclosing system.

1. Weyuker's first property requires that not every system can have the same metric values. This property is satisfied by the code reuse metrics $AppReusePerc(S, P)$ and $RepReusePerc(S, P)$ if we assume that number of classes (on the application and reusable sides) of a given system is a discrete random variable characterised by some general distribution function. Further, all such numbers are independent and identically distributed. Thus, systems classes (S as well as P) follow a statistical distribution not apparent to an observer of the system. Following this assumption, the reuse percentages of the application as well as of the repository of two different systems (S_0, P_0) and (S_1, P_1) are independent and identically distributed which implies that:

$$AppReusePerc(S_0, P_0) \neq AppReusePerc(S_1, P_1)$$

$$RepReusePerc(S_0, P_0) \neq RepReusePerc(S_1, P_1)$$

2. Weyuker's second property requires that a metric should assign the same value only for a finite of objects. Since the domains of the metric constitute finite OO executable systems, each of which contains a finite set objects, this property will be met by any metric, including our reuse percentages $AppReusePerc(S, P)$ and $RepReusePerc(S, P)$.

3. Weyuker's third property requires that different systems (entities) might have the same metric value. Here, the S and P sets of different systems can include different classes. The reuse metrics are defined as ratios of the sizes of the two sets expressed in terms of $NCSL$. The metrics are independent of the structure of the systems and also of the class semantics and structures. It follows that property three is satisfied.

4. Weyuker's fourth property requires that different systems exhibiting the same behaviour (their classes implementing the same functionality) might have different metric values. Here, the same argument applies as in the

case of the third property. The reuse metrics values are not determined by the functionality of systems classes, rather by their $NCSL$. Therefore, property four is satisfied.

5. Weyuker's fifth property states that the reuse metric values of a system obtained by combining different systems (through class concatenation) should be at least as great as the metric values of either systems.

   We show that Weyuker's fifth property is *not* satisfied by the percentage reuse metrics $AppReusePerc(S, P)$ and $RepReusePerc(S, P)$.

   Consider two systems characterised by the tuples (S_0, P_0) and (S_1, P_1), and let (S, P) denote their combination. Consider the case where (S_1, P_1) consists of a single application class, i.e., $Card(E_1) = 1$ and $P_1 = \phi$. In such a case, $ReusePart(S_1, P_1) = 0$ and consequently the percentages of reused code to the size of S_1 and to the size of P_1 vanish. Furthermore, $ReusePart(S, P) = ReusePart(S_0, P_0)$ and because $Size(S) > Size(S_0)$, it follows:

   $$AppReusePerc(S, P) < AppReusePerc(S_0, P_0)$$

   On the other hand, because $Size(P) = Size(P_0)$, the repository reuse percentages are related as follows:

   $$RepReusePerc(S, P) = RepReusePerc(S_0, P_0)$$

6. Weyuker's sixth property states that: given two systems having equal metric values, the metric values of their combination with a third system can be different.

   Let (S_0, P_0) and (S_1, P_1) denote two different systems with their reuse metrics equal. Consider a third system (S_2, P_2) together with two other systems obtained by combining each of the first two systems with the (S_2, P_2). In general, class concatenations with (S_2, P_2) results in different sets of application and reusable classes. The differences between corresponding sets in the resulting systems are independent of reuse metric values. Therefore, reuse metric values of the systems resulting from combining each of (S_0, P_0) and (S_1, P_1) with (S_2, P_2) can be different. Thus, property six is satisfied.

7. Weyuker's seventh property requires that a permutation of elements within a system should change the metric value of that system. Permutating class statements does not change its size but may change its semantics and consequently its reuse. As a result, reuse metric values of systems change according to modifications made to their classes. Thus property seven is satisfied.

8. According to Weyuker's eighth property, metric values should not change with the change of system labelling or the names of their constituent classes. Reuse metrics are defined independently of systems' names and depend only on the systems' structures and their class relationships. Consequently, this property is satisfied by our reuse metrics.

9. Weyuker's ninth property states that the reuse metric value of a combination of systems is greater than the sum of the values of the constituent systems.

   We show that Weyuker's ninth property is *not* satisfied by the percentage reuse metrics $\mathit{AppReusePerc}(S, P)$ and $\mathit{RepReusePerc}(S, P)$. Following the same arguments used in discussing property 5, we have:

   $$\mathit{AppReusePerc}(S_1, P_1) = 0$$

   $$\mathit{RepReusePerc}(S_1, P_1) = 0$$

   $$\mathit{AppReusePerc}(S, P) < \mathit{AppReusePerc}(S_0, P_0)$$

   $$\mathit{RepReusePerc}(S, P) = \mathit{RepReusePerc}(S_0, P_0)$$

   It follows that:

   $$\mathit{AppReusePerc}(S, P) < \mathit{AppReusePerc}(S_0, P_0) + \mathit{AppReusePerc}(S_1, P_1)$$

   $$\mathit{RepReusePerc}(S, P) = \mathit{RepReusePerc}(S_0, P_0) + \mathit{RepReusePerc}(S_1, P_1)$$

   Thus, it is not the case that the reuse percentages of the combined systems is greater than the sum of reuse percentages of the constituent systems. Therefore, property nine is not satisfied.

   Most of Weyuker's properties are satisfied by $\mathit{AppReusePerc}(S, P)$ and $\mathit{RepReusePerc}(S, P)$ code reuse metrics with two exceptions: Properties five and nine. Failing to meet property five implies that it is possible to divide a system where the application reuse percentage of individual constituents is greater than that of the original system. Failing to satisfy property nine means that the application reuse percentage could increase if a system is divided into smaller systems. Intuitively, this result is to be expected if systems are divided in a such a way that the whole reusable part is associated with part of the system leaving the other part with an an empty repository of reusable classes.

6.3 Empirical Validation: A Case Study

As noted in Section 5.4 where empirical validation of DCM is discussed, reflective systems based on the separation of base level and meta-level objects show less coupling when compared to equivalent static systems exhibiting the same behaviour. In this section, we conduct a case study to support the hypothesis that, in addition, reflective systems show a higher degree of code reuse.

6.3.1 An approach to code reuse

To incorporate our approach for the enhancement of code reuse, we extend the model previously described by adding the concept of a sub-system. An OO software system S is defined as a collection of related class entities. A *sub-system* (or a component) is a subset of the system's collection of logically connected classes. In other words, a sub-system encompasses a cluster of class entities that share a common property. A property represents a certain aspect of the system's behaviour. Components constitute abstractions allowing a coarse view of the system. At the component level of abstraction, a system consists of inter-dependent parts each of higher or (at least equal) granularity when compared to individual class entities; the smallest component is a class entity.

The approach we adopt consists of identifying a certain aspect of system behaviour, extracting the corresponding code from the application and re-implementing the resulting component in a generic manner thus making it independent of the system specifics. Being generic and independent of the system, such components constitute reusable pieces of code. Separating certain parts of the system is the first step. The second step is to find a mechanism for composing (or glueing) the independent part with (or to) its target system.

To implement our method, we consider two systems providing the same functionality. The second system is derived from the first by refactoring one component out, implementing this component in a generic manner (as a meta-programme) independently of the system's specifics; later we combine the component with the rest of the application to reproduce the original behaviour.

6.3.2 Improving OO code reuse through reflection

In this section, a case study is conducted with the aim of exposing the usage of the proposed OO code reuse metrics for showing the impact of reflection on code reuse. Consider the same example considered in Section 3.4.2.2 of transforming objects' attributes into active JavaBeans properties emphasizing the MVC features. We implement the example by firstly, using the JavaBeans classes directly and, secondly, extracting the code of catching and firing an event of change from application specifics and making it part of a library of reusable classes. Henceforward, we will refer to the first system as S_0 and to the second as S_1.

6.3.2.1 Direct reuse of the JavaBeans classes

Listing 6.1 shows the application classes implementing the MVC pattern with the help of the JavaBeans classes. A `PointModel` fires a `PropertyChangeEvent` every time a set method is called (lines 13 - 16); firing the event is realised by delegating the task to the aggregate object of type `PropertyChangeSupport` (line 14). A `PointView` is an event listener and is registered as such upon its creation (line 27). An application (lines 33 - 40) creates a model and a view and then calls set methods on the model object. The change of state of the model object results in the firing of an event and the view object reacts by printing out the initial and final values of the attribute (lines 29 - 31). The `Application` class represents the starting point (i.e., belongs to the root set, R).

6.3.2.2 Reusable code as meta-program

In Section 3.4.2.2, we implemented the mechanism of firing an event of change and thereby informing interested listeners using dynamic proxies. Java's dynamic proxies support behavioural reflection and allow generic coding. The code can then be applied to any application employing the MVC pattern as part of its GUI implementation. The classes used to implement the same behaviour of the previous section include a meta-program realised by the classes: `IPropertyChange` (Listing 3.3) with 6 $NCSL$, `ActivePropertyHandler` (Listing 3.4) with 50 $NCSL$ and `ProxyFactory`(Listing 3.1) with 10 $NCSL$. The application classes are similar to those of Listing 6.1 and are shown in Listing 6.2. Note that `PointModel` is independent of the JavaBeans classes and that adding a `PointView` as listener is realised by the meta-program. The `Application` class (lines $21 - 30$) provides the same behaviour as before and represents the starting point at which the generic reusable part is combined with the application classes.

6.3.3 Reuse Analysis

In case of reuse through reflection, i.e. system S_1, the set of reusable entities consists of two parts P_0 and P_1. P_0 contains those classes of the `java.beans` package needed to implement the Event-Trigger mechanism, that is, the classes `PropertyChangeSupport`, `PropertyChangeEvent` and `PropertyChangeListener`. P_1 contains the entities `ActivePropertyHandler`, `ProxyFactory` and `IPropertyChange` developed to abstract the Event-Trigger mechanism from application specifics. In the case of direct reuse of the JavaBeans classes, i.e. system S_0, the set of reusable entities is $P = P_0$ and for S_1, $P = P_0 \bigcup P_1$. We note that only explicitly used and reachable classes are considered; super-classes of the reusable entities such as `EventObject`, `EventListener` of the `java.util` package and their dependencies are not considered. We further note that the set of Entities E_0 of S_0 contains fewer elements than E_1, the set of class entities of S_1.

```java
import java.beans.PropertyChangeSupport;
import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeEvent;

interface IPointModel {
   public int getx();
   public void setx(int x);
}
class PointModel implements IPointModel {
   private int x = 0;
   private PropertyChangeSupport sup= new PropertyChangeSupport(this);
   public int getx(){ return x; }
   public void setx(int newX) {
       sup.firePropertyChange("x", x, newX);
       x = newX;
   }
   public PropertyChangeSupport getPropertyChangeSupport() {
       return sup;
   }
}
class PointView implements PropertyChangeListener {
   private IPointModel pmodel;
   public PointView(PointModel pm) {
       this.pmodel=pm;
       pm.getPropertyChangeSupport().AddPropertyChangeListener(this);
   }
   public void propertyChange(PropertyChangeEvent e){
       // Do something about the change of property ...
   }
}
public class Application {
   public static void main(String [] args) throws Throwable {
       PointModel wrap=new PointModel();
       PointView pview=new PointView(wrap);
       wrap.setx(<arbitrary value>);  // set x to an arbitrary value
       wrap.getx();
   }
}
```

Listing 6.1: MVC implementation using JavaBeans

Table 6.1 shows the entities of the two systems with their weights. The "Entity" column refers to all entities existing in both systems, S_0 and S_1. There are entities common to both systems; the two systems share the same application classes (`IPointModel, PointModel, PointView, Application`) with different implementations as their weights indicate. The two systems also share the reusable JavaBeans classes (`PropertyChangeEvent, PropertyChangeSupport, PropertyChangeListener`). The values of $NCSL(S_0 \text{ or } S_1)$ refer to the weights of entities. If an entity is assigned a zero weight, it does not belong to the corresponding system. For example, the reusable classes `ActivePropertyHandler`, `ProxyFactory`, `IPropertyChange` which were extracted from S_1 to implement

```
interface IPiontModel {//as before }                                //1
                                                                    //..
class PointModel implements IPointModel {
   private int x = 0;                                               //7
   public int getx() { return x; }                                  //8
   public void setx(int newX) { x = newX; }                         //9
}
class PointView implements PropertyChangeListener {                 //12
   private IPointModel pmodel;                                      //13
   public PointView(PointModel pm) {                                //14
       this.pmodel=pm;                                              //15
   }                                                                //16
   public void propertyChange(PropertyChangeEvent e) {              //17
       // Do something about the change of property ...
   }                                                                //19
}                                                                   //20
public class Application {                                          //21
   public static void main(String [] args) throws Throwable {       //22
       PointModel pt=new PointModel();
       PointView pview=new PointView(pt);
       ProxyFactory bpf=new ProxyFactory();
       IPointModel wrap=(IPointModel)bpf.createProxy(pt, pview);
       wrap.setx(<arbitrary value>);  // set x to an arbitrary value
       wrap.getx();
   }
}                                                                   //30
```

Listing 6.2: Application classes of the second system

the Event-Trigger mechanism in a generic way are not part of S1 and are therefore assigned zeros for $NCSL(S_0)$. These classes, together with other reusable and reachable classes such as `Proxy`, `InvocationHandler`, `Serializable` and `Object` are assigned appropriate weights under the $NCSL(S_1)$ column. The column "reusable" identifies whether an entity belongs to the reusable part $(Y - value)$ or to the application part $(N - value)$ of the system. It follows that if an entity is assigned a zero weight, i.e., an entity does not belong to the system, the reusable boolean value $(YorN)$ becomes irrelevant. The reusable sets of the two systems, $P = P_0$ and $P = P_0 \bigcup P_1$ can be derived from the combination of the corresponding weight column $(NCSL)$ with the "reusable " column. We note that the measure of the application part of S_0 is greater than that of S_1 (c.f. Section 6.1.1):

$$AppPart(S_0) = \sum_{e \, \epsilon \, E_0 - P_0} weight(e) = 4 + 12 + 10 + 8 = 34$$

$$AppPart(S_1) = \sum_{e \, \epsilon \, E_1 - P} weight(e) = 4 + 5 + 9 + 10 = 28$$

The measures of the reuse part of the two systems exhibit the opposite relationship:

$$ReusePart(S_0, P_0) = \sum_{e \,\epsilon\, Closure(S_0,P_0)} weight(e) = 28 + 195 + 4 + 38 = 265$$

$$ReusePart(S_1, P) = 28 + 195 + 4 + 50 + 10 + 6 + 146 + 4 + 3 + 38 = 484$$

The reuse part of S_1 is larger than that of S_0 and uses the Java class library to a larger extent than S_0. The ratio of the code developed to enhance code reuse (`ActivePropertyHandler, ProxyFactory, IPropertyChange`) to the reuse measure totals $66/484 = 14\%$. Table 6.1 also shows the reuse measures of the two systems, the *AppReusePerc* and the *RepReusePerc* values. In calculating the latter, we assume that the set of reusable entities is P of S_1 consisting of all reusable classes whose weight is non-zero. The *AppReusePerc* ratios (i.e., the $NCSL$ of $Y/(Y + N)$ from Table 6.1) give $265/299 = 89\%$ and $484/512 = 95\%$ for S_0 and S_1, respectively. The *RepReusePerc* values $265/484 = 55\%$ and $484/484 = 100\%$ for S_0 and S_1, respectively, indicate a significantly higher reuse of repository code by S_1.

Entity e	$NCSL(S_0)$	$NCSL(S_1)$	Reusable
PropertyChangeEvent	28	28	Y
PropertyChangeSupport	195	195	Y
PropertyChangeListener	4	4	Y
ActivePropertyHandler	0	50	Y
ProxyFactory	0	10	Y
IPropertyChange	0	6	Y
Proxy	0	146	Y
InvocationHandler	0	4	Y
Serializable	0	3	Y
Object	38	38	Y
IPointModel	4	4	N
PointModel	12	5	N
PointView	10	9	N
Application	8	10	N

System	AppReusePerc	RepReusePerc
S_0	89%	55%
S_1	95%	100%

Table 6.1: Entities of both systems and reuse percentages

We note that the application code of S_1 is 28 $NCSL$ compared with 34 for S_0; application code has been reduced as a result of using reflection. The higher reuse percentages of S_1 are due to the fact that we, firstly, re-factored the firing of events and informing of listeners out and separated them from basic classes (Views and Models) and secondly, we implemented the Event-Trigger mechanism in a generic manner and thus made it reusable. In the following section, we discuss some of the issues raised by this analysis.

6.4 Discussion

A number of metrics in the past have tried to capture the extent of software reuse [35, 63, 87]. None of these metrics are suitable for measuring code reuse of an entire program and none of them takes the effect of reflection on code reuse into consideration. The metric of Poulin [87] is concerned with estimating the financial benefit of software development within *product lines.* The metrics defined by Chidamber and Kemerer [35] are applicable at the design level and those which can be considered as reuse metrics are useful for assessing OO reuse attributes of individual classes only. The metrics proposed by Kang and Bieman [63] are useful for identifying the most appropriate shape of an inheritance hierarchy for supporting code reuse. Our metrics, on the other hand, can be used to assess the code reuse of entire programs and are not restricted to OO reuse mechanisms supported by OO paradigm, namely, inheritance and aggregation.

A number of issues need to be addressed with a study of the type conducted in this chapter. Firstly, we have compared our approach with an application that uses non-reflective techniques. Although there are various other models against which we could compare our approach, in this case at least we feel that the benefits of reuse obtained (on balance) outweigh an obvious, yet necessary overhead of extra lines of code for implementing the reflective architecture (213 extra lines of code). The key point of the research however is that lines of application code have decreased, a benefit that we believe would be magnified with larger systems (the overhead would remain the same). Secondly, we see our approach as a step towards showing that use of reflective techniques can have tangible benefits. We see these techniques as ways that could help developers more easily produce highly modular, reusable code. Thirdly, use of appropriate metrics has allowed a comparison in concrete terms to be made. One of the key reasons why Chen's metrics were chosen was because of the ease with which we could apply them to the OO paradigm.

Finally, there is the issue of automating the process described in this chapter. Automating the collection of reusable data would require identifying the reusable parts of the system and hence those elements of the application to be refactored. In our approach, the reusable part includes also those classes that can be reached through polymorphism. For strongly-typed programming languages like C++ and Java, it is easier to identify the polymorphic classes in the inheritance hierarchy when compared to languages such as Smalltalk that do not perform type checking at compile time. To extend this work to industrial-sized systems, we would need to develop tools to assist the process. Further studies of an empirical nature would help to establish conclusions about reflective techniques and allow their value to be confirmed or refuted.

6.5 Conclusions

In OO systems, there is significant potential for code reuse through reflective techniques. In this chapter, we adopted an approach whereby certain aspects of an application were identified, refactored and implemented in a generic manner. Composing or re-integrating the reusable code with the rest of the application was realised at runtime. To assess the impact of reflection on code reuse, we proposed OO code reuse metrics and used them to assess and compare reusability levels of two system models representing different architectural design approaches while exhibiting the same behaviour. Our analysis affirmed the hypothesis that reflective systems where meta-code is implemented generically show a higher level of reusability when compared with identical non-reflective systems. We have showed that by separating reusable parts from the application and by applying the metrics of code reuse, benefits in terms of percentage reuse could be demonstrated.

Chapter 7

Conclusions and future work

7.1 Theme and goals of the thesis

It is widely agreed that compared to other engineering disciplines, software technology has a relatively low degree of reusability. Applications adapt poorly to changes in environment or intended use, scale poorly and are difficult to debug, maintain and enhance. As the scale of complexity of applications grow, there is a need to develop software systems with the ability to adapt to both changes in their environment and to requirements.

Reflective systems are equipped with means that makes them more adaptable to new requirements when compared with non-reflective systems. This is due to the fact that such systems are supported by powerful runtime environments and promote code reuse and less coupling if implemented in a generic manner.

Although OO languages support reflection (to varying degrees) and although reflection can be useful in controlling and adapting software, little effort has been made to promote reflection in the development of industrial systems. Also, little research been conducted in investigating software features such as coupling and reuse. The research reported in this thesis attempts to fill this gap.

The objectives of the thesis as stated in Chapter 1 were to investigate the reflective capabilities of OO programming languages taking Java as a good representative and to assess quantitatively the extent of coupling and code reuse of reflective systems; in particular, systems employing Java's reflection models.

7.2 The contribution of the thesis

In general terms, the main contribution of the thesis was to shed light on reflective capabilities of OO languages and, in particular, that of Java. Analysis of systems employing reflective techniques showed that such systems support open implementation. Reflective systems also exhibit less coupling and a higher degree of code reuse when compared with equivalent systems following the classical (non-reflective) OO paradigm. Features, such as support of open implementation, low coupling and high reuse support adaptability and potentially lower cost maintenance.

In particular, the contribution of the research reported in the thesis can be summarized as follows:

1. A behavioural reflection model based on Java's dynamic proxies has been proposed and it has been shown that this model supports open implementation design principle and generic coding. The model allows customisation of applications behaviour at runtime without changing the underlying default implementation [53].

2. The proposed model has been shown to be applicable to distributed systems including RMI-based, CORBA and Web applications. Here, customising generic code is implemented at the server side and represents services which can be requested by client objects. Depending on the networking technology, clients' requests are communicated using the appropriate protocol; Java Remote Method Protocol (JRMP) for RMI-based systems, the Internet Inter Object request broker Protocol (IIOP) for CORBA applications and the HyperText Transfer Protocol (HTTP) for Web communications [57].

3. The issue of coupling in systems employing the proposed reflection model by applying measurement theory has been addressed. A Dynamic Coupling Metric (DCM) representing coupling in a dynamic sense has been defined. In the context of measuring object coupling, a system is viewed as a collection of interacting objects. One important feature of the DCM is that it expresses explicitly the dependency of coupling on programme execution steps or object interactions [55].

   For the theoretical validation of DCM, the metric has been evaluated against two sets of formal criteria: Weyukers's set of measurement principles and the set of coupling metrics properties due to Briand et al.

4. The question of the empirical validation of the DCM by developing a coupling measurement tool using AOP techniques has also been addressed. Intercepting code needed to capture objects' exchange of messages can be inserted in sample programs automatically by specialising the generic implementation of the tool and using AOP weaving tools [56].

5. The issue of code reuse in systems employing the proposed reflection model using appropriately defined OO code reuse metrics has been addressed. Contrary to previous OO reuse metrics defined for individual classes, for inheritance trees and for assessing the economical benefits of

product lines, our metrics are applicable to entire OO systems. That is, they are mappings from the domain of OO systems into the set of real numbers where the numbers represent the code reuse values of systems under study.

For the theoretical validation of the code reuse metrics, Weykers's set of measurement has been used as validation criterion.

For empirical validation, a procedure based on reflection and refactoring has been applied as part of a case study. The study allowed us to confirm the hypothesis about the relatively higher level of code reuse in reflective systems when compared to equivalent systems not employing reflection [54].

6. The impact of employing the proposed reflection model on design patterns of Gamma et al. has been considered. As a result of applying the reflection model, some pattern implementations became simpler, some showed less coupling and higher reuse, whereas for some patterns, in particular the creational patterns, the model had no impact.

The investigations followed in this work have shown, at least in the case of Java as a strongly-typed OO programming language providing limited support to reflection, that reflection supports low coupling, enhances the level of code reuse and, as a result, is beneficial in reducing potential maintenance costs. Applying reflective techniques is a useful programming practice that opens systems, lessens coupling, promotes reuse and consequently leads to the construction of robust and flexible software systems.

In the case of the Java language extension supporting the AOP paradigm, i.e., AspectJ and Hyper/J, we have shown that reflection adds to the power of these languages supporting the principle of separation of concerns [51, 52]. Applying Java's introspective API which amounts to an non-extensible structural reflection model leads to the development of generic and loosely coupled AspectJ aspects as well generic and loosely coupled Hyper/J hyperslices. We expect that the results obtained apply also to other languages such as Smalltalk-80, C++ and C#.

7.3 Future work

The research reported in this thesis opens the way for a number of issues that could be addressed as part of future research program. Future research could:

1. Investigate the reflective capabilities of other OO languages and check whether they show similar features such as supporting open implementation, low coupling and a higher code reuse level.

2. Investigate the impact of applying compile-time MOPs such as those of OpenJava and OpenC++ on code reuse and coupling.

3. Compare DCM with static coupling metrics defined at the class level. The aim is to test the predictive power of DCM in relation to external system attributes and compare results with those known from previous investigations performed with static metrics.

4. Apply DCM to industrial sized systems using the measuring tool developed for validating the metric experimentally.

5. Develop tools for automating the procedure proposed for enhancing the reuse level of a system and for collecting code reuse data for industrial sized systems.

6. Investigate the question of how the proposed reflection model which supports open implementation design can be used in solving a potential combinatorial explosion problem or the scaling problem of Java libraries.

7. Address the problem of using the proposed reflection model for defining Java MOPs. That is, defining a programming interface that allows Java users to customise the behaviour of the language itself instead of customising applications.

References

[1] *About the Object Management Group.* available at OMG site http://www.omg.org/gettingstarted/index.htm, 2004.

[2] *Ada Home: The Web Site for Ada.* http://www.adahome.com, 2004.

[3] *Aspect Oriented Software Development (AOSD) research projects.* http://www.aosd.net/technology/research.php, 2004.

[4] *AspectJ.* at IBM site http://www.eclipse.org/aspectj/, 2004.

[5] *CORBA Technology and the Java 2 Platform, Standard Edition.* available at Sun Microsystems site http://java.sun.com/j2se/1.4.2/docs/guide/corba/index.html, 2004.

[6] *HTTP specifications.* available at World Wide Web Consortium (W3C) site http://www.w3.org/Protocols/HTTP/, 2004.

[7] *Java 2 platform.* available at Sun Microsystems site http://java.sun.com, 2004.

[8] *Java Remote Method Invocation.* available at Sun Microsystems site http://java.sun.com/products/jdk/rmi/, 2004.

[9] *OMG IDL: Details.* available at OMG site http://www.omg.org/gettingstarted/omg_idl.htm, 2004.

[10] *UML: Catalog of OMG Modelling and Metadata Specifications.* Object Management Group (OMG) site at http://www.omg.org/technology/documents/formal/uml.htm, 2004.

[11] C. Alexander. *The Timeless Way of Building.* Oxford University Press, NY, USA, 1979.

[12] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King and S. Angel. *A Pattern Language.* Oxford University Press, NY, USA, 1977.

[13] E. Arisholm. Dynamic coupling measures for object-oriented software. In *Proceedings of the 8th IEEE Symposium on Software Metrics (METRICS'02), Ottawa, Canada)*, pages 33–42. IEEE Computer Society, 2002.

[14] E. Arisholm, L. C. Briand and A. Føyen. Dynamic Coupling Measurement for Object-Oriented Software. Simula TR 2003-5 and Carleton TR SCE-03-18, Simula Research Laboratory and Carleton University Technical Reports, 2003.

[15] V. R. Basili, L. C. Briand and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, October 1996.

[16] V. R. Basili and H. D. Rombach. Toward a comprehensive framework for reuse: A reuse-enabling software evolution environment. Tech. Report CS-TR-2 158, CS Dept., Univ. of Maryland, 1988.

[17] J. M. Bieman and J. X. Zhao. Reuse through inheritance: A quantitative study of C++ software. In *Proceedings ACM Symposium on Software Reusability (SSR'95), Seattle, WA, USA*, pages 47–52. ACM SIGSOFT Software Engineering Notes, Special Issue, 1995.

[18] T. J. Biggerstaff. Classification of reusable modules. In T. J. Biggerstaff and A. J. Perlis, editors, *Software Reusability- Concepts and Models*, pages 99–124. ACM Press, New York, 1989.

[19] T. J. Biggerstaff. The library scaling problem and the limits of concrete component reuse. In *Proceedings of the Third International Conference on Software Reuse, Rio de Janeiro, Brazil*, 1994.

[20] T. J. Biggerstaff. Second order reusable libraries and meta-rules for component generation. In *7th Annual Workshop on Institutionalizing Software Reuse (WISR 7), St. Charles, Illinois, USA*, 1995.

[21] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales and D. A. Moon. Common lisp object system specification. *ACM SIGPLAN Notices*, 23(Special Issue):1–142, September 1988.

[22] D. G. Bobrow, R. P. Gabriel and J. L. White. CLOS in context: The shape of the design space. In A. Paepke, editor, *Object-Oriented Programming: The CLOS perspective.* MIT Press, MA, USA, 1993.

[23] D. G. Bobrow and M. Stefik. *The LOOPS Manual.* Xerox PARC, 1983.

[24] A. Borning and T. OShea. Deltatalk: an empirically and aesthetically motivated simplification of the Smalltalk-80 language. In *European conference on object-oriented programming (ECOOP'87), Paris, France*, pages 1–10. Springer Verlag, 1987.

[25] L. C. Briand, J. Daly and J. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, 1999.

[26] L. C. Briand, P. T. Devanbu and W. L. Melo. An investigation into coupling measures for C++. In *Proceedings of the 19th international Conference on Software Engineering (ICSE'97), Boston, USA*, 1997.

[27] M. Bunge. *Treatise on Basic Philosophy: Vol. 3: Ontology I: The Furniture of the World.* Reidel, Boston, MA, 1977.

[28] M. Bunge. *Treatise on Basic Philosophy: Vol. 4: Ontology II: A World of Systems.* Reidel, Boston, MA, 1979.

[29] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerland and M. Stal. *PatternOriented Software Architecture.* John Wiley and Sons, 1996.

[30] C#. *Microsoft Visual C# .NET : Language Reference.* Microsoft Corporation, Microsoft Press, Edmond, WA, USA, 2002.

[31] G. Cargill. *C++ Programming Style.* Addison-Wesley, Reading, MA, USA, 1992.

[32] Y. Chen, B. Krishnamurthy and K. Vo. An objective reuse metric: Model and methodology. In *Proceedings of the 5th European Software Engineering Conference, Barcelona, Spain*, pages 109–123. Springer-Verlag London, UK, 1995.

[33] J. C. Cherniavsky and C. H. Smith. On Weyuker's axioms for software complexity measures. *IEEE Transactions on Software Engineering*, 17(6):636–638, 1991.

[34] S. Chiba and M. Tatsubori. Yet another java.lang.class. In *Proceedings of the ECOOP'98 Workshop on Reflective Object-Oriented Programming and Systems, Brussels, Belgium*, July, 1998.

[35] S. R. Chidamber and C. F. Kemerer. A metric suite for object-oriented design. *IEEE Transactions on Software Engineering*, 20(6):467–493, 1994.

[36] P. Cointe. *Génie logiciel et réflexion.* DEA 2003-04 : sujets et cours du groupe OBASCO, available online at http://www.emn.fr/x-info/obasco/dea/Welcome.htmltc, 2003/2004.

[37] J. Coplien, D. Schmidt and eds. *Pattern Languages of Program Design.* Addison Wesley, Reading, USA, 1995.

[38] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools and Applications.* Addison-Wesley, NJ, USA, 2000.

[39] F. N. Demers and J. Malenfant. Reflection in logic, functional and object-oriented program-ming: a short comparative study. In *Proceedings IJ-CAI'95 Workshop on Reflection and Metalevel Architectures and Their Applications in AI*, pages 29–38, 1995.

[40] P. Donohue. *Software Product Lines: Experience and Research Directions.* Kluwer Academic Publishers, Netherlands, 2000.

[41] J. Eder, G. Kappel and M. Schrefl. Coupling and Cohesion in Object-Oriented Systems. Technical Report, University of Klagenfurt, 1994.

[42] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous & Practical Approach, 2nd Edition.* International Thomson Computer Press, London, 1997.

[43] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 2000.

[44] J. Freber. Computational reflection in class based object-oriented languages. In *Proceedings of the Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'89)*, pages 317–326. SIGPLAN Notices, ACM Press, 1989.

[45] P. Freeman. A perspective on reusability. In P. Freeman, editor, *Tutorial: Software Reusability*, pages 2–8. IEEE Computer Society Press, Washington D.C., 1987.

[46] Richard Gabriel. *Patterns of Software: Tales from the Software Community.* Oxford University Press, 1996.

[47] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, USA, 1994.

[48] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[49] D. A. Gustafson and B. Prasad. Properties of software measures. In T. Denvir et al, editor, *Formal Aspects of Measurement*, pages 179–193. Springer-Verlag, Berlin, 1992.

[50] R. Harrison, S. J. Counsell and R. V. Nithi. An investigation into the applicability and validity of object-oriented design metrics. *Empirical Software Engineering*, 3(3):255–273, 1998.

[51] Y. Hassoun and C. A. Constantinides. The development of generic definitions of hyperslice packages in Hyper/J. *Electronic Notes in Theoretical Computer Science. Elsevier pubishing company*, 82(5), 2003.

[52] Y. Hassoun and C. A. Constantinides. Visibility considerations and notion of aspect reusability in AspectJ. In *Proceedings of the 3rd Workshop on Aspect-Oriented Software Development of the SIG Object-Oriented Software Development, Germany, March 4-5, 2003*, pages 33–38. German Informatics Society (GI), 2003.

[53] Y. Hassoun, R. Johnson and S. Counsell. Reusability, open implementation and java's dynamic proxies. In *Proceedings ACM 2nd International Conference on the Principles and Practice of Programming in Java (PPPJ'03)*, pages 3–6. ACM International Conference Proceeding Series, 2003.

[54] Y. Hassoun, R. Johnson and S. Counsell. Code Reuse Through Reflection: An Empirical Perspective. Technical Report BBKCS-04-06, Birkbeck College London, School of Computer Science and Information Systems, 2004.

[55] Y. Hassoun, R. Johnson and S. Counsell. A dynamic runtime coupling metric for meta-level architectures. In *Proceedings 8th European Conference on Software Maintenance and Reengineering (CSMR'04)*, pages 339–346. IEEE Computer Society Press, 2004.

[56] Y. Hassoun, S. Counsell and R. Johnson. Dynamic Coupling Metric: Proof of Concept. *To appear in Journal of IEE Proceedings- Software*, 2005.

[57] Y. Hassoun, R. Johnson and S. Counsell. Applications of dynamic proxies in distributed environments. *Journal of Software- Practice and Experience*, 35(1):75–99, Jan 2005.

[58] B. Henderson-Sellers. *A Book of Object-Oriented Knowledge: Object-Oriented Analysis, Design and Implementation.* Prentice Hall Object-Oriented Series, 1992.

[59] B. Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity.* Prentice Hall, NJ, USA, 1996.

[60] M. Hitz and B. Montazeri. Measuring coupling and cohesion in object-oriented systems. In *Proceedings of the International Symposium on Applied Corporate Computing, Monterey, Mexico*, 1995.

[61] D. Ince and M. Shepperd. *The Derivation and Validation of Software Metrics.* Oxford University Press, 1993.

[62] R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[63] B. K. Kang and J. Bieman. Inheritance tree shapes and reuse. In *Proceedings of the 4th International Software Metrics Symposium (Metrics'97), Albuquerque, NM, USA*, pages 34–42, November, 1997.

[64] S. E. Keene. *A programmer's guide to object-oriented programming in Common LISP*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1988.

[65] G. Kiczales. Beyond the black box: open implementation. *IEEE Software*, 13(1):8–11, January 1993.

[66] G. Kiczales. Towards a new model of abstraction in software engineering. In *Proceedings of the International Workshop on Reflection and Meta-Level Architecture (IMSA'92), Tokyo, Japan*, Nov, 1992.

[67] G. Kiczales, J. M. Ashley, L. Rodriguez, A. Vahdat and D. G. Bobrow. Metaobject protocols: Why we want them and what else they can do. In A. Paepcke, editor, *Object-Oriented Programming: The CLOS Perspective*, pages 101–118. The MIT Press, Cambridge, MA, 1993.

[68] G. Kiczales, J. des Rivieres and D. G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, 1991.

[69] G. Kiczales, J. Lamping, C. V. Lopes, C. Maeda, A. Mendhekar and G. Murphy. Open implementation design guidelines. In *Proceedings of the 19th international conference on Software engineering, Boston, MA, USA*, pages 481–490. ACM Press New York, NY, USA, 1997.

[70] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97), Jyvaskila, Finland, June 1997*, pages 220–241. Lecture Notes in Computer Science 1241, 1997.

[71] G. E. Krasener and S. T. Pope. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, 1988.

[72] C. W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–184, 1992.

[73] W. Li and S. Henry. Object-oriented metrics that predict maintainability. *Journal of System and Software*, 23(2):111–122, 1993.

[74] B. H. Liskov. Data abstraction and hierarchy. *ACM SIGPLAN Notices*, 23(5):17–34, May 1988.

[75] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics.* Prentice-Hall, NJ, USA, 1994.

[76] P. Maes. Concepts and experiments in computational reflection. In *Proceedings of the conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'87)*, pages 147–155. ACM Press New York, NY, USA, 1987.

[77] P. Maes. Issues in computational reflection. In P. Maes and D. Nardi, editors, *Meta-Level Architectures and Reflection.* Elsevier Science Inc., NY, USA, 1988.

[78] B. Meyer. Reusability: the case of software design. In W. Tracz, editor, *Software Reuse: Emerging Technology*, pages 201–215. IEEE Computer Society Press, 1988.

[79] B. Meyer. *Eiffel: The Language.* Prentice Hall, NJ, USA, 1991.

[80] B. Meyer. *Object-Oriented Software Construction, 2nd Edition.* Prentice Hall, NJ, USA, 1997.

[81] A. Mitchell and J. F. Power. Toward a definition of run-time object-oriented metrics. In *7th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering, Darmstadt, Germany*, 2003.

[82] P. Norvig. *Design Patterns in Dynamic Programming.* Object World 96, Boston MA, USA, 2004.

[83] W. Opdyke. Refactoring Object-Oriented Frameworks. Phd Thesis, university of illinois, MIT Laboratory of Computer Science, 1992.

[84] J. Ortega-Arjona and G. Roberts. The architectural development pattern. In *Proceedings of the 4th European Conference on Pattern Languages of Programming (EuroPLoP'99), Kloster Irsee, Germany*, July, 1999.

[85] D. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 5(12):1053–1058, December 1972.

[86] G. A. Pascoe. Encapsulators: A new software paradigm in Smalltalk-80. In *Proceedings of the conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'86), Portland, OR, USA*, pages 341–346, 1986.

[87] J. Poulin and S. Jeffrey. The economics of software product lines. *International Journal of Applied Software Technology*, 3(1):20–34, 1997.

[88] R. Prieto-Diaz. Making software reuse work: An implementation model. *IEEE Software*, 16(3):61–61, 1991.

[89] R. Prieto-Diaz. Status report: Software reusability. *IEEE Software*, 10(3):61–66, May 1993.

[90] B. C. Smith. Reflection and Semantics in a Procedural Language. Phd Thesis, mit Technical Report 272, MIT Laboratory of Computer Science, 1982.

[91] B. C. Smith. What do you mean, meta? In *Proceedings of the ECOOP/OOPSLA'90 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming, Ottawa, Ontario, Canada*, 1990.

[92] B. Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, Reading, MA, 1994.

[93] G. T. Sullivan. Advanced Programming Language Features for Executable Design Patterns. Lab Memo, number AIM-2002-005, MIT Artificial Intelligence Laboratory, 2002.

[94] P. Tarr and H. Ossher. *Hyper/J User and Installation Manual.* IBM Research, 2000.

[95] M. Tatsubori, S. Chiba, M. O. Killijian and K. Itano. Openjava: A class-based macro system for java. In W. Cazzola, R. J. Stroud and F. Tisato, editors, *Reflection and Software Engineering*, pages 117–133. Lecture Notes in Computer Science, Springer-Verlag, 2000.

[96] W. Tracz. Software reuse myths. *ACM Software Engineering Notes*, 13(1):17–21, January 1988.

[97] W. Tracz. Summary. In *3rd International Conference on Software Reusability*, pages 21–25. ACM SIGSOFT, Software Engineering Notes, 1995.

[98] J. Vlissides, N. Kerth, , J. Coplien and eds. *Pattern Languages of Program Design 2.* Addison Wesley, Reading, USA, 1996.

[99] Y. Wand and R. Weber. An ontological model of an information system. *IEEE Transactions on Software Engineering*, 16(11):1282–1292, 1990.

[100] P. Wegner. Varieties of reusability. In P. Freeman, editor, *Tutorial: Software Reusability*, pages 24–38. IEEE Computer Society Press, Washington D.C., 1987.

[101] D. L. Weinreb and D. A. Moon. *Lisp Machine Manual.* Artificial Intelligence Laboratory, MIT, Cambridge, MA, July, 1981, 1981.

[102] E. J. Weyuker. Evaluating software complexity measures. *IEEE Transactions on Software Engineering*, 14(9):1357–1365, 1988.

[103] Niklaus Wirth. *Modula-2.* Modula-2 site at http://www.modula2.org/modula-2.php, 2004.

[104] J. Withey. Investment Analysis of Software Assets for Product Lines. Technical Report, CMU/SEI-96-TR-010, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1996.

[105] S. Yacoub, H. Ammar and T. Robinson. Dynamic metrics for object-oriented designs. In *Proceedings of the 6th International Symposium on Software Metrics (Metrics'99), Boca Raton, Florida, USA*, pages 50–61. IEEE Computer Society, 1999.

[106] H. Zuse. Properties of software measures. *IEEE Transactions on Software Engineering*, 1:225–260, 1992.