Université des Sciences et Technologies de Lille – Lille 1

Département de formation doctorale en informatique **École doctorale SPI Lille**

UFR IEEA

Adaptability and Encapsulation in Dynamically-Typed Languages: Taming Reflection and Extension Methods

THÈSE

présentée et soutenue publiquement le 21 janvier 2016

pour l'obtention du

Doctorat de l'Université des Sciences et Technologies de Lille

(spécialité informatique)

par

Camille Teruel

Composition du jury

Rapporteurs :	Wolfgang De Meuter (Professor – Vrije Universiteit Brussel)
	Roland Ducournau (Professor – Université de Montpellier)
Examinateur :	Serge Stinckwich (Maître de conférences – Université de Caen Normandie)
Directeur de thèse :	Stéphane Ducasse (Directeur de recherche – INRIA Lille Nord-Europe)
Co-Encadreur de thèse :	Damien Cassou (Maître de conférences – USTL – INRIA Lille Nord-Europe)

Laboratoire d'Informatique Fondamentale de Lille — UMR USTL/CNRS 8022

INRIA Lille - Nord Europe

Numéro d'ordre: 41970

Abstract

Users expect modern software to be both continually available and updated on the fly. Introduction of new features and integration of bug fixes should not trouble the smooth running of an application. Developers also benefit from dynamic modifications of software, *e.g.* they can adapt code to new unanticipated situations or they can run dynamic analyses to get feedback about the behavior of running applications for monitoring, debugging and optimization purposes. Programming languages and their runtimes should thus provides developers with mechanisms that improve software adaptability.

At the same time, the increasing size and complexity of software call for applications made up of many interacting components developed by different parties. In this context, either all components are trusted or each component needs to be properly protected from other ones. For example, a plugin should not be able to manipulate the internal structures or leaks private data of its host application unrestrictedly. Likewise, a malicious library should not be able to corrupt its clients or interfere with their operations. Programming languages should thus provide developers with mechanisms that improve software isolation, such as encapsulation and custom access control policies.

The need for dynamic adaptations that can change nearly every aspect of an application on the one hand and the need for properly confined components on the other hand pulls programming language design in two opposite directions: either more flexibility or more control. This dissertation studies this tension in the context of dynamically-typed object-oriented languages with two language mechanisms that promote software adaptability: *reflection* and *extension methods*. For both mechanisms, we demonstrate their usefulness, their harmful effect on encapsulation and propose solutions to retain their power while maintaining encapsulation.

Résumé

Les utilisateurs s'attendent à ce que les logiciels d'aujourd'hui soient à la fois continuellement disponibles et mis-à-jour à la volée. L'introduction de nouvelles fonctionnalités et l'intégration de correctifs ne doit pas perturber la bonne marche d'une application. Les développeurs bénéficient aussi de la modification dynamique des logiciels: par exemple, ils peuvent adapter le code à de nouvelle situations non anticipées ou bien ils peuvent exécuter des analyses dynamiques pour obtenir du retour sur le comportement d'applications en cours d'exécution, et ce, à des fins de monitorage, de déboggage et d'optimisation. Les languages de programmation et leurs environments d'exécution doivent donc fournir aux développeurs des mécanismes qui améliorent l'adaptabilité logicielle.

Dans le même temps, l'augmentation de la taille et de la complexité des logiciels requièrent des applications faites de nombreux composants développés par différents groupes. Dans ce contexte, soit tous les composant sont supposés dignes de confiance ou bien chaque composant doit être convenablement protégé des autres. Par exemple, un greffon logiciel ne doit pas être capable de manipuler les structures internes ou bien de divulguer les données privée de son application hôte de manière non contrôlée. De la même manière, une bibliothèque malveillante ne doit pas pouvoir corrompre ses clients ou interférer dans leurs opérations. Les languages de programmation doivent donc fournir aux développeurs des mécanismes qui améliorent l'isolation logicielle, tels que l'encapsulation et des politiques de contrôle d'accès sur mesure.

Le besoin de supporter des adaptations dynamiques qui peuvent change quasiment chaque aspect d'une application d'un côté, et le besoin d'avoir des composants convenablement confinés d'un autre côté, poussent la conception des languages de programmation dans deux directions opposées: soit plus de flexibilité, soit plus de contrôle. Cette dissertation étudie cette tension dans le contexte des languages orientée-objet dynamiquement typés via deux mécanismes qui promeuvent l'adaptabilité logicielle: la *réflexion* et les *méthodes d'extension*. Pour chacun de ces mécanismes, nous démontrons leur utilité, leurs effets néfastes sur l'encapsulation et proposons des solutions qui conservent leurs forces tout en maintenant l'encapsulation.

Publications

The works presented in this dissertation ensue from publications in international and peer-reviewed journals, conferences and workshops.

Access Control to Reflection with Object Ownership,
Camille Teruel, Stéphane Ducasse, Damien Cassou, Marcus Denker, *Proceedings of the 11th Dynamic Languages Symposium*, 2015

Propagation of Behavioral Variations with Delegation Proxies,
Camille Teruel, Erwann Wernli, Stéphane Ducasse, Oscar Nierstrasz, *Transactions on Aspect-Oriented Software Development, Volume 12*, 2015

Object Graph Isolation with Proxies,
Camille Teruel, Damien Cassou, Stéphane Ducasse, *Proceedings of the 7th Workshop on Dynamic Languages and Applications)*, 2013

Delegation Proxies: The Power of Propagation,
Erwann Wernli, Oscar Nierstrasz, Camille Teruel, Stéphane Ducasse, *Proceedings of the 13th International Conference on Modularity*, 2014

Handles: Behavior-Propagating First Class References For Dynamically-Typed Languages,
Jean-Baptiste Arnaud, Stéphane Ducasse, Marcus Denker, Camille Teruel, *Journal of Science of Computer Programming, Volume 98*, 2014

Ghost: A uniform and general-purpose proxy implementation,
Mariano Martinez Peck, Noury Bouraqadi, Luc Fabresse, Marcus Denker, Camille Teruel, *Journal of Science of Computer Programming, Volume 98*, 2014

Acknowledgments

I deeply thank Steph and Damien for their constant and unfailing support for the last three years. Their guidance and motivation helped me in all the time of writing this thesis.

Then I want to express my gratitude to the RMoD team for the welcoming and jolly atmosphere and for all the fun we have had in the last years.

Last but not the least, I thank my family and my girlfriend for their incredible resilience during all the tough times we endured. I dedicate this thesis to Jo whose tireless and enthusiastic mind always put hope in our hearts.

Contents

CHAPTER 1

Introduction

Contents

Users expect modern software to be both always available and updated on the fly. Introduction of new features or integration of bug fixes should not trouble the smooth running of an application. Developers also benefit from dynamic modifications of software, *e.g.* they can adapt code to new unanticipated situations or they can run dynamic analyses to get feedback about the behavior of running applications for monitoring, debugging and optimization purposes. Programming languages and their runtimes should thus provide developers with mechanisms that improve software adaptability.

At the same time, the increasing size and complexity of software call for applications made up of many interacting components developed by different parties. In this context, either all components are trusted or each component needs to be properly protected from other ones. For example, a plugin should not be able to manipulate the internal structures or leaks private data of its host application unrestrictedly. Likewise, a malicious library should not be able to corrupt or interfere with the operations of its clients. Programming languages should thus provide developers with mechanisms that improve software isolation, such as encapsulation and custom access control policies.

The need for dynamic adaptations that can change nearly every aspect of an application on the one hand and the need for properly confined components on the other hand pulls programming language design in two opposite directions: either more flexibility or more control. This dissertation studies this tension in the context of dynamically-typed object-oriented languages with two language mechanisms that promote software adaptability: *reflection* [Smith 1984] and *extension methods* [Bergel 2003, Akai 2012]. For both mechanisms, we demonstrate their usefulness, their harmful effect on encapsulation and propose solutions to retain their power while maintaining encapsulation.

The first mechanism, reflection, empowers programs with the ability to reason and act upon their own state and behavior. Reflection is a powerful tool for the implementation of generic code and dynamic analyses. Unfortunately, the power of reflection steams from its ability to bypass object encapsulation and other isolation mechanisms. For example, if reflection can implement access control mechanisms it also provides the ability to circumvent these mechanisms. We study reflection and its ability to alter software behavior for monitoring and access control purposes. We then show how the *proxy* mechanism can scope such alteration to various spatial and temporal extends. We finally analyse the encapsulation problems caused by reflection and propose and access control policy to reflective operations based on *object ownership* [Clarke 1998, Noble 1999, Gordon 2007] that prevent these problems.

The second mechanism, extension methods, allows packages to define methods for classes defined in other packages. Extension methods permit developers to augment the behavior of packages they do not own, thereby promoting support for unanticipated changes. Unfortunately, typical implementations make extension methods globally visible, and leads to class encapsulation problems and thus to interference between packages. Scoped version of extension methods have been proposed for dynamically-typed languages, but the implication of their semantics on class encapsulation has not been studied. We show applications and problems of globally-visible extension methods. We then present and compare four different approaches to scoped extension methods. Next we formalize and analyse and the design space for scoped extension methods. We finally propose an efficient implementation for single-dispatch languages.

This chapter introduces the context and the problem this dissertation tackles. It then summarizes our contributions and gives the outline of this dissertation.

1.1 Context

This section motivates the need for software that is both adaptable and encapsulated and then explains the tension between these two goals.

1.1.1 Software Should be Adaptable

The *waterfall* software development methodology mimics the sequential production process of physical goods: requirements are collected, the system is then designed, implemented, and finally validated against supposedly eternally relevant requirements. Continuing its hardware analogy, this methodology assumes that changing software is overly costly if not impossible. Doing so, it does the hypothesis that the requirements and the environment of the system are set once and for all and will never change. This hypothesis can only hold for simple systems but is unrealistic for the vast majority of modern software systems. Eventually, new constraints emerge, bugs are discovered, old requirements become outdated and new ones appear, etc.

Consequently, developers aim to design software that is easy to change, extend and fix. They use different techniques, such as modularization, parametrization and subclassing to introduce variation points in code. But if it is easy to predict that *some* aspects of a system will change eventually, it is difficult to forecast all the aspects that will or could need variation. Designing systems that meet a wide range of different variation points a priori is difficult and leads to contrived designs. Many anticipated variation points may turn out unnecessary while other necessary variation points were unforeseen. Anticipating too many variation points makes a system less understandable and thus more difficult to change. Instead, software systems should support *unanticipated changes*. They should be open to modification and extension not only by design but also by construction.

Also, software with high availability requirements need to be adaptable while in production. Such system should ideally be available and adaptable at the same time, and should support run-time modifications of their behavior. Unanticipated changes of running systems is of interest in a variety of situations, ranging from functional upgrades to on-the-fly debugging, dynamic optimizations, analyses and monitoring. Self-adaptive systems go a step further by actively supporting and controlling their own modification. For example, Gabriel and Goldman [Gabriel 2006] propose a vision where software maintains itself and adapts to new circumstances and users. Be a system self-adaptable or just adaptable, it needs to be both available and maintainable by supporting run-time modifications of its behavior. Supporting dynamic adaptability is the key to achieve sustainable and long-lived applications.

1.1.2 Software Components Should Be Encapsulated

The need for adaptable software imposes that programs have a large degree of freedom over their own behavior. But at the same time, large software aggregates many different and independently developed components. The design of applications thus strives for maximizing collaboration while minimizing interference between components. *Encapsulation* is the main tool to achieve this goal. By encapsulation we designate any language mechanism that allows some software entity to hide its internals from other ones. Another definition of encapsulation is specific to object-oriented programming and means to group data and related procedures together. Here we use a broader definition of encapsulation. Encapsulation can apply to the static entities of a language, like classes and packages in object-oriented languages, or to their dynamic incarnations, like objects. For example, object-level encapsulation ensures that an object can hide its internals from other objects. Class-level encapsulation ensures that a class has control over the behavior of its instances (*e.g.* unauthorized parties should not be able to change the code of a method) and over which methods can be overridden in its subclasses. We will study both kinds of encapsulation: object-level encapsulation with reflection and class-level encapsulation with extension methods.

The first use case of encapsulation is to improve modularity and maintainability. Developers use apply the *information hiding* [Parnas 1972] methodology to hide implementation aspects that are likely to change. The idea is that clients of a software component should not depend upon the specifics of its internal but upon a well defined interface instead. Since client code communicates with these components only through well defined interfaces, implementations can evolve without impacting client code as long as the interfaces remain the same.

The second use case of encapsulation is security. In the case where the multiple components of an application do not trust each other, they need to be isolated from each other. In such situations, encapsulation breaches become a security issue. The hidden part is not kept away from clients only to enhance maintainability, but also to ensure that clients cannot access to sensible information and operations. The *object-capability model* (*OCap* model) [Miller 2006], is a security paradigm that builds upon the object paradigm. The *OCap* model forces that all effects are carried out via message-passing. The *OCap* model applies naturally to object-oriented programming as the best practices of object design lead to good security properties. For example the *single-responsibility principle* and the *dependency inversion principle* [Martin 2003] correspond to the *principle of least authority* and "*no ambient authority*" [Miller 2006] respectively. As its a necessity that there is no mean for client code to access a hidden method or instance variable, encapsulation is critical to the *OCap* model.

The need for flexible and adaptable software on the one hand, and enforced encapsulation on the other hand, drives language design in opposite directions. This thesis studies this tension in the context of two mechanisms that promote software adaptability: reflection and extension methods.

1.1.3 Problem

One the one hand, software should be dynamic updatable and should support dynamic analysis for monitoring, self-optimizations an dynamic software updates. This supposes that the language offers a lot of flexibility. For this reason we set ourselves in the context of dynamically-typed languages that are typically more flexible than statically-typed ones, especially when considering advance reflective features. On the other hand the different parts of a system should be properly confined. Each software component should be protected from other, potentially malicious, components. This supposes that the language strives for more control and limits flexibility. These two goals are antagonistic and pull language design in two opposite directions. The freedom and the power required to allow dynamic adaptations are a threat to the encapsulation of separate software components.

Dynamically-typed languages are good candidates to study this tension. First, because they imposes less constraints on the structure of programs, they often offer features like advanced reflection and extension methods. These features makes the structure and behavior of programs malleable. Second, these same features and the lack of static type information makes static analyses difficult. So it is harder to statically ensure that encapsulation boundaries are respected.

This thesis studies this tension in the context of dynamically-typed object-oriented languages by focusing on two approaches for software adaptation. The first approach is reflection, and particularly behavioral intercession, is studied for its ability to dynamically adapt software behavior. Such variation of software behavior enables monitoring, dynamic analyses and even the implementation of application-specific security policies. In particular, we focus on proxies and their ability to scope behavioral variation to different temporal and spatial extents. The second approach is a mechanism known as extension methods, is studied for its ability to adapt software structure by allowing software components to customize each others. Encapsulation problems caused by each of these mechanism and solutions are presented for both approaches.

1.2 Contributions

The contribution of this dissertation are the following. We first focus on reflection and the behavioral variations that behavioral intercession enables.

- We present reflection and its various application in object-oriented languages.
- We give the formalization of an object oriented language with a simple Metaobject Protocol (MOP) and propose an implementation in the Pharo programming language. This operational semantics is then used to formalize the different language mechanisms presented in this dissertation.
- We present the proxy mechanism, that enables fined-grained behavioral intercession, different implementations and design variations.
- We show how proxies are realized in the context of our MOP and formalize this approach.

- We show how proxies can dynamically propagate behavioral variations to various temporal and spatial scopes.
- We study the encapsulation problems caused by reflection and propose an access control policy to reflective operations based on dynamic object ownership [Noble 1998, Gordon 2007] that prevents these problems.

Finally we focus on structural variations by studying and comparing different extension methods mechanisms.

- We present extension methods, their applications and the problems caused by their global visibility.
- We compare four approaches to scoped extension method mechanisms, their strength and their issues.
- We formalize the design space of scoped extension methods and compare each axis.
- We propose an implementation of scoped extension methods in Pharo, based on a variant of name mangling that incurs little performance loss.

1.3 Structure of the Dissertation

In Chapter 2, we present reflection state of the art and give a formalization of a simple class-based language with a simple Metaobject Protocol that will be used to describe the different mechanisms presented in the thesis. Chapter 3 studies proxies and their ability to scope behavioral variations to various temporal and spatial scopes. In Chapter 4, we present the encapsulation problems caused by reflection and proposes an ownership-based access control policy to control reflective abilities on a per object basis. Finally, Chapter 5 compares different scoped extension method mechanisms, the implication of their implication with encapsulation, formalize the design space and propose an implementation in Pharo.

CHAPTER 2

Reflection and Object-Centric MOP

Contents

This chapter presents reflection, its terminology and its applications. We discuss metaobject protocols (MOP) and particularly MOPs that are focused on the semantics of individual objects, that we call *object-centric* MOP. We focus on the alteration of the semantics that *implicit* enable. Finally, we give the operational semantics of a simple object-centric MOP and present an implementation in Pharo. This operational semantics will be used as a basis to formalize the different mechanisms presented in this dissertation.

Reflection is a powerful ability that allows programs to examine and modify their own structure and behavior [Smith 1984]. By allowing alteration of program interpretation, it has numerous applications.

Generic Code and Frameworks. By allowing programs to inspect their own structure reflection allows for very generic code that interacts with code whose structure and behavior are unknown beforehand.

Factor Non Functional Concerns. Like *Aspect Oriented Programming*, reflection can factor non functional concerns. AOP and reflection are in fact related domains.

Self-Adaptive Programs. Programs can monitor themselves and to adapt to different contexts, to optimize and to repair themselves.

Language Extensions. By exposing part of the language implementation to the developers, reflection provides an interface to develop language extensions, providing a language that can handles unanticipated changes.

Development tools. Reflection is also a basis for the implementation of development tools like code browsers, refactorings, debuggers.

2.1 Terminology

This section sets the reflection terminology used in this thesis and begins with a definition of *reflective language* using the terminology of Maes [Maes 1987a]. A *computational system* is a set of software and hardware components that collaborate to model a *domain* of application. The goal of a computational system is to act on or reason about its domain. When a computational system and its domain are linked in such a way that a change in one of the two is reflected on the other, they are said to be *causally-connected*. Causal connection ensures that a computational system and its domain are always synchronized. A *reflective system* is a computational system that, in addition to its primary domain, is also its own domain. The causally-connected representation that a reflective system has of itself is called its *self-representation*. A reflective system consequently has the ability to reason about itself and alter its own state and behavior. A *reflective language* is a programming language that considers reflection an essential feature. Its programs can reflect on themselves, *i.e.* each program in a reflective language is potentially a reflective system. The part of a language that implements its reflective abilities is called its *reflective architecture*.

In a reflective language, causal connection between programs and their self-representation is maintained by two processes called *reification* and *absorption* [Friedman 1984] (absorption was originally, and arguably confusingly, called reflection). Reification is the process that gives programs access to (a part of) their self-representation, by modeling the data structures of the language interpreter as first-class language entities manipulable by programs. These first-class models of the data structures of the interpreter are also called reifications. The inverse process is absorption, by which a reification is reinstalled back into the interpreter, thereby altering the future behavior of the program.

2.1.1 Taxonomy of Reflective Features

Reflective languages provide a wide range of different reflective abilities. Different axes are used to categorize these reflective features according to:

- what aspects of the program it concerns (structural or behavioral reflection),
- what the program can do with these aspects (introspection, self-modification or intercession),
- when reflective computations take place (Implicit or explicit reflection).

2.1.1.1 Structural and Behavioral Reflection

The first axis splits reflection into two categories depending on which aspects of the program is concerned. *Structural reflection* is concerned with static aspects of programs and *behavioral reflection* is concerned with dynamic aspects.

Structural reflection concerns static aspects of programs, *i.e.* their code and information that can be inferred from the code. For example, the typical structure of a class-based language is:

- packages contain classes,
- classes contain methods and instance variable definitions,
- methods contain parameters and statements.

Structural reflection in a class-based language is thus concerned with the manipulation of this hierarchical nesting of static entities. Examples of structural reflection are listing all the classes of a package or adding a method to a class.

Behavioral reflection concerns dynamic aspects of programs *i.e.* the run-time semantics of the language. This includes the interpretation of the structural part of programs and the dynamic entities of the language (like object and method invocation in an object-oriented language). In an object oriented language, behavioral reflection concerns aspects like:

- how an instance variable is read or written,
- how a method is executed,
- how a message is sent.

Of course, the distinction between structural and behavioral reflection applies outside the object paradigm. In a logic language structural reflection is concerned with predicates and clauses definition and behavioral reflection with the deduction process. In a functional language structural reflection is concerned with aspects such as data type and function definitions and behavioral reflection with aspects such as environments and continuations.

Because structural changes affects the behavior of programs, structural reflection can serve as the basis to implement behavioral reflection. Implementing behavioral reflective abilities in terms of structural ones consists in rewriting instructions that corresponds to the behavioral events of interest, like instance variable assignments. The replacing snippets of code, called *Meta-Level Interceptions* [Zimmermann 1996], explicitly invoke a corresponding reflective operation that users can redefine. This is the technique we use in the implementation of our object-centric MOP.

2.1.1.2 Introspection, Self-Modification and Intercession

The second axis categorizes reflective operations depending on the kind of access they exercise on the self-representation. Introspection corresponds to the set of reflective features that allows a program to gain knowledge about its own state, *i.e.* the self-representation is only read. Self-modification corresponds to the set of reflective features that allows a program to modify its own state, *i.e.* the self-representation is modified. Intercession corresponds to the set of reflective features that allows a program to alter its own meaning. Intercession is thus concerned with behavioral aspects. Some authors use a more general definition of intercession that also includes self-modification.

2.1.1.3 Implicit and Explicit Reflection.

The third axis categorize reflection according to when reflective computation takes place. With explicit reflection, reflective computation does not happen automatically but only when base-level code explicitly invokes a reflective operation. On the other hand, with implicit reflection, the interpretation of a program automatically and continuously triggers the invocations of reflective operations on the occurrence of specific interpretation events. Implicit reflection is a form of intercession (and as such mostly concerns behavioral aspects) but intercession is not necessarily implicit. An example of explicit intercession is 3-LISP *reflective procedure* [Smith 1984]. A reflective procedure is explicitly invoked and takes as argument a function that will be called with the current environment and continuation as arguments. Another example of explicit intercession is a specialized *eval* function with altered semantics. In this thesis, we are interested with implicit intercession, so when we talk about intercession we imply that it is implicit.

2.2 Reflection in Object-Oriented Languages

Early, reflection has been considered to fit the object paradigm well [Maes 1987b, Ferber 1989], making object-oriented languages the vehicles of choice to implement reflective architectures. Reflection has been accepted by most mainstream object-oriented programming languages (*e.g.* Ruby, Python, JavaScript, Java, Scala, Smalltalk, CLOS). The most common reflective features are limited to the introspection and self-modification. Dynamically-typed languages usually offer more advanced reflective features like limited forms of implicit reflection. This ability allows these languages to be easily extended.

The most common reflective architectures in object-oriented languages are simply based on a set of APIs. These reflective architectures often only offer introspection and limited self-modification. The reflective architecture of Java is an example. The method `getClass()` returns a reification of the class of the receiver. Java also supports a limited form of intercession thanks to dynamic proxies.

This section presents two kinds of advanced reflective architectures in object-oriented languages: mirror-based architectures [Bracha 2004] and particularly metaobject protocols [Kiczales 1991]. It focuses on metaobject protocols that support intercession on a per-object basis with a formalization and the description of an implementation.

2.2.1 Mirror-based architectures

Mirror-based architectures [Bracha 2004] are reflective architectures based on the concept of *mirror*. In a mirror-based reflective architecture, the reflective APIs are *stratified*. This means that reflective operations are not directly accessible from their targets (like the `getClass()` method in Java) but instead via special objects called *mirrors*. A mirror-based reflective architecture follows three design principles:

- Encapsulation: The implementation of reflective operations is encapsulated. It is then possible to substitute one implementation with another, *e.g.* for adapting existing development tools to a different runtime or to provide reflection on remote objects.

- Stratification: The meta-level is totally separated from the base-level. Mirrors are not accessed directly via base-level objects but instead via a *mirror factory*.

- Ontological correspondence: The reflective API describes the reflected language in its entirety and distinguishes between static and dynamic aspects of the language.

To perform reflection upon a resource, a client needs a reference over a *mirror factory*. Without access to a mirror factory, clients cannot use reflection at all. Typically, a default mirror factory creates mirrors that expose all available reflective operations. Thanks to the adherence to the Abstract Factory design pattern [Gamma 1995], mirror-based reflective architectures allows for the design custom mirror factories.

Mirror-based architectures typically only offer explicit reflection. There is the notable exception of the AmbientTalk reflective architecture [Mostinckx 2009] that enables implicit intercession thanks to the concept of *mirage*. A mirage is an object that is given an *implicit mirror* at creation. The operations of this mirror are triggered during interpretation, altering the behavior of the mirage. This makes the reflective architecture of AmbientTalk close to a metaobject protocol.

2.2.2 Metaobject Protocols

Object-oriented languages that enable implicit reflection typically does so by providing a *metaobject-protocol* or *MOP* [Kiczales 1991]. In such architecture, an interpreter manipulates *metaobjects* that represent some constructions of the language (such as objects, classes and methods). A metaobject defines a set of methods that are triggered by the interpreter when specific events occur. Such set of methods forms an interface (or protocol), hence the name “metaobject protocol”. Default metaobjects implement the default semantics of the language. Implicit reflection is carried out by substituting custom metaobjects to the default ones, thereby altering the language semantics. Programmers can replace one or more of these metaobjects with specialized ones to affect specific aspects of specific parts of their program. Such alteration is not global but affects only the base-level objects whose behavior specification involve the substituted metaobjects. Many flavors of MOPs have been considered over time: for class-based languages [Kiczales 1991, Ferber 1989, Denker 2008], for prototype-based languages [Mostinckx 2007], or with focus on specific concerns like distribution and concurrency [McAffer 1995] or simplicity [De Meuter 1998]. In this thesis we are interested in specific MOPs that allows the behavioral alteration of individual objects.

2.3 Object-Centric MOPs

We are interested in MOPs where each object behavior is described by a *distinct* metaobject: there is a one-to-one correspondence between an object and its metaobject. A metaobject is the meta-level representation of a single object. The base-level object that a metaobject controls is called its *referent*. Metaobjects are also objects. Consequently, a metaobject has a metaobject itself. This gives a *reflective tower*, a virtually infinite chain of metaobjects. In practice, metaobjects that describe the normal semantics are created lazily.

This kind of MOP has been described in detail in the literature [Maes 1987b, Ferber 1989,Mostinckx 2009]. We borrow the terminology of Ressia [Ressia 2012] and refer to these MOPs as *object-centric MOPs*. In a class-based language with an object-centric MOP, the metaobject of an object is distinct from its class: the metaobject is concerned about behavioral aspects of the object and the class about structural ones. The next section gives a formalization of an object-centric MOP and then describes an implementation in Pharo Smalltalk.

2.3.1 Semantics

We propose MOPLITE, a small-step semantics for an object-centric MOP. To our knowledge, this is the first operational semantics that describes a MOP. MOPLITE does not feature structural reflection but can be extended to support it. Here, we are interested in behavioral reflection, so we leave such extension as future work. This operational semantics will be used to formalize the different mechanisms presented in this thesis: proxies, ownership and extension methods. The formalized MOP supports the explicit invocation and the implicit interception of the following operations:

Instance variable read: via the message `get(iv)` that corresponds to reading the instance variable named `iv`.

Instance variable write: via the message `set(iv, obj)` that corresponds to changing the value of the instance variable named `iv` to `obj`.

Message reception: via the message `receive(m,args,c)` that corresponds to receiving a message `m` with arguments `args` where the method lookup algorithm starts in the class `c` (typically the class of the receiver except for super-sends).

Message sending: via the message `send(obj,m,arg1s,c)` that corresponds to sending a message `m` with arguments `args` to the object `obj` where the method lookup algorithm starts in the class `c` (typically the class of `obj` except for super-sends).

2.3.1.1 Syntax

The syntax of MOPLITE is given in Figure 2.1. Elements of the surface syntax are in bold. A program consists of a sequence of class definitions followed by an initial "main" expression. A class definition consists of the class identifier (*i.e.* its name), the superclass identifier, a list of instance variables followed by a list of methods. A method has an identifier, a list of formal parameters, and a body expression. An expression is either an identifier, an assignment, a reference to the current object **`self`**, a message send, a super-send, an instantiation, a local variable or a sequence of expressions or an access to a metaobject.

$$\begin{array}{rcl}
p \in \mathcal{P} & ::= & \mathit{cls}^* \; \mathit{exp} \\
\mathit{cls} \in \mathcal{C} & ::= & \textbf{class} \; \mathit{id} \; \textbf{extends} \; \mathit{id} \; \textbf{\{} \; \mathit{id}^* \; \mathit{meth}^* \; \textbf{\}} \\
\mathit{meth} \in \mathcal{M} & ::= & \mathit{id}\,\textbf{(}\mathit{id}^*\textbf{)} \;\; \textbf{\{} \; \mathit{exp} \; \textbf{\}} \\
\mathit{exp} \in \mathcal{E} & ::= & \mathit{id} \mid \mathit{id} \; \textbf{:=} \; \mathit{exp} \mid \textbf{self} \mid \mathit{exp}\,\textbf{.}\,\mathit{id}\,\textbf{(}\mathit{exp}^*\textbf{)} \\
 & & \mid \textbf{super}\,\textbf{.}\,\mathit{id}\,\textbf{(}\mathit{exp}^*\textbf{)} \mid \mathit{id}\,\textbf{.new(}\mathit{exp}^*\textbf{)} \\
 & & \mid \textbf{let} \; \mathit{id} \; \textbf{=} \; \mathit{exp} \; \textbf{in} \; \mathit{exp} \mid \mathit{exp} \; \textbf{;} \; \mathit{exp} \\
 & & \mid \mathit{exp}\,\textbf{.meta} \\
\mathit{id} \in \mathcal{I} & &
\end{array}$$

Figure 2.1: Syntax of MOPLITE

2.3.1.2 Core Classes

Before the evaluation of a program, the code of two core classes is included into the code of that program. These two classes do not strictly respect the above syntax and they are the only ones that are allowed to do so.

The first core class is `Object` from which all classes inherit. This class only contains a method `init` that does nothing. This class has no superclass so its declaration cannot respect the syntax: the header of its declaration says "**`extends nil`**".

```
class Object extends nil {
  init() { self }
}
```

The second class is DefaultMO is the class of default metaobjects that define the normal object semantics. This class defines one instance variable referent (that points to the base-object of the metaobject) and contains five methods. The body of each method consist of a call to a *primitive*. Primitive calls are not part of the syntax: DefaultMO is the only class that can contain primitives. Other calls to primitives are generated during execution. Each of the first four methods correspond to one of the four interceptable operations: receiving messages (method receive), sending messages (method send), reading instance variables (method get) and writing instance variables (method set). The fifth method meta changes the metaobject of the referent.

```
class DefaultMO extends Object {
  referent
  receive(id,args,cls) { RECEIVE(referent,id,args,cls) }
  send(rcv,id,args,cls) { SEND(rcv,id,args,cls) }
  get(iv) { GET(referent,iv) }
  set(iv,obj) { SET(referent,iv,obj) }
  meta(mobj) { SETMO(referent,mobj) }
}
```

2.3.1.3 Well-Formed Programs

The syntax gives the basic structure of programs. But once the code of the two core classes Object and DefaultMO is included into a programs, they must respect additional constraints.

- Name conflicts are forbidden.
  - A program cannot define two classes with the same identifier.
  - A class cannot define two methods that have the same identifier and the same number of arguments.
  - Instance variable identifiers are unique within the hierarchy of a class, *i.e.* a class cannot define an instance variable if it inherits from a class that already defines an instance variable with the same name.

  - The identifier of a parameter cannot be the same than the one of an instance variable of the class hierarchy, or of a class of the program.
  - The identifier of a local variable cannot be the same than the one of an outer local variable, of a parameter of the method, of an instance variable of the class hierarchy, or of a class of the program.

- The superclass identifier given in a class definition must correspond to another class defined by the program.
- Inheritance must be acyclic, *i.e.* a class cannot inherit from itself directly nor indirectly.
- In methods, an identifier id must correspond to a local variable, a parameter or an instance-variable of the hierarchy of the class.
- Instantiation expressions (id **.new(**exp^***)**) are an exception to previous constraint: the identifier id must correspond to a class of the program.
- Finally, the identifier id of an assignment id **:=** exp must correspond to an instance variable.

A program that does not respect these constrains is not *well-formed* and cannot be evaluated.

2.3.1.4 Functions and Relations

Several functions and relations are used to query the relationships of the elements defined by a given program p.

- The partial function $\text{class}_p\colon \mathcal{I} \rightharpoonup \mathcal{C}$ retrieves classes: $\text{class}_p(\texttt{C})$ gives the class named `C` defined in p or is undefined if p defines no such class.
- The partial function $\text{method}_p\colon \mathcal{I}\times\mathbb{N}\times\mathcal{C} \rightharpoonup \mathcal{M}$ retrieves methods: $\text{method}_p(\texttt{m}, n, c)$ gives the method named `m` with n parameters that is defined in class c or is undefined if c defines no such method.
- The partial function $\text{super}_p\colon \mathcal{C} \rightharpoonup \mathcal{C}$ maps a class to its direct superclass, *i.e.* $\text{super}_p(a) = b$ if $a =$ **class** `A` **extends** `B` **{** ... **}** and $\text{class}_p(\texttt{B}) = b$.

This function is defined for all classes expect `Object`, which is the only class that has no superclass.

- The relation $\lhd_p \subset \mathcal{C} \times \mathcal{C}$ associates a class with classes defined upper in its hierarchy, *i.e.* $a \lhd_p b$ reads "a inherits from b". This relation is the transitive reflexive closure of super_p ($\lhd_p = \mathrm{super}_p^*$).

- The relation $\sqsubset_p \subset \mathcal{I} \times \mathcal{C}$ associates an instance variable identifier with classes than define an instance variable with that name, *i.e.* $\mathtt{iv} \sqsubset_p \mathit{cls}$ reads "cls defines instance variable `iv`".

- The relation $\sqsubset_p^{hrc} \subset \mathcal{I} \times \mathcal{C}$ associates an instance variable identifier with classes that have an instance variable with that name defined in their hierarchy, *i.e.* $\mathtt{iv} \sqsubset_p^{hrc} \mathit{cls}$ reads "`iv` is defined in the hierarchy of cls". This relation is the composition of the inverse of $\lhd_p$ with $\sqsubset_p$ ($\sqsubset_p^{hrc} = \lhd_p{}^{-1} \circ \sqsubset_p$).

- The partial function $\mathrm{lookup}_p \colon \mathcal{I} \times \mathbb{N} \times \mathcal{C} \rightharpoonup \mathcal{M} \times \mathcal{C}$ performs the method lookup algorithm: $\mathrm{lookup}_p(\mathtt{m}, n, c)$ gives the first method named `m` with n parameters defined in the hierarchy of c and the class where that method is defined. This function is undefined if not such method is defined in the hierarchy of c. This function is defined as follows:

$$\mathrm{lookup}_p(\mathit{id}, n, \mathit{cls}) =$$
$$\begin{cases} \langle \mathrm{method}_p(\mathit{id}, n, \mathit{cls}), \mathit{cls} \rangle & \text{if } \mathrm{method}_p(\mathit{id}, n, \mathit{cls}) \text{ is defined} \\ \mathrm{lookup}_p(\mathit{id}, n, \mathrm{super}_p(\mathit{cls})) & \text{if } \mathrm{super}_p(\mathit{cls}) \text{ defined } (\textit{i.e. } \mathit{cls} \neq \mathrm{class}_p(\mathtt{Object})) \\ \text{undefined} & \text{otherwise } (\textit{i.e. } \mathit{cls} = \mathrm{class}_p(\mathtt{Object})) \end{cases}$$

2.3.1.5 Abstract Machine

Programs are executed on an abstract machine whose *configurations* consist of a *call stack* and a *store*.

The call stack represents the state of the unique thread of execution of the abstract machine: $\left\lfloor \begin{smallmatrix} f_2 \\ f_1 \end{smallmatrix} \right\rfloor$ denotes a stack made of two stack frames f_1 and f_2 (oldest frames at bottom and newest at top). Each stack frame is a triplet $\langle \mathit{exp}, \mathit{adr}, \mathit{cls} \rangle \in \mathcal{E} \times \mathcal{A} \times \mathcal{C}$ where exp is the expression under evaluation, adr is the address of the current object (**`self`**) and cls is the name of the current class (*i.e.* the class that defines the method under evaluation). The modelization of the call stack is not capital for expressing the semantics of our MOP. We will use it for the formalization of context-sensitive method lookup algorithms of scoped extension methods in Chapter 5. Also, this modelization of the call stack makes our formalization closer to an actual implementation.

The store is a partial function from addresses to objects: $S : \mathcal{A} \rightharpoonup \mathcal{O}$. We note $S[\ adr \mapsto obj\ ]$ the store S updated with the address adr pointing to the object obj. An object is a triplet $\langle cls, adr_{mo}, ivs\rangle \in \mathcal{C} \times \mathcal{A} \times (\mathcal{I} \rightharpoonup \mathcal{A})$ where cls is its class, adr_{mo} is the address of the metaobject and ivs is a mapping from instance variable identifiers to addresses.

The reduction function $\hookrightarrow$ formalizes the evaluation of expressions to addresses. This function is expressed as a set of reduction rules over the configurations of the abstract machine. The initial configuration of a program consists of an expression evaluated in the context of the *nil* object (whose address is noted **`nil`**). The nil object is simply a specific instance of `Object` that is preallocated on the store at start. That is, for a program $P = \ldots\ \ exp$, the initial configuration is: $\left\langle \left\lfloor\ exp, \mathbf{nil}, \mathtt{Object}\ \right\rfloor, \{\mathbf{nil} \mapsto \langle \mathtt{Object}, \mathbf{nil}, \{\}\rangle\}\right\rangle$.

2.3.1.6 Evaluation Contexts

In the style of Felleisen and Hieb [Felleisen 1992], reduction rules specify the evaluation of expressions within an *evaluation context* E. An evaluation context is an expression with a "hole" (noted "[ ]") at the point where the next evaluation step takes place: $E[exp]$ denotes the evaluation context E with the hole filled by exp. Figure 2.2 gives the syntax of the evaluation contexts of MOPLITE. This syntax determines the order of evaluation. For example, the order of evaluation for message sends is specified by the cases $E\,\mathtt{.}\,id\,\mathtt{(}exp^*\mathtt{)}$ and $adr\,\mathtt{.}\,id\,\mathtt{(}adr^*, E, exp^*\mathtt{)}$. In the first case ($E\,\mathtt{.}\,id\,\mathtt{(}exp^*\mathtt{)}$), the hole is located in the receiver, thereby forcing the evaluation of the receiver subexpression until it reduces to an address. In the second case ($adr\,\mathtt{.}\,id\,\mathtt{(}adr^*, E, exp^*\mathtt{)}$) the receiver is already reduced to an address adr and the hole is located between an arbitrary number of reduced arguments and an arbitrary number of non-reduced arguments. Together, these two cases thus specify that the receiver of a message send is evaluated first and then the arguments are evaluated from left to right.

$$\begin{array}{rcl} E & ::= & [\ ] \mid id\ \texttt{:=}\ E \mid E\,\texttt{.}\,id\,\texttt{(}exp^*\texttt{)} \mid adr\,\texttt{.}\,id\,\texttt{(}adr^*, E, exp^*\texttt{)} \\ & & \mid \texttt{super.}\,id\,\texttt{(}adr^*, E, exp^*\texttt{)} \mid id\,\texttt{.new(}adr^*, E, exp^*\texttt{)} \\ & & \mid \texttt{let}\ id\ \texttt{=}\ E\ \texttt{in}\ exp \mid E\ \texttt{;}\ exp \end{array}$$

Figure 2.2: Evaluation Contexts

2.3.1.7 Reduction rules

The reduction function $\hookrightarrow$ specifies the evaluation of expressions. This function is expressed as a set of reduction rules. A reduction rule describes one evaluation step: it gives the next configuration of the abstract machine from the current one. Since the evaluation of program is deterministic, only one rule is applicable at a time. If no rule matches the current configuration the evaluation is stuck, representing an error.

Self references. The rule [*self*] simply states that **`self`** reduces to the current object address.

$$\begin{array}{ll} & \left\langle \left\lfloor \begin{array}{c} \langle E[\ \texttt{self}\ ], adr, cls\rangle \\ \vdots \end{array} \right\rfloor, S \right\rangle \\ \hookrightarrow & \left\langle \left\lfloor \begin{array}{c} \langle E[\ adr\ ], adr, cls\rangle \\ \vdots \end{array} \right\rfloor, S \right\rangle \end{array} \quad [\ self\ ]$$

Sequences of Expressions. The rule [*seq*] describes sequence of expressions. Once reduced to an address, the first expression of the sequence is simply dropped and evaluation continues with the second.

$$\begin{array}{ll} & \left\langle \left\lfloor \begin{array}{c} \langle E[\ adr'\ \texttt{;}\ exp\ ], adr, cls\rangle \\ \vdots \end{array} \right\rfloor, S \right\rangle \\ \hookrightarrow & \left\langle \left\lfloor \begin{array}{c} \langle E[\ exp\ ], adr, cls\rangle \\ \vdots \end{array} \right\rfloor, S \right\rangle \end{array} \quad [\ seq\ ]$$

Local Variables. The rule [*let*] declares a local variable. The expression $\texttt{let}\ id\ \texttt{=}\ adr_{loc}\ \texttt{in}\ exp$ replaces each occurrence of the local variable identifier id in the subexpression exp with the evaluated address adr_{loc}.

$$\begin{array}{ll} \left\langle \left\lfloor \begin{array}{c} \langle E[\ \texttt{let}\ id\ \texttt{=}\ adr_{loc}\ \texttt{in}\ exp\ ], adr, cls\rangle \\ \vdots \end{array} \right\rfloor, S \right\rangle & \\ \hookrightarrow \left\langle \left\lfloor \begin{array}{c} \langle E[\ exp[adr_{loc}/id]\ ], adr, cls\rangle \\ \vdots \end{array} \right\rfloor, S \right\rangle & [\,\textit{let}\,] \end{array}$$

Instantiation. The rule [*new*] gives the semantics of instantiation. An instantiation expression reduces to a new object address adr_{new} (*i.e.* not in the domain of the store S) and updates the store to map this address to a new object. The second element of the new object triplet is **nil** to show that this object has a default metaobject. This reflects that default metaobjects are created lazily: only metaobjects that gives special semantics are consistently stored. Otherwise the store would need to be infinite since each metaobject should also have its own metaobject. The third element maps the identifier of each instance variable defined in the class hierarchy to **nil**. This new object is then initialized with the message `init` and the arguments given in the instantiation expression.

$$\begin{array}{ll} \left\langle \left\lfloor \begin{array}{c} \langle E[\ cls\,\texttt{.new(}adr^*\texttt{)}\ ], adr, cls\rangle \\ \vdots \end{array} \right\rfloor, S \right\rangle & \\ \hookrightarrow \left\langle \left\lfloor \begin{array}{c} \langle E[\ adr_{new}\,\texttt{.init(}adr^*\texttt{)}\ \texttt{;}\ adr_{new}\ ], adr, cls\rangle \\ \vdots \end{array} \right\rfloor, S' \right\rangle & [\,\textit{new}\,] \\ \text{where} \begin{array}{l} S' = S[adr_{new} \mapsto \langle cls, \texttt{nil}, \{id_{iv} \mapsto S(\texttt{nil}) \mid \forall id_{iv}.\ id_{iv} \sqsubset_p^{hrc} cls\}\rangle] \\ adr_{new} \notin \mathrm{dom}(S) \end{array} & \end{array}$$

Metaobject Access. The rule [*set-mo-prim*] gives the semantics of the SETMO primitive that changes the metaobject of an object. This primitive is only used in the body of the `meta` method of the class `DefaultMO`. The metaobject of the target object changes to the specified one (here pointed by the address adr_{mo}) and the instance variable `referent` of new metaobject is updated to point to the target object address.

$$\left\langle \begin{bmatrix} \langle E[\ \textsc{setmo}(adr_{targ}, adr_{mo})\ ], adr, cls \rangle \\ \vdots \end{bmatrix}, S \right\rangle$$

$$\hookrightarrow \left\langle \begin{bmatrix} \langle E[\ adr_{mo}\ ], adr, cls \rangle \\ \vdots \end{bmatrix}, S' \right\rangle \qquad [\ \textit{set-mo-prim}\ ]$$

$$\text{where } \begin{array}{l} S(adr_{targ}) = \langle cls_{targ}, ..., ivs_{targ} \rangle \\ S(adr_{mo}) = \langle cls_{mo}, adr_{mo2}, ivs_{mo} \rangle \\ S' = S \begin{bmatrix} adr_{targ} \mapsto \langle cls_{targ}, adr_{mo}, ivs_{targ} \rangle\ , \\ adr_{mo} \mapsto \langle cls_{mo}, adr_{mo2}, ivs_{mo}[\texttt{referent} \mapsto adr_{targ}] \rangle \end{bmatrix} \end{array}$$

Accesses to metaobjects are described by the rules [*get-def-mo*] and [*get-mo*]. The rule [*get-def-mo*] is responsible for the lazy creation of default metaobjects (instances of `DefaultMO`) using the SETMO primitive.

$$\left\langle \begin{bmatrix} \langle E[\ adr_{targ}\,\texttt{.meta}\ ], adr, cls \rangle \\ \vdots \end{bmatrix}, S \right\rangle$$

$$\hookrightarrow \left\langle \begin{bmatrix} \langle E[\ \textsc{setmo}(adr_{targ}, \texttt{DefaultMO.new()})\ ], adr, cls \rangle \\ \vdots \end{bmatrix}, S \right\rangle [\ \textit{get-def-mo}\ ]$$

$$\text{where } S(adr_{targ}) = \langle ..., \texttt{nil}, ... \rangle$$

The rule [*get-mo*] returns the metaobject of objects that have a non-nil metaobject, *i.e.* the second element of the target object triplet.

$$\left\langle \begin{bmatrix} \langle E[\ adr_{targ}\,\texttt{.meta}\ ], adr, cls \rangle \\ \vdots \end{bmatrix}, S \right\rangle$$

$$\hookrightarrow \left\langle \begin{bmatrix} \langle E[\ adr_{mo}\ ], adr, cls \rangle \\ \vdots \end{bmatrix}, S \right\rangle \qquad [\ \textit{get-mo}\ ]$$

$$\text{where } \begin{array}{l} S(adr_{targ}) = \langle ..., adr_{mo}, ... \rangle \\ adr_{mo} \neq \texttt{nil} \end{array}$$

Message Sending and Message Reception. Message sending is specified by the rules [*send*], [*super*] and by the SEND and RECEIVE primitives. The rule [*send*] reduces to the SEND primitive.

$$\left\langle \left\lfloor \begin{array}{c} \langle E[\ adr_{rcv}\,\texttt{.}\,id_m\,\texttt{(}adr^*_{args}\texttt{)}\ ],\ adr,\ cls\rangle \\ \vdots \end{array} \right\rfloor, S \right\rangle$$
$$\hookrightarrow \left\langle \left\lfloor \begin{array}{c} \langle E[\ \textsc{send}(adr_{rcv}, id_m, adr^*_{args}, cls_{rcv})\ ],\ adr,\ cls\rangle \\ \vdots \end{array} \right\rfloor, S \right\rangle [\,\mathit{send}\,]$$

where $S(adr_{rcv}) = \langle cls_{rcv}, ..., ...\rangle$

Super-sends are almost identical regular message sends except that the lookup class is the superclass of the current class (the class that defines the method under evaluation).

$$\left\langle \left\lfloor \begin{array}{c} \langle E[\ \texttt{super}\,\texttt{.}\,id_m\,\texttt{(}adr^*_{args}\texttt{)}\ ],\ adr,\ cls\rangle \\ \vdots \end{array} \right\rfloor, S \right\rangle$$
$$\hookrightarrow \left\langle \left\lfloor \begin{array}{c} \langle E[\ \textsc{send}(adr, id_m, adr^*_{args}, cls_{super})\ ],\ adr,\ cls\rangle \\ \vdots \end{array} \right\rfloor, S \right\rangle [\,\mathit{super}\,]$$

where $\mathrm{super}_p(cls) = cls_{super}$

The rules [*send-prim*] and [*send-mo-prim*] defines the SEND primitive. The rule [*send-prim*] sends a message to an object whose metaobject is not yet created. The rule pushes a new stack frame onto the call stack with the body of the method found with the function lookup_p, starting the method lookup in the specified class cls_{lookup}. Occurrences of the method parameters identifiers are substituted with the message arguments.

$$\left\langle \left\lfloor \begin{array}{c} \langle E[\ \textsc{send}(adr_{rcv}, id_m, adr^*_{args}, cls_{lookup})\ ],\ adr,\ cls\rangle \\ \vdots \end{array} \right\rfloor, S \right\rangle$$
$$\hookrightarrow \left\langle \left\lfloor \begin{array}{c} \langle exp_{body}[adr^*_{args}/id^*_{params}],\ adr_{rcv},\ cls_m\rangle \\ \langle E[\ \textsc{send}(adr_{rcv}, id_m, adr^*_{args}, cls_{lookup})\ ],\ adr,\ cls\rangle \\ \vdots \end{array} \right\rfloor, S \right\rangle [\,\mathit{send\text{-}prim}\,]$$

where $\begin{array}{l} \mathrm{lookup}_p(id_m, |adr^*_{args}|, cls_{lookup}) = \langle meth, cls_m\rangle \\ meth = id_m\,\texttt{(}id^*_{params}\texttt{)}\ \ \texttt{\{}\ exp_{body}\ \texttt{\}} \end{array}$

The rule [*mo-send-prim*] sends a message to an object that has a non-nil metaobject: it reduces to another call to the SEND primitive to invoke the `receive` method of the metaobject with the method identifier the message arguments and the lookup class. Using the primitive SEND instead of a normal message send avoids infinite recursion: if instead this rule had reduced to the message send

adr_{mo}`.receive(`id_m, adr^*_{args}, cls_{lookup}`)`, the rule [*send*] would be triggered again, then the rule [*mo-send-prim*] again, etc.

$$\begin{array}{l} \left\langle \left[\begin{array}{c} \langle E[\ \text{SEND}(adr_{rcv}, id_m, adr^*_{args}, cls_{lookup})\ ], adr, cls \rangle \\ \vdots \end{array} \right], S \right\rangle \\ \hookrightarrow \left\langle \left[\begin{array}{c} \langle E[\ \text{SEND}(adr_{mo}, \texttt{receive}, (id_m, adr^*_{args}, cls_{lookup}), cls_{mo})\ ], adr, cls \rangle \\ \vdots \end{array} \right], S \right\rangle \ [\ \textit{mo-send-prim}\ ] \end{array}$$

$$\text{where} \begin{array}{l} S(adr_{rcv}) = \langle ..., adr_{mo}, ... \rangle \\ S(adr_{mo}) = \langle cls_{mo}, ..., ... \rangle \ adr_{mo} \neq \mathbf{nil} \end{array}$$

Finally, the primitive RECEIVE is used only in the body of the `receive` method of the default metaobject class `DefaultMO`. The sole purpose of this primitive is to avoid an infinite regression. Indeed, using the SEND primitive in the `receive` method of `DefaultMO` would invoke that same method again and again indefinitely. Consequently, apart from the name of the primitive, the rule [*receive-prim*] is identical to the rule [*send-prim*].

$$\begin{array}{l} \left\langle \left[\begin{array}{c} \langle E[\ \text{RECEIVE}(adr_{rcv}, id_m, adr^*_{args}, cls_{lookup})\ ], adr, cls \rangle \\ \vdots \end{array} \right], S \right\rangle \\ \hookrightarrow \left\langle \left[\begin{array}{c} \langle exp_{body}[adr^*_{args}/id^*_{params}], adr_{rcv}, cls_m \rangle \\ \langle E[\ \text{RECEIVE}(adr_{rcv}, id_m, adr^*_{args}, cls_{lookup})\ ], adr, cls \rangle \\ \vdots \end{array} \right], S \right\rangle \ [\ \textit{receive-prim}\ ] \end{array}$$

$$\text{where} \begin{array}{l} \text{lookup}_p(id_m, |adr^*_{args}|, cls_{lookup}) = \langle meth, cls_m \rangle \\ meth = id_m\,\texttt{(}id^*_{params}\texttt{)}\ \ \texttt{\{}\ exp_{body}\ \texttt{\}} \end{array}$$

Return. The rule [*return*] shows that once the evaluation of the current method has reduced to an object address adr_{ret}, the current stack frame is popped and the object address replaces the subexpression exp in the evaluation context E of the stack frame below.

$$\begin{array}{l} \left\langle \left[\begin{array}{c} \langle adr_{ret}, adr_2, cls_2 \rangle \\ \langle E[\ exp\ ], adr_1, cls_1 \rangle \\ \vdots \end{array} \right], S \right\rangle \\ \hookrightarrow \left\langle \left[\begin{array}{c} \langle E[adr_{ret}], adr_1, cls_1 \rangle \\ \vdots \end{array} \right], S \right\rangle \end{array} \quad [\ \textit{return}\ ]$$

Instance Variable Reads. Reading instance variables is specified by the rules [*get*], [*get-prim*] and [*mo-get*]. The rule [*get*] is used for objects with a nil metaobject: it reduces to the GET primitive.

$$\left\langle \begin{bmatrix} \langle E[\ id_{iv}\ ], adr, cls\rangle \\ \vdots \end{bmatrix}, S \right\rangle$$

$$\hookrightarrow \left\langle \begin{bmatrix} \langle E[\ \text{GET}(adr, id_{iv})\ ], adr, cls\rangle \\ \vdots \end{bmatrix}, S \right\rangle \quad [\ \textit{get}\ ]$$

$$\text{where } S(adr) = \langle ..., \mathbf{nil}, ...\rangle$$

The rule [*get-prim*] defines the GET primitive: it reduces to the object address adr_{iv} pointed by the instance variable identifier id_{iv} in the instance variable mapping ivs of the target object adr_{targ}. The target object is the receiver ($adr_{targ} = adr$) except when the GET primitive is invoked from the method `get` of the default metaobject class `DefaultMO`.

$$\left\langle \begin{bmatrix} \langle E[\ \text{GET}(adr_{targ}, id_{iv})\ ], adr, cls\rangle \\ \vdots \end{bmatrix}, S \right\rangle$$

$$\hookrightarrow \left\langle \begin{bmatrix} \langle E[\ adr_{iv}\ ], adr, cls\rangle \\ \vdots \end{bmatrix}, S \right\rangle \qquad [\ \textit{get-prim}\ ]$$

$$\text{where } \begin{array}{l} S(adr_{targ}) = \langle ..., ..., ivs\rangle \\ ivs(id_{iv}) = adr_{iv} \end{array}$$

The rule [*mo-get*] is used for objects with a non-nil metaobject: it sends a `get` message to the metaobject using the SEND primitive (to avoid infinite recursion like in rule [*send-mo-prim*]).

$$\left\langle \begin{bmatrix} \langle E[\ id_{iv}\ ], adr, cls\rangle \\ \vdots \end{bmatrix}, S \right\rangle$$

$$\hookrightarrow \left\langle \begin{bmatrix} \langle E[\ \text{SEND}(adr_{mo}, \texttt{get}, (id_{iv}), cls_{mo})\ ], adr, cls\rangle \\ \vdots \end{bmatrix}, S \right\rangle \quad [\ \textit{mo-get}\ ]$$

$$\text{where } \begin{array}{l} S(adr) = \langle ..., adr_{mo}, ...\rangle \\ S(adr_{mo}) = \langle cls_{mo}, ..., ...\rangle \\ adr_{mo} \neq \mathbf{nil} \end{array}$$

Instance Variable Writes. Writing instance variables is specified by the rules [*set*], [*set-prim*] and [*mo-set*]. The rule [*set*] is used for objects with a nil metaobject: it reduces to the SET primitive.

$$
\begin{aligned}
&\left\langle \begin{bmatrix} \langle E[\ id_{iv}\ \texttt{:=}\ adr_{val}\ ], id, cls\rangle \\ \vdots \end{bmatrix}, S \right\rangle \\
\hookrightarrow &\left\langle \begin{bmatrix} \langle E[\ \textsc{set}(id_{iv}, adr_{val}, id)\ ], id, cls\rangle \\ \vdots \end{bmatrix}, S \right\rangle [\ \textit{set}\ ] \\
\text{where}\ & S(adr) = \langle ..., \texttt{nil}, ...\rangle
\end{aligned}
$$

The rule [*set-prim*] defines the SET primitive: it updates to the object address pointed by the instance variable identifier id_{iv} to the new value adr_{val} in the instance variable mapping ivs of the target object adr_{targ} and reduces to adr_{val}. Like with the GET primitive, the target object is the receiver ($adr_{target} = adr$) except when the SET primitive is invoked from `DefaultMO`.

$$
\begin{aligned}
&\left\langle \begin{bmatrix} \langle E[\ \textsc{set}(adr_{targ}, id_{iv}, adr_{val})\ ], adr, cls\rangle \\ \vdots \end{bmatrix}, S \right\rangle \\
\hookrightarrow &\left\langle \begin{bmatrix} \langle E[adr_{val}], adr, cls\rangle \\ \vdots \end{bmatrix}, S' \right\rangle \qquad [\ \textit{set-prim}\ ] \\
\text{where}\ & \begin{array}{l} S(adr_{targ}) = \langle cls_{targ}, adr_{mo}, ivs\rangle \\ S' = S[adr_{targ} \mapsto \langle cls_{targ}, adr_{mo}, ivs[id_{iv} \mapsto adr_{val}]\rangle] \end{array}
\end{aligned}
$$

The rule [*mo-set*] is used for objects with a non-nil metaobject: it sends a `set` message to the metaobject using the SEND primitive.

$$
\begin{aligned}
&\left\langle \begin{bmatrix} \langle E[\ id_{iv} adr_{val}\ \texttt{:=}\ ], adr, cls\rangle \\ \vdots \end{bmatrix}, S \right\rangle \\
\hookrightarrow &\left\langle \begin{bmatrix} \langle E[\ \textsc{send}(adr_{mo}, \texttt{set}, (id_{iv}, adr_{val}), cls_{mo})\ ], adr, cls\rangle \\ \vdots \end{bmatrix}, S \right\rangle [\ \textit{mo-set}\ ] \\
\text{where}\ & \begin{array}{l} S(adr) = \langle ..., adr_{mo}, ...\rangle \\ S(adr_{mo}) = \langle cls_{mo}, ..., ...\rangle \\ adr_{mo} \neq \texttt{nil} \end{array}
\end{aligned}
$$

This concludes the presentation of the reduction rules and the semantics of MOPLITE. The next section describes an implementation of our MOP in Pharo Smalltalk.

2.3.2 Implementation

This describes an implementation of a object-centric MOP in Pharo Smalltalk, following the semantics of MOPLite. It allows the interception of the same operations than MOPLite at the granularity of objects: reading and writing instance-variables and receiving and sending message. An overview of the implementation is given in Figure 2.3.

The cost of reflection is often regarded as problematic. We believe the power of reflection outweighs its performance cost. Moreover performances of reflective languages can be largely improved thanks to optimization techniques like meta-tracing, partial evaluation [Marr 2015] and adaptive recompilers [Hölzle 1995]. Also the performance penalty only needs to be paid where and when reflection is effectively used: this is called *partial reflection* [Tanter 2003]. Our implementation performs partial reflection: only objects with a non-default metaobject pay the performance overhead of reflection.

2.3.2.1 Metaobjects

Like it is expressed in the semantics of MOPLite, metaobjects are created on demand. Indeed, metaobjects like other objects have their own metaobject. This gives a virtually infinite chain of metaobjects that cannot be allocated on finite memory. Since an object with a default metaobject has a default semantics, the code of its metaobject doesn't need to be really executed. So default metaobjects are created only when an object receives the message `#meta`. The default metaobjects that are created are cached to insure that different invocations of `#meta` yield the same metaobject. Objects with a non-default metaobject points directly to their metaobject.

2.3.2.2 Code generation

Our implementation makes a distinction between two concepts of classes: *conceptual classes* and *implementation classes*. This distinction remains irrelevant from the developer point of view. Conceptual classes are classes written by developers. Implementation classes are either conceptual classes or synthetic classes. The implementation class of objects with a default metaobject is simply their conceptual class. The implementation class of an object with a non-default metaobject is a synthetic class, hidden form the developers, that contains generated methods.

Interceptions are implemented by code rewriting. The code of methods is transformed to insert *meta-level interceptions* (MLI) into new generated methods that are installed into synthetic classes. For example the `#increment` method of a class `Counter` is transformed as follow.

```
Counter>>increment
   ↑ value := value + 1

(synthetic class)>>increment
   ↑ meta receive: (Message selector: #increment arguments #())

(synthetic class)>>_increment
   ↑ meta write: #value to: (meta read: #value) + 1
```

The first generated method sends `#receive:` to the metaobject of the receiver with a message reification as argument. The second generated method, named `#'_increment'`, is private and contains the transformed code of the original method. It is invoked when the special metaobject code, such as the following, performs the original operation with a super-send to the default metaobject class.

```
MySpecialMO>>receive: aMessage
   | return |
   ... do something before ...
   return := super receive: aMessage.
   ... do something after ...
   ↑  return
```

Lazy code generation. Generated methods are compiled lazily by intercepting message reception with a second mechanism.

A special class with no method dictionary, that we call the *trap class* (depicted with a cross in Figure 2.3), is the ancestor of all synthetic classes. In this situation, the method lookup fails and the VM sends the message `cannotInterpret:` with the message as argument but starting the method lookup in the superclass of the trap class, `ActiveObject`. The default implementation of `cannotInterpret:` in the class `ProtoObject` raises an error. By redefining this method in the superclass of the trap class, we can intercept all messages. This technique comes from the

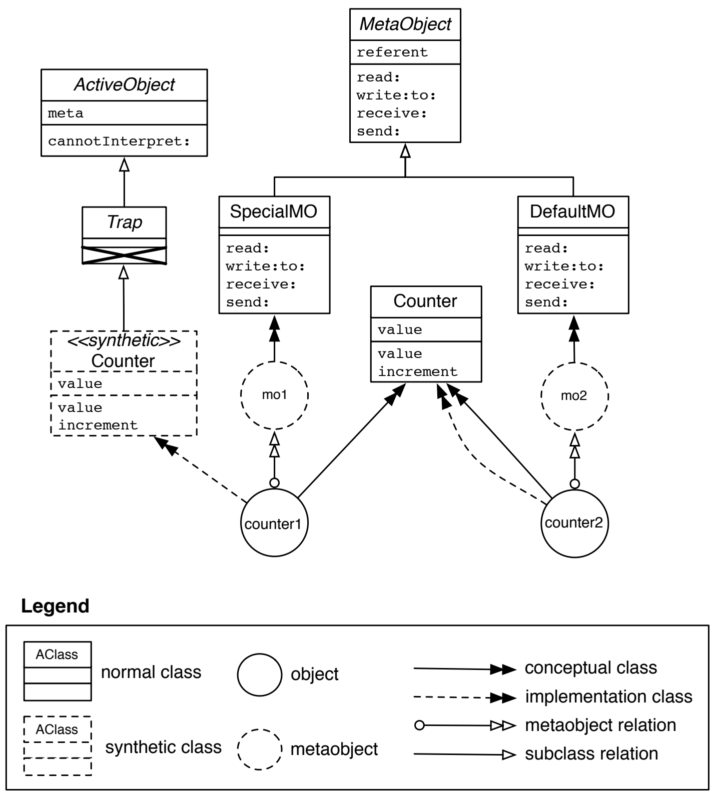


Figure 2.3: Difference between conceptual and implementation classes. The object `counter1` has a non-default metaobject so its implementation class is a synthetic class whose methods are the one of its conceptual class rewritten with MLIs. The object `counter2` has a normal metaobject so its implementation is the same as its conceptual class

proxy library Ghost [Martinez Peck 2014]. It has the advantage that all messages can be intercepted instead of only those that are not understood with the classic `#doesNotUnderstand:` method.

Upon message reception, if no method has been generated for the selector of that message, the `#cannotInterpret:` method in class `ActiveObject` is executed and triggers the compilation of the generated method in the synthetic class. Method redefinitions in synthetic subclasses cannot be intercepted with this mechanism if the redefined method are already generated in superclasses. Consequently, these methods are compiled eagerly: when a synthetic class is created all method redefintion are generated.

2.3.2.3 Composition of Behavioral Variations

A special metaobject redefines the basic operations of the language to alter the interpretation of its base object. We call these alteration *behavioral variations*. A behavioral variation is typically a cross-cutting concern such as security, logging or persistance.

As we saw previously, one way to implement a behavioral variation is to subclass the root metaobject class to redefine some operations and perform a super-send to invoke the default operation. The problem of this approach is that it is not composable. If a developer wants a metaobject that traces and profiles messages, he cannot reuse a message tracing metaobject class and a profiling metaobject class. Instead, he needs to create a new metaobject class that combines both behavioral variations. This approach does not scale because it leads to an explosion of metaobject classes. So instead we choose to follow a decorator approach for implementing behavioral variations: a behavioral variation is implemented as a metaobject decorator as illustrated in Figure 2.4. Naturally, the order of composition matters *e.g.* tracing before profiling affects the reported duration os profiling that takes the time to trace into account.

2.3.2.4 Examples of Behavioral Variations

This section presents a few examples of simple behavioral variations. The first three examples are behavioral variations that perform simple dynamic analyses by intercepting messages: tracing, logging and statistical typing. The last one intercepts instance variable assignments to ensure that an object is immutable.

Tracing. A really simple behavioral variation is tracing. When a message is received, its selector and arguments are simply printed on the transcript.

```
Tracing>>receive: aMessage
   Transcript
      print: aMessage selector;
      print: aMessage arguments;
      cr.
   ↑ wrappedMO receive: aMessage
```

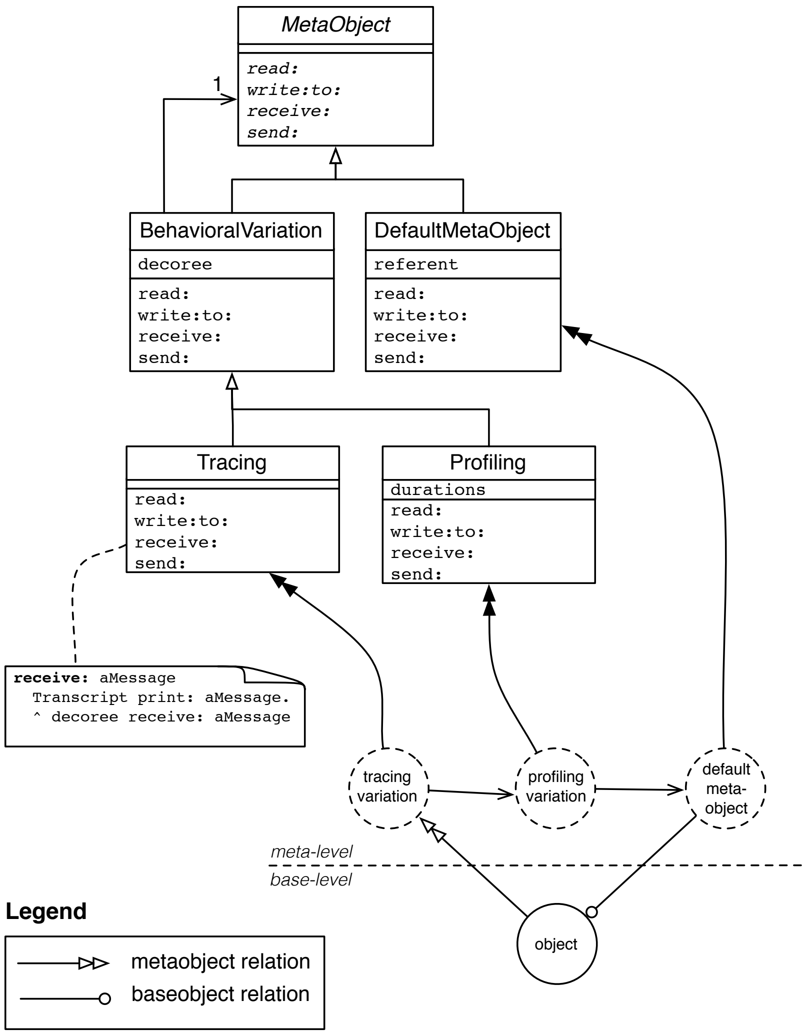


Figure 2.4: Example of two behavioral variations composed together. The metaobject of `object` is a tracing behavioral variation that decorates a profiling behavioral variation that decorates a default metaobject pointing back to its referent `object`.

Profiling. Another behavioral variation that performs a behavioral analysis is profiling.

```
Profiling>>initialize
   tallies := Dictionary new

ProfilingBA>>receive: aMessage
   | start |
   start := Time now.
   [ ↑  wrappedMO receive: aMessage ] ensure: [
      | duration |
      duration := Time now - start.
      (tallies
         at: aMessage selector
         ifAbsentPut: [ OrderedCollection new ]) add: duration ]
```

The `tallies` instance variable is a dictionary used to map a selector to a list of durations. The `#receive:` method measure the duration of each message and stores it in the dictionary. A method `#showResults` then prints the average duration the standard deviation and the number of execution for each selector.

```
ProfilingBA>>showResults
   tallies keysAndValuesDo: [ :selector :durations |
      Transcript
         print: selector;
         show: ' average: ';
         print: durations average;
         show: ' stdev: ';
         print: durations stdev
         show: ' executions: ';
         print: duration size;
         cr ]
```

Statistical Typing. More advanced dynamic analyses are also possible. Here we consider behavioral variation that stores the class of the arguments of each messages to gives information about their possible dynamic types. This is useful to gain understanding about a complex program behavioral in a dynamically-typed language.

```
StatisticalTyping>>initialize
   types := Dictionary new

StatisticalTyping>>receive: aMessage
   | array |
   array := types
      at: aMessage selector
      ifAbsentPut: [ Array new: aMessage arguments size ].
   1 to: aMessage arguments size [ :index |
      | bag |
      bag := (array at: index) ifNil: [ array at: index put: Bag
   new ].
      bag add: (aMessage arguments at: index) class ].
   ↑  wrappedMO receive: aMessage
```

The `types` instance variable is a dictionary that maps a selector to an array. Each element of such array is a bag that stores the types of arguments of the received messages. The method `argumentTypesOf:` prints the dynamic types of each arguments of a given selector with a percentage of occurence.

```
StatisticalTyping>>argumentTypesOf: aSelector
   (types at: aSelector) withIndexDo: [ :argBag :argIndex |
      Transcript
         show: 'argument ';
         print: argIndex;
         crShow: ':'.
      argBag sortedCounts: [ :count :class |
         Transcript
            show: class;
            space;
            print: (count / argBag size) round: 2;
            crShow: '%' ] ]
```

Read-Only. This last example shows a simple use of instance variable write interception to ensure that an object is immutable.

```
ReadOnly>>write: anIV to: anObject
   ↑  Error signal: 'Illegal write to ', anIV , ' in ', anObject
   printString
```

2.4 Conclusion

We presented reflective concepts and reflective architecture in object-oriented languages. We presented the reflective architecture that is used in this thesis that takes the form of an object-centric MOP. Such architecture can enable behavioral variations on a per-object basis. Behavioral variations can focus on monitoring and adaptive aspects, thereby improving adaptability but also on encapsulation and security aspects, thereby improving security. In the next chapter we will study proxies, that allow an even more fine-grained form of implicit reflection. Proxies enable a behavioral variation on a per-reference basis. Different clients can thus have different behavioral variations implementing different encapsulation/security rules.

CHAPTER 3

Scoping Behavioral Variations with Proxies

Contents

The last chapter presented reflection and gave a formalization of a simple object-centric MOP. We saw how implicit reflection can customize the behavior of individual objects by altering the default semantics of the language with custom metaobjects. This chapter presents the proxy mechanism, that offers many ways to scope behavioral variations in space and time. After a generic presentation of proxies, this chapter reviews the typical implementations and see how proxies are realized in an object-centric MOP. We will discuss proxies limitations and different possible semantics. Finally, we will show how proxies can propagate behavioral variations in space and time.

3.1 Introduction to Proxies

A proxy, or wrapper, is an object that acts as a surrogate for another object called its *target*. A proxy mediates interactions between its target and its clients. This indirection has many use cases *e.g.* base-level behavior adaptation, access-control, contracts [Strickland 2012], logging, profiling, dynamic analyzes etc.

The classical realization of a proxy is to implement a proxy class that adapts the interface of another class. A proxy points to its target, an instance of the adapted class. This technique is at the core of the *Proxy* and the *Decorator* design patterns [Gamma 1995]. The problem of this kind of realization is that may lead to a lot of duplicated code. For one behavioral variation, developers need to implement one proxy class for each class that needs to be adapted. With many behavioral variations and adapted classes, it leads to an explosion of proxy classes. Furthermore, the set of adapted classes is not open-ended: it is not possible to adapt a class that has no proxy class for a given behavioral variation.

Consequently, several solutions have been proposed to make generic proxies [Pascoe 1986, Eugster 2006, Van Cutsem 2010, Martinez Peck 2014]. The idea is that proxies can intercept certain operations during execution just like in a MOP, typically the reception of messages and the accesses to the object state. The proxy can take some actions before, after or even instead of performing the original operation on the target. This permits proxy to implement various behavioral variation: tracing, profiling, contracts, access control, read-only access...

Proxies have several advantages. First, behavioral variations expressed with proxies compose naturally. For instance, tracing and profiling behavioral variations can be implemented by separate proxies that can be combined to apply both behavioral variations. Different parties can add their own variations without being aware of others already active for the same target. Also, proxies naturally support *partial reflection* [Tanter 2003] at an object-level granularity: proxies scope behavioral variation to certain clients only. All other objects in the system remain unaffected and pay no performance overhead. This means that the behavioral variation of a proxy is not enabled when accessing its target directly. A variation is enabled for clients who possess a reference to the proxy while other clients may have a reference to the target or to another proxy implementing another behavioral variation. It is up to the creator of the proxy to decide whether to pass the proxy, that enables to behavioral variation, or the target, that does not.

3.2 Proxy Implementations

We review three main implementation styles for proxy implementation. First those that redefine a method lookup failure hook, then modern stratified implementation and finally in a context of an object-centric MOP.

3.2.1 Method Lookup Failure Hook

Let us first look at a common implementation of proxies in dynamically-typed languages [Pascoe 1986, McCullough 1987, Ducasse 1999]. Certain dynamically-typed languages allow objects to answer messages they don't implement. When an object receives a message it does not understand (*i.e.* the method lookup fails), the runtime sends a special message to the object passing the message name and the message arguments. The class of that object can redefine the corresponding method to return a "fail-back" answer. This special failure message exists in many languages:

- `doesNotUnderstand: aMessage` in Smalltalk,
- `doesNotRecognizeSelector: aSelector` in Objective-C,
- `method_missing(method, *args, &block)` in Ruby,
- `__getattr__(self, name)` in Python,
- and `__noSuchMethod__(name, args)` in SpiderMonkey Javascript.

This facility can be used to implement generic proxies. Since only messages not understood can be intercepted the proxy class is made to understand only a few messages. For example, in *Squeak* and *Pharo* the root class is not `Object` but its direct superclass `ProtoObject`. This class has far less methods than `Object`. While normal classes inherit `Object`, classes that redefine `doesNotUnderstand:` typically inherit from `ProtoObject` to be able to intercept more messages. *Ruby* follows the same scheme with the class `BasicObject` and its direct subclass `Delegator` for implementing proxies. *Objective-C* defines a second special root class `NSProxy`.

3.2.1.1 Example In Smalltalk

The following listing shows an example of a tracing proxy in Smalltalk.

```
"Tracing proxy is a subclass of ProtoObject and has a 'target'
    instance variable"
ProtoObject subclass: #TracingProxy
   instanceVariableNames: 'target'
   classVariableNames: ''
   category: 'ProxyExample'

"Instantiation"
TracingProxy class >> target: anObject
   ↑ self new
      initializeWithTarget: anObject;
      yourself

"Initialization"
TracingProxy >> initializeWithTarget: anObject
   target := anObject

"Print message selector on Transcript then forward message to the
    target"
TracingProxy >> doesNotUnderstand: aMessage
   Transcript crShow: aMessage selector.
   ↑ aMessage sendTo: target
```

A tracing proxy `prx` for an object `obj` can then be created with the expression `prx := TracingProxy target: obj`. Messages sent to `prx` are then printed on the transcript[1].

3.2.1.2 Limitations

This kind of implementation has two limitations. The first limitation is that only message reception can be intercepted and only for messages not understood by the proxy class. Consequently, implementations look for proxy class with a minimum number of implemented methods. In Pharo and Squeak Smalltalk, the root of the class hierarchy is `ProtoObject` and understands far less messages than its direct

[1]The transcript is an equivalent of a console in a Smalltalk environment.

subclass `Object`. This class has been introduced precisely for classes that redefine `#doesNotUnderstand:`. Ruby uses a similar solution with the class `BasicObject` and its subclass `Delegate` used for proxies. Objective-C introduced another root class `NSProxy` for the same reason.

The second limitation of this kind of solution is that it conflates the base-level with the meta-level operation for interception. There is a clash between the API of the target and the API of the proxy implementation. Since these solutions are only concerned with one operation (message reception), and consequently one special method (here `doesNotUnderstand:`) there is little risk of confusion. If the special method is not made private clients can call it directly with a forged message reification to bypass methods implemented by the proxy class. If a proxy class is built with the first limitation in mind, this can be problematic.

3.2.2 Stratified Solutions

To solve this problem, modern proxy mechanisms stratify the base and meta levels of proxies [Eugster 2006, Van Cutsem 2010, Strickland 2012, Martinez Peck 2014]. In such case, the behavior of a proxy is defined by a separate object, typically called its *handler*. Handler classes implement one method per interceptable operation. These methods are called *traps* [Van Cutsem 2010]. When an operation is applied to a proxy, the proxy intercepts it and invokes the corresponding trap in its handler. Like with the previous approach, the handler can take some actions before, after or even instead of performing the original operation on the target. Figure 3.1 shows the relationships between a proxy, its handler and its target with an example of message interception. We depict proxies differently than non-proxy objects with half-circles.

These solutions are said to be stratified because a proxy and its handler are distinct, but usually such solution is not fully stratified as the reflective architecture of the host language is not stratified itself. This distinction allows references to the handler and the proxy to be kept separately: the creator of a proxy typically keeps a reference to the handler while clients are only given a reference to the proxy. But to forward operations to the target, these solutions fall under two categories and neither comply to the stratification principle. In the first category the target

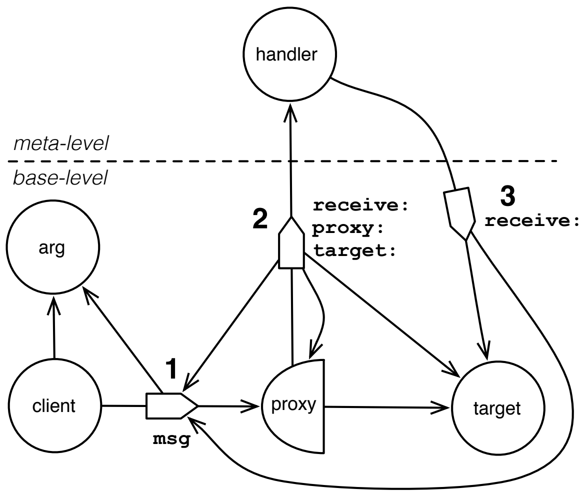


Figure 3.1: Example of message interception in a typical stratified solution. First, the client sends the message #msg to the proxy (1). Then, the proxy intercepts the message and invokes the handler's message reception trap, #receive:proxy:target:, with a reification of the message, the proxy and the target as arguments (various set of arguments are possible). Finally, the handler ask the target to reflectively invoke the message with #receive: (3)

exposes directly meta-level operations. In the second category meta-level operations are exposed globally. So if these proxy implementations are indeed stratified themselves, the host language reflective API is typically not stratified. It is consequently interesting to see how proxies are implemented within a stratified reflective architecture like an object-centric MOP.

3.2.3 In an Object-Centric MOP

In an object-centric MOP, proxies are easily implemented. The proxy lies at the base-level, its metaobject is the equivalent of the handler in the previous implementations and the set of available traps is the available MOP. A proxy is an object whose behavior is an alteration of the behavior of its target. The proxy's metaobject points to the target's metaobject, instead of the proxy pointing to its target directly. The trap methods of the proxy's metaobject typically forwards to the corresponding target's metaobject trap methods before or after taking some actions that implements the behavior alteration. Note that in an object-centric MOP, and contrary to the previous solutions, metaobjects cannot be shared among several proxies as handlers can.

Figure 3.2 shows an example of message interception. We depict proxies differently than non-proxy objects with half-circles. The **receive:** method simply forwards the message to the target, by also sending **receive:** to the metaobject of the target.

```
ProxyMO >> receive: aMessage
   ↑ targetMO receive: aMessage
```

In the rest of this chapter we assume that we are in an object-centric MOP proxy implementation but all conclusions extend to other kinds of implementation. For the sake of clarity certain figures will omit metaobject and instead will show a direct reference from an object to its target.

3.3 Proxies Limitation

Despite their wide range of applications, proxies also have limitations. These limitations comes from the same characteristic that gives proxies their strength: a proxy

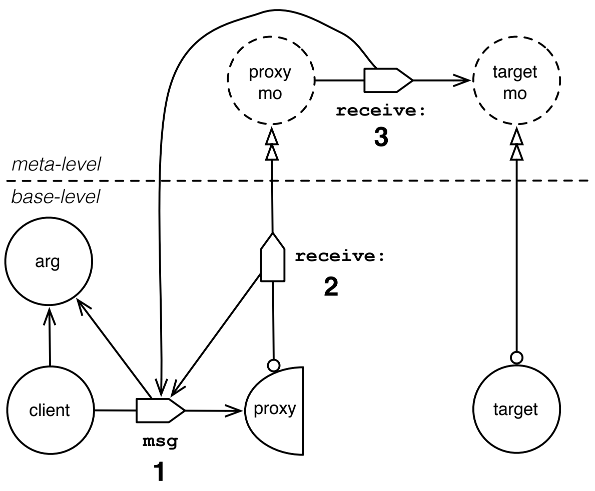


Figure 3.2: Example of message interception in an object-centric MOP. First, the client sends the message #msg to the proxy (1). Then, the proxy intercepts the message and invokes (#receive:) method of its metaobject with a reification the #msg message as argument (2). Finally proxy's metaobject forwards this message to its target's metaobject (3).

and its target are different objects. While this distinction permits proxy creators to scope a behavioral variation to some clients only, it is also the source of an issue known as the *two-body problem* [Eugster 2006, Welch 1999, Renaud 2001]. The two-body problem can in fact be decomposed in two subproblems: the **self** *problem* [Lieberman 1986] and the *encapsulation problem*.

3.3.1 Self Problem

The **self** problem arises because the **self** pseudovariable is different in a proxy and its target. When a proxy intercepts a message, it can execute code before or after forwarding the message to the target. But doing so, it looses the control of the execution: after forwarding self-sends are not intercepted. Figure 3.3 shows an example of this problem. In this example, a proxy wraps a widget to trace messages. The widget has a method `paint` that draws the widget on a canvas and a method `repaint` that clears the canvas zone occupied by the widget and call `paint`.

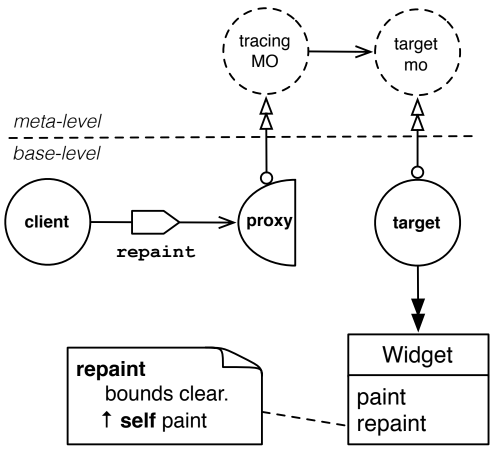


Figure 3.3: Illustration of the `self` problem. The client sends `repaint` on the proxy that intercept and trace the message. But once the proxy forward the message to the target it loose control and cannot intercept self-sends: the message send `paint` is not traced.

When the client (presumably from the widget library) sends repaint to the widget only the message `repaint` is traced.

To solve the `self` problem, proxies can operate by *delegation* [Lieberman 1986] like object inheritance in prototype object languages. With delegation the proxy does not forward the message to its target but executes itself the code of the method the target would have performed. In other words, the value of `self` is always the proxy. We will study the difference between forwarding and delegation hereafter in Section 3.4.

3.3.2 Encapsulation Problem

The encapsulation problem also comes from the fact that the `self` pseudovariable is different in a proxy and its target. Encapsulation here means that a proxy should not expose its target. This is especially important when using proxies for access control or other security-related behavioral variations like *e.g.* a read-only behavioral variation. If clients are able to retrieve the target of a proxy, the policy the proxy wants to enforce is bypassed. The simplest most common form of the problem arises when the target returns a `self`-reference. Because `self` is bound to the target, the proxy lets a reference to its target escape to the clients. Using delegation instead of forwarding solves this form of the problem. A less common form of the encapsulation problem is when the target returns a reference to itself not via `self`. The target might store a reference to itself in one of its instance variables or receive it via an argument from a collaborating object. To solve both forms of the problem at once, the proxy can test if return values from message interceptions are the target and return itself instead. Here is the code of the `#receive:` method of the metaobject class for such a proxy.

```
EncapsulatedProxyMO >> receive: aMessage
   | return |
   return := targetMO receive: aMessage.
   ↑ return == targetMO referent
      ifTrue: [ self referent ]
      ifFalse: [ return ]
```

Still, such a proxy metaobject class fails to completely solve the encapsulation problem because a proxy could return an object that gives access to its target. We will see how membranes solves this issue in Subsection 3.5.2.

3.4 Forwarding vs. Delegation

Delegation [Lieberman 1986] is a mechanism used to implement object inheritance in prototype-based languages. It permits the behavior of an object to be composed dynamically from other objects with partial behaviors. It is opposed to forwarding (sometimes called *consultation*) that is the classical semantics of object-oriented languages. The difference between delegation and forwarding is the treatment of the `self` variable. Delegation has also been used in other scenarios than prototype-based languages, for example in languages that combine class inheritance and object inheritance [Kniesel 1999, Viega 2000]. Another example are delegation layers [Ostermann 2002] which extend the notion of delegation from objects to graph of collaborating objects. A delegation layer affects not only a wrapped object but its collaborating objects as well, refining specific sets of methods of the objects in the collaboration.

Delegation has also been studied in the context of proxies [Büchi 2000, Bettini 2007, Wernli 2014, Teruel 2015]. The difference between conventional forwarding and delegation is how `self` is bound during the execution of the target methods. Traditional proxy implementations found in class-based object-oriented languages, such as *Java*'s *dynamic proxies*, operate by forwarding. In *ECMAScript 6*, because it's a prototype-based language, proxies operate by delegation.

When a message is intercepted by a proxy, the proxy may decide to *forward* the message to its target: the method corresponding to the message is executed with self-references bound to the target. This implies that the proxy loses control of the execution. In the context of Figure 3.3, we saw that with normal message forwarding, the proxy first traces the message `repaint` before resending the message to the target. At this point, the proxy has lost the control of the execution, its target executes its `repaint` method that self-sends `paint` but the later message is not traced. With delegation, the proxy intercepts the message `repaint`, traces it, and then *delegates* this message to the target. That is the proxy executes itself the method the target would have executed: the value of `self` is the proxy. Consequently, the `self` reference in the body of the `repaint` method refers to the proxy instead of the target: the self-send `paint` is correctly intercepted by the proxy and traced. Delegation is mandatory to intercept operations occurring during a method execution such that object state reads and writes and message sends (as opposed to message reception). We refer to these interpretation operations as *sub-method operations*.

3.4.1 Effect of Delegation on Proxy Composition

With delegation the identity of the proxy that originally intercepted the operation is maintained; this permits several behavioral variations to be composed by forming chains of proxies. By using proxies as the targets of other proxies, we obtain chains of proxies. In this case, when an operation is intercepted, the corresponding trap methods of each proxy metaobject are executed in the order specified by the chain. Consequently, the order of composition matters. This offers a natural way to compose multiple behavioral variations.

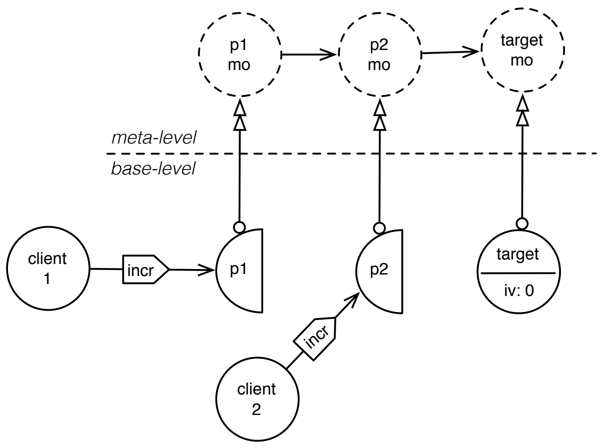


Figure 3.4: A chain of two proxies composing two behavioral variations. Client 1 has a composition of the behavioral variations of `p1` and `p2`.

In Figure 3.4, we have two proxies, a target, their respective metaobject and two clients. The target of `p1` is `p2` and the target of `p2` is `target`. This forms a chain of two proxies. The first client object sends a message *incr* that is supposed to change the state of the target object by modifying its instance variable *iv*. The second client has only the behavioral variation of the second proxy enabled. Let see in details the execution of the message `incr` for the first client with forwarding and delegation.

With forwarding execution begins with the message interception that triggers the message-reception meta-level operation in the metaobject of `p1`. The metaobject may at some point forward the intercepted message to the target of `p1` — namely the proxy `p2` — losing control of the execution at the same time. The same scenario applies to `p2`: it intercepts the message, invokes its message-reception meta-level operation, its metaobject forwards the message to `target` which then processes the message normally and increments the value of its instance variable `iv`. Without delegation the instance variable write is not intercepted by `p1` nor `p2`. It would be the same thing with a self-send.

With delegation the execution begins the same: `p1` intercepts the message and invokes the message-reception operation in its metaobject. But instead of forwarding the message to `p2`, the metaobject of `p1` instead delegates the message: it specifies that the receiver (**`self`**) should be bound to `p1`. Then `p2` intercepts that message, which applies its own behavioral variation and delegates the message to `target`. At this point `p1` is still the receiver and has still control over the execution: it can intercept sub-method operations, in particular the modification of `iv`. In reaction to this interception, the state-write operation of `p1` is executed in its metaobject. The state-write operation of `p2` is then invoked. This example shows that delegation permits behavioral variations to be composed correctly even in the context of sub-method operations.

To sum up, using delegation instead of forwarding for proxy-based intercession permits proxies to intercept sub-method operations and to compose behavioral variations by forming chains of proxies.

3.5 Proxy Propagation

We present two approaches to isolation with propagation of proxies during execution. The first one, known as *membrane* [Miller 2006, Van Cutsem 2010], dynamically isolates a graph of objects through a layer of proxies. References exchanged through a membrane are wrapped with proxies enabling one of two behavioral variations depending on the direction they cross the membrane boundary. The second one, that we call *control flow propagation* [Wernli 2014, Teruel 2015, Arnaud 2010, Arnaud 2015], applies a behavioral variation to all objects involved in a given execution from a particular message send. Before presenting these two forms of propagation, we first discuss the requirements for the dynamic wrapping of proxies required for both.

3.5.1 Wrapping Objects During Propagation

To propagate a behavioral variation, the automatic wrapping of object with proxy should be done with certain precautions. Without taking care, an object could be wrapped by different proxies, *i.e.* an object wrapped at different times yields different proxies. This approach is inefficient because a proxy that already exists can be reused. A more important issue is that object identity is not preserved across the boundaries, potentially breaking code that relies on object identity. Consequently proxies are cached. Clients always obtain the same proxy for the same object.

Also an object that is already wrapped should not be rewrapped multiple times because the behavioral variation would be applied more than once, yielding poor performance and possibly incorrect results. The wrapping operation must be idempotent: normal objects get wrapped and proxies are just return untouched.

Finally, caching proxies should not prevent garbage collection. When a proxy is no longer referenced, it should be garbage collected. But such cache is usually implemented as a hash table that maps objects to their cached proxy. A naive solution is to use a hash table with weak keys: if a key is no longer referenced outside the cache the associated value also becomes collectable. But if an object used as a key references a proxy used as a value, garbage collection is impossible. Consequently, a more sophisticated finalization scheme is needed: *ephemeron* [Hayes 1997]. An ephemeron is an association that holds weakly on its key and strongly on its value. Once the key is garbage collected the value is held weakly. Ephemerons can be used to implement a special kind of hash table known as a *weak map* . Weak maps solve the problem of garbage collection. Given an available implementation of weak maps, the code of the wrapping operation of a proxy cache is quite straightforward.

```
ProxyCache >> wrap: anObject
   ↑ proxyMap
      at: anObject
      ifAbsentPut: [ self basicWrap: anObject ]
```

3.5.2 Membrane

The first form of proxy propagation is a *membrane* [Miller 2006, Van Cutsem 2010] A membrane isolates a whole object graph from client by wrapping objects with proxies, by transitively propagating a behavioral variation to all references exchanged through the membrane. A membrane wraps all objects crossing the boundary of the graph in both directions: all interaction between the isolated graph and the outside is mediated via proxies.

3.5.2.1 Wrapping Rules

Figure 3.5 shows the wrapping rules in action. At start, a membrane consists of a sole proxy whose target is the root of the object graph to isolate. When a message

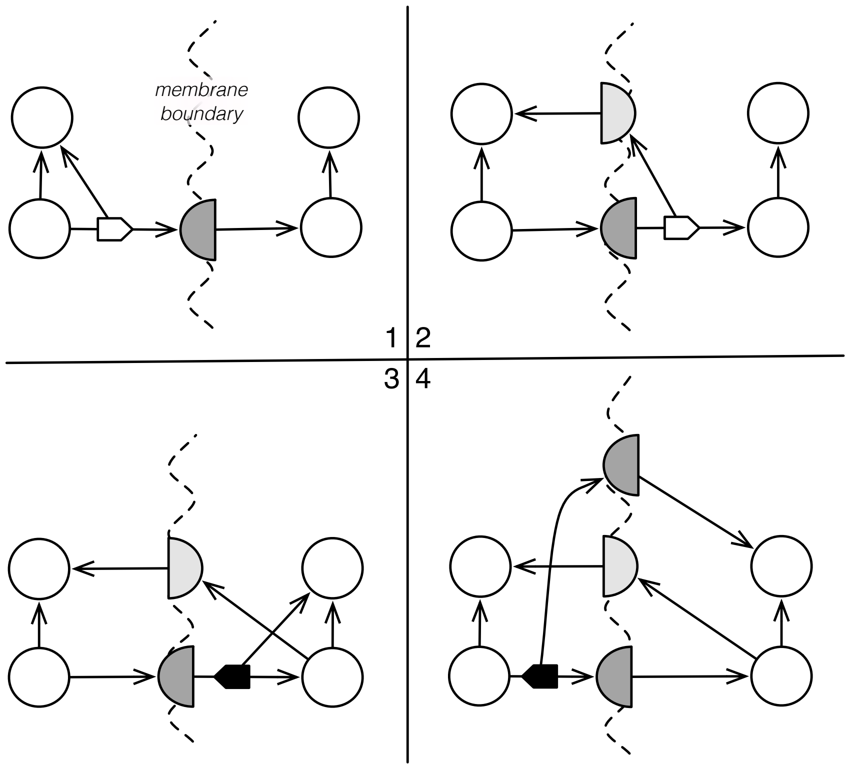


Figure 3.5: Wrapping rules of membranes (metaobjects are omitted for clarity). The client object sends a message with an argument (1) and the argument is wrapped when crossing the membrane (2). The target is about to return an object (3) and its proxy wraps it before returning it to the client (4).

is sent to this proxy (situation 1 in the Figure 3.5), the arguments are wrapped and the message passed to the target (2). Once the target has processed the message and returns an object (3), its proxy wrap this object before handing it over to the other side (4). The wrapping of arguments and return value is symmetric: it extends to references exchanged in the other direction. This means that when an object from the isolated graph sends a message to a proxy (obtained as argument in a previous message), the arguments and the return value are also wrapped. Likewise, all messages return proxies from the root proxy and all objects passed as argument into the isolated graph are wrapped too.

But these wrapping are not applied indiscriminately because some objects would be wrapped several times when crossing back and forth. For example, if an object

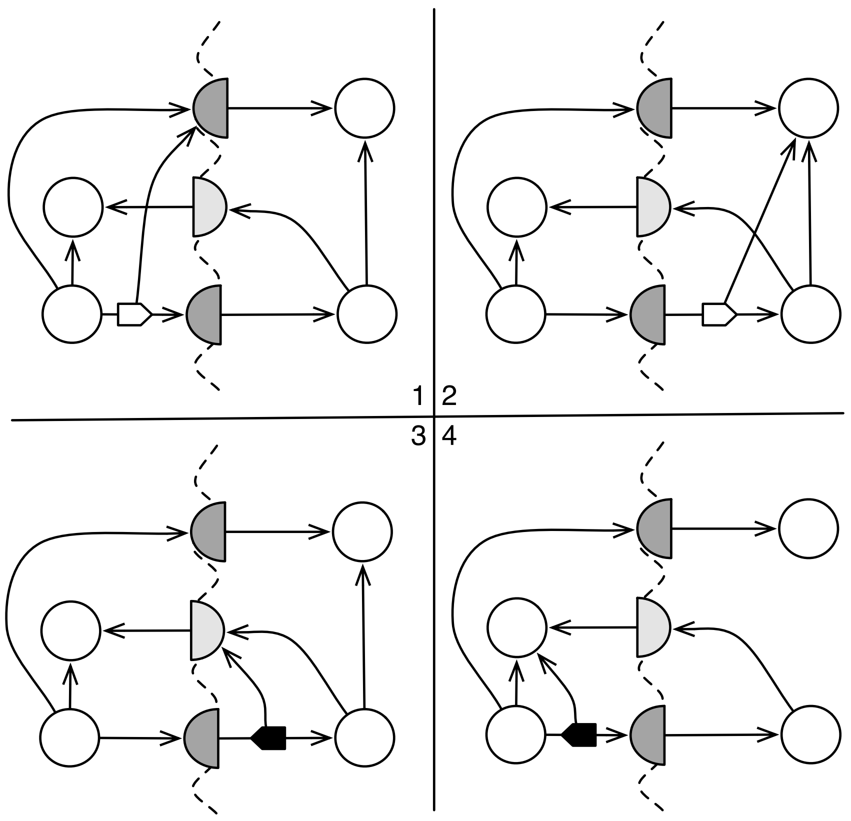


Figure 3.6: Unwrapping rules of membranes (metaobjects are omitted for clarity). The client object sends a message with a proxy as argument (1) and the proxy is unwrapped when crossing the membrane (2). The target is about to return a proxy (3) and the its proxy unwraps it before returning it to the client (4).

sends a message to a proxy to the other side with also a proxy to the other as argument, the argument would be wrapped twice. So instead, in such situation, objects are unwrapped when they come back to the side they belong. Figure 3.6 depicts these unwrapping rules. When a message with a proxy to the other side as argument is sent across the membrane (1), the argument is unwrapped and the message passed to the other side (2). Likewise, when a message returns a proxy to the other side (3), it is also unwrapped (4). Like wrapping, these unwrapping rules are also symmetric.

The term membrane usually refers to this kind of propagation when the propagated behavioral variation is revocability [Miller 2006, Van Cutsem 2010]. A revocable proxy transparently forwards message to its target as long as it is active. The creator of the proxy can then deactivate it so that it no longer forwards message to its target. The permission granted to clients to send messages to the target is effectively revoked. A revocable membrane applies revocability to a whole object graph: all the objects that clients obtained from the graph can be revoked all at once. We call them *revocable membranes* and use a broader meaning for membrane: a membrane is the of propagation presented here, independently from the particular behavioral variation it propagates.

3.5.2.2 Inside-out and Outside-in Behavioral Variations Distinction

Wrapping and unwrapping rules are symmetric because a membrane has two sides: objects inside the membrane and objects outside. There are consequently two kinds of proxies: *outside-in proxies* that mediate the interactions from the outside to the inside (in dark grey in Figure 3.5 and Figure 3.6) and *inside-out proxies* that mediate the interactions from the inside to the outside (in light grey in Figure 3.5 and Figure 3.6) Each kind of proxy can enable its own behavioral variation. For example a membrane could enable tracing in both direction and read-only only for outside-in proxies.

Membranes have many different use cases but when it comes to encapsulation and security. A membrane can be used in two different ways: protecting the outside from the inside and protecting the inside from the outside. In the first case, the behavioral variation of inside-out proxies enforce a policy while outside-in proxies are just here to automatically transform objects passed as argument to the inside into inside-out proxies. Outside-in proxies mark their target as untrusted: they wrap arguments into inside-out proxies and return values into outside-in proxies but they typically enforce no policy. In the second case where a membrane is used to protect the inside from the outside, the situation is reversed: outside-in proxies enforce a policy while inside-out proxies enforce no policy but are used to mark their respective target as untrusted.

3.5.2.3 Implementation

We saw that the wrapping and unwrapping rules of membrane for each side are symmetric. An implementation can profit from this symmetry. In such implementation, the logic of membrane wrapping rules are split in two objects, each one representing one side of the membrane. Each side has a proxy cache and references the other. Here is the code of a membrane proxy metaobject and of membrane side.

```
MembraneProxyMO>>receive: aMessage
   aMessage arguments: aMessage arguments collect: [ :arg |
      mySide toOtherSide: arg ]
   ↑ mySide toMySide: (targetMO receive: aMessage).

MembraneSide>>toMySide: anObjectOrProxy
   ↑ (proxyCache includesKey: anObjectOrProxy)
      ifTrue: [ anObjectOrProxy ]
      ifFalse: [
         | proxy |
         proxy := proxyCache wrap: anObjectOrProxy.
         proxy meta side: self.
         proxy ]

MembraneSide>>toOtherSide: anObjectOrProxy
   ↑ otherSide toMySide: anObjectOrProxy
```

3.5.3 Control flow propagation

A proxy can encode a behavioral variation that will be consistently propagated to all objects accessed during the evaluation of a message send, *i.e.* the *dynamic extent* of the message. Scoping behavioral variations to dynamic extents increases expressiveness in useful ways [Tanter 2008, Tanter 2009]. With such a form of scoping, it is possible to execute code in a read-only manner [Arnaud 2010] (thus improving safety), or to track all state mutations to ease recovery in case of errors (thus improving reliability), or to trace and profile code at a fine-grained level (thus improving monitoring). A typical use case is to let proxies mimic their target state. Likewise, one can think of a proxy as the entry to a parallel universe where each object has potentially an alter-ego with a different state. This mechanism is close

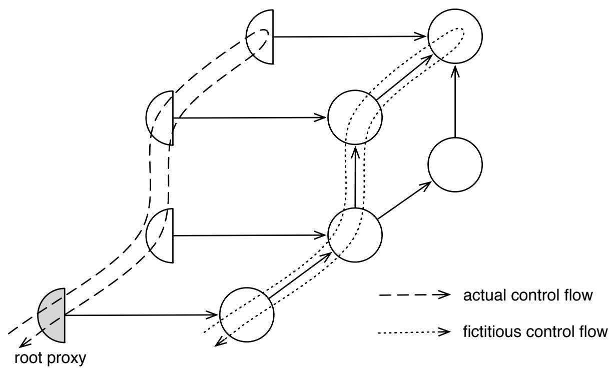


Figure 3.7: A depiction of control flow propagation (metaobjects are omitted for clarity). A root proxy (in grey) wraps a target object that is connected to some object graph. The dashed line depicts the actual control flow of the execution. Proxies are created on need and are eligible for garbage collection once the control flow leaves them. The actual control flow parallels a fictitious control flow (dotted line) *i.e.* the control flow that would have resulted if execution had not been intercepted by the root proxy.

to the concepts of *virtual copies* [Mittal 1986], *worlds* [Warth 2011] and *handles* [Arnaud 2010]. As an example, this permits to implement software transactional memory quite easily.

With this technique a proxy propagates a behavioral variation to the dynamic extents of the messages it receives. These dynamic extents are the parts of the execution delimited by the processing of a message received by the proxy, from the reception of the message until the corresponding method returns. All objects accessed during this dynamic extent are consistently represented by proxies that are created on-demand. We refer to the first proxy that initiates the propagations as the *root proxy*. Other proxies created during the propagation are called *non-root proxies*. The root proxy can be seen as the entry point to a lazily-created parallel object graph as depicted in Figure 3.7.

3.5.3.1 Tracing example

We will illustrate control flow propagation with a tracing example. Let us consider the method `Integer>>fib` which computes the Fibonacci value of an integer using recursion:

```
Integer>>fib
   self < 2 ifTrue: [ ↑ self ].
   ↑ (self - 1) fib + (self - 2) fib
```

The computation of the Fibonacci value of `2` corresponds to the following sequence of message sends (first the receiver of the message, then the message with its arguments):

```
2 fib
2 < 2
false ifTrue: [ ↑  self ]
2 - 1
1 fib
1 < 2
true ifTrue: [ ↑  self ]
[ ↑  self ] value
2 - 2
0 fib
0 < 2
true ifTrue:  [ ↑  self ]
[ ↑  self ] value
1 + 0
```

To automatically trace message sends, we can use a proxy to intercept message sends and print them. Delegation ensures that messages received by the proxy are traced, including self-sends. However, it would fail to trace messages sent to other objects and would print `2 fib, 2 < 2, 2 - 1, 2 - 2`, but all the messages sent to `1, 0, true, false` and the closure `[ ↑  self ]` would not be traced. To consistently apply a behavioral variation during the execution of a message send, all objects accessed during the execution must be represented with proxies.

3.5.3.2 Wrapping rule

The wrapping rules of control flow propagation are simple: wrap the receiver and the arguments of messages *sent* and unwrap the value of state writes. When an instance variable is assigned a new value, the value of the assignment is unwrapped before performing that assignment. This way, the proxies created during the propagation are only referred to from within the call stack and don't corrupt the object graph connected to the target. Additionally, one rule must be decided for the return values messages that start the propagation: consistently wrapping or unwrapping it. Indeed, with only the two previous rules, the return value can be a proxy or not depending on the code executed. If return value are consistently wrapped, objects returned would also have the behavioral variation enabled for the dynamic extents of the message they receive and also the objects these objects return, in a similar fashion than membranes. If return values are consistently unwrapped, objects returned would also have the behavioral variation enabled for the dynamic extents of the message they receive and also the objects these objects return. Here is the code of a metaobject class enabling control flow propagation with unwrapped returns Behavioral variations concerned with encapsulation or security typically use wrapped returns values and behavioral variations concerned with monitoring can wrap return values or not.

```
ControlFlowPropagationMO>>receive: aMessage
   ↑ self unwrap: (aMessage delegateTo: targetMO)

ControlFlowPropagationMO>>send: aMessage
   aMessage receiver: (self wrap: aMessage receiver)
   aMessage arguments: (aMessage arguments collect: [ :arg | self
     wrap: arg ])
   ↑ targetMO send: aMessage

ControlFlowPropagationMO>>write: anIV to: anObject
   ↑ targetMO write: anIV to: (self unwrap: anObject)
```

3.5.3.3 Examples

We focus in this section on new examples enabled by control flow propagation.

Read-only Execution Read-only execution [Arnaud 2010] prevents mutation of state during evaluation. Read-only execution dynamically guarantees that the evaluation of a given piece of code yields no mutation. Classical proxies could restrict the interface of a given object to the subset of read-only methods. However, they would fail to enable read-only execution of arbitrary methods, or to guarantee that methods are deeply read-only. Read-only execution can be implemented trivially with a behavioral variation that fails upon state writes.

```
ReadOnlyMO>>write: anIV to: anObject
   ReadOnlyError signal
```

Thanks to proxy-based intercession, the target object is still available to trusted clients that can modify it. Only clients holding a reference to the proxy are affected by the read-only policy.

Object Versioning. To tolerate errors, developers implement recovery blocks that undo mutations and leave the objects in a consistent state. Typically, this entails cloning objects to obtain snapshots. Control flow propagation enables the implementation of object versioning concisely. Before any field is mutated, the metaobject class shown below records the old value into a log using a reflective field read. The log can be used in recovery block, for instance to implement rollback.

```
RecordingMO>>write: anIV to: anObject
  | oldValue |
  oldValue := targetMO read: anIV.
  log add: { targetMO. anIV. oldValue deepCopy }.
  ↑ targetMO write: anIV to: anObject
```

The log can then be used to reflectively undo changes if needed.

```
aLog reverseDo: [ :change | change first write: change second
    to: change third ]
```

3.5.4 Other Variations of Propagation

Proxies provide flexible building blocks to implement various forms of scopes, possibly blurring the line between static and dynamic scoping, similarly to Tanter's *scoping strategies* [Tanter 2009].

3.5.4.1 Filtering on package

Propagation can for instance be adapted to enable a behavioral variation only for instances of classes belonging to specific packages. This can be used to select which parts of an execution are subject to a behavioral variation such as tracing. It is especially useful for excluding kernel classes (string, dictionaries, arrays, etc.) and focusing instead on the classes of the application under analysis.

To implement this form of scoping, it is possible to implement a filtering proxy metaobject class with a set of packages. This class wraps an object only if this object class is declared in one of the packages of interest.

```
PackageScopeControlFlowPropagationMO>>wrap: anObject
  ↑ (packages includes: t class package)
   ifTrue: [ super wrap: anObject ]
   ifFalse: [ self unwrap: anObject ]
```

3.5.4.2 Protecting the Target from Clients

With full control flow propagation, a read-only behavioral variation ensures that no state is modified from within the dynamic extent of messages received by the root proxy. Propagation can be relaxed to lessen the constraints imposed on clients. We can ensure that no state *of the object graph connected to the target* is modified from within the dynamic extent of messages received by the root proxy. To achieve that we can have an alternative propagation that does not wrap initial arguments, *i.e.* the arguments of messages sent to the root proxy. The rationale is that the client necessarily has a reference to each of the objects it passes as arguments in the message to the root proxy. The client can therefore access these objects with the behavioral variation disabled in any case. In such scenarios, we also typically want initial returns to be wrapped so that objects returned by the target are still protected by the behavioral variation. Consequently, this alternative propagation does not apply this unwrapping rule.

3.5.4.3 Protecting the Clients from the Target

A proxy can also propagate a behavioral variation to protect the clients from the targets. Revocable membranes are an example of such protection. The behavioral variation can also be propagated along the control flow, but only to objects passed as argument to the proxy. This is a combination of the control flow propagation and membrane propagation, where the inside-out proxies of a membrane perform a behavioral variation following the control flow. For example, a client ensure that all objects it passes to an untrusted graph will enable read-only execution.

3.6 Related Mechanisms

Many different mechanisms have been proposed to adapt the behavior of objects. This section reviews a few, compare them with proxies and present works related to membranes and control-flow propagation. The control flow propagation technique presented here is inspired from Wernli et myself [Wernli 2014, Teruel 2015]. This previous work use other wrapping rules based on other interception operations like literal and global variable resolution.

Composing Behavior. Inheritance leads to an explosion in the number of classes when multiple variations of a given set of classes must be designed. Static traits [Schärli 2003] or mixins enable the definition of units of reuse that can be composed into classes, but they do not solve the issue of class explosion. Ressia *et al.* proposed *talents* [Ressia 2014] which enable adaptations of the behavior of individual objects by composing trait-like units of behavior dynamically on a per-object basis. Another solution is the use of decorators that refine a specific set of known methods, *e.g.* the method `paint` of a window. Büchi and Weck [Büchi 2000] proposed *generic wrappers*. A generic wrapper dynamically decorates an object to redefine some methods of its statically known type. Bettini *et al.* [Bettini 2007] proposed a similar construct but composition is dynamic. Unlike decorators, proxies find their use when the refinement applies to unknown methods, *e.g.* to trace all invocations.

AOP's Pointcut-Advice Model. Proxy-based intercession differs from the traditional pointcut-advice model of aspect oriented programming. In the pointcut-advice model, an aspect groups definition of pointcuts with corresponding advices, *i.e.* a behavioral variation and its deployment. The metaobject class of a proxy specifies the actions taken upon interception of certain operations. This looks similar to the pointcut-advice model: the body of a metaobject method is akin to an advice and the method itself matches certain points of execution, just like a pointcut does. However, a proxy metaobject class does not specify which objects it will affect. This allows developers to deploy a behavioral variation on specific object references but it does not specifies a deployment scheme. This means that either an application code or its clients code must be modified to include object wrapping instructions. Consequently, aspects and proxies can be complementary since aspects can be used to specify how proxies should be deployed within an application.

Control Flow Pointcuts in AOP. In AOP, control flow pointcuts are popular and supported by mainstream AOP implementations, *e.g.* AspectJ's `cflow` and `cflowbelow`. Aware of the limitations of control flow pointcuts, some AOP implementations provide specific constructs to scope to the dynamic extent of a block of code, *e.g.* CaesarJ's `deploy` [Aracic 2006]. Implemented naively, control flow pointcuts are expensive since they entail a traversal of the stack at run time, but they can be implemented efficiently using partial evaluation [Masuhara 2003].

Composition Filters. In the *Composition Filters* model [Aksit 1993], the incoming and outgoing messages of an object pass through stacks of message filter that can modify the messages in various ways. This model can be implemented with proxies if interception of outgoing message is available. With composition filters, an object can rewrite outgoing messages to change their receiver to an object with that same behavior. As far as we are aware of, composition filters implementation does not offer a mechanism to intercept instance variable accesses but we think it could be easily added. Adding this facility would allow composition filters to realize the control flow propagation technique presented here.

Security Metaobjects. Riechmann *et al.* [Riechmann 1997] presented a propagation of behavioral variation similar to membranes. They propose to extend the *OCap* model with *Security Metaobjects* (SMO). They note that the frequent exchange of object references makes hard to check which part of an application can access a given capability. A SMO can be attached to an object reference to control the messages that can be performed via this reference. Such facility can be emulated by proxies. Also an SMO can attach itself or other SMOs to incoming references (message arguments) and outgoing references (message returns). They show how this facility can implement SMOs that propagate an access control policy to all exchanged references, in a similar fashion than membranes. Riechmann *et al.* later proposed an extension of this model in the context of a role-based access control mechanism [Riechmann 1998]. In this extension, they use two kind SMOs: *principal SMOs* provides principal information that *access control SMOs* use to check access.

Handles. Similarly to our control-flow propagation, Arnaud *et al.* presented *handles* [Arnaud 2013a, Arnaud 2013b] that enable the adaptation of references with behavioral variations that propagate across the control flow. The propagation belongs to the semantics of the handles, whereas in our approach, the propagation is encoded reflectively in a specific proxy metaobject. Our approach is more flexible since it decouples the notion of propagation from the notion of proxy but the handle approach is more efficient since it is implemented at the runtime level.

Contextual behavior In context-oriented programming (COP) [Hirschfeld 2008, von Löwis 2007], variations can be encapsulated into layers that are dynamically activated in the dynamic extent of an expression. Unlike the propagation technique presented in this paper that work better with *homogenous* variations, COP has a better support for *heterogenous* variations [Apel 2008]. COP can be seen as a form of multi-dimensional dispatch, where the context is an additional dimension. Other mechanisms where the behavior of objects varies in a contextual manner are roles [Kristensen 1996], perspectives [Smith 1996], and subjects [Harrison 1993].

3.7 Conclusion

There are advantages to use proxies to scope behavioral variations and thus to support unanticipated changes. We saw the effect of delegation is important for composition and the `self`-problem. Proxies can form chains to compose their behavioral variations: different parties can add their own behavioral variation without being aware of others already active for the same target. We can for instance trace and profile an execution by using tracing proxies and profiling proxies that form chains of delegation (composition). Adapting objects during an execution will not affect other objects in the system (partial reflection [Tanter 2003]). Also, objects are wrapped selectively and a behavioral variation is enabled only for the proxy. A variation is enabled for clients who possess a reference to the proxy while other clients may have a reference to the target or to another proxy implementing another behavioral variation. It is up to the creator of the proxy to decide whether to pass the proxy or the target.

Additionally, with specific wrapping rules it is possible to propagate a behavioral variation to isolate an object graph with membranes or a portion of execution with control flow propagation. Since the propagation is written reflectively, it can be customized to achieve other forms of scoping.

When using a proxy for access control or other security-related concerns, an important requirement is the absence of target leaking: it should be impossible to gain access to the target from the proxy itself. But like with any security mechanism that is implemented reflectively, it is important to ensure that reflection cannot be used to bypass them. Consequently it is important to secure the underlying reflective architecture. The next chapter will present an ownership-based access control policy to metaobjects that secure access to reflection. This access control policy permits to leverage the security benefits that proxy brings.

CHAPTER 4

Ownership-based Access Control to Reflection

Contents

In the last chapters, we saw how reflection allows programs to examine and modify their own structure and behavior. Reflection helps writing highly generic code and frameworks. It can alter program interpretation to create language extensions, to perform dynamic analyses and to factor non-functional concerns. Reflection is a solid fundament to support unanticipated changes, development tools, dynamic software updates and self-adaptive programs.

However, most reflective operations break object encapsulation, making it at the same time a bless and a curse. Reflection supports adaptable software thanks to scoped alterations of the language semantics, to dynamic code transformations and to generic code that works on an open-ended set of data-types. At the same time reflection power steams from its ability to bypass language rules such as encapsulation. As long as reflective code is written with genericity in mind (*e.g.* listing all the instance variables of an object instead of expecting it to have a specific instance variable), such encapsulation breaches are not a problem from the modularity point-of-view. However, as soon as we consider the collaboration of multiple software components developed by different parties, these encapsulation breaches become a security threat. Malicious code can use reflection to corrupt critical behavior of to gain access to protected operations. The canonical example is *object state introspection*, where clients break into the encapsulation boundaries of other objects to obtain new references. Object state introspection is a reflective operation available in many languages: `getattr()` in Python, `Field.get()` in Java, `instVarNamed:` in Smalltalk, `instance_variable_get()` in Ruby, etc.

To illustrate the problem we take an example that stresses the security issue of these encapsulation breaches. This example consists of two "person" objects, *Alice* and *Bob*, each holding a reference to its own private wallet object. In this context, if *Alice* has a reference to *Bob*, she can use object state introspection to access *Bob*'s wallet without his consent:

```
bobWallet := bob instVarNamed: #wallet.
```

With a reference to *Bob*'s wallet, *Alice* can now access its content, like its credit card. Such leaked object reference can in turn be introspected: from a reference to one object, all indirectly-connected objects become reachable. So once *Alice* has obtained *Bob*'s credit card she can introspect it to get its PIN:

```
creditCard := bobWallet creditCard.
pin := creditCard instVarNamed: #pin.
```

From a security point of view the tension between reflection and object encapsulation is problematic. An unrestricted access to reflective operations allows any code to inspect and corrupt any code loaded in the runtime or any computation running therein. For example, the deployment of an application that requires reflection support in a shared runtime relies on trusting this application to not misuse reflection. This situation is not satisfactory. Access to reflective operations needs to be controlled.

This tension is also problematic when reflection is used to implement security mechanisms [Ancona 1999, Riechmann 1997, Riechmann 1998]. In Chapter 3, we saw that proxies are good candidates to implement security-related behavioral variations. The problem here is that reflection, which is used to implement these mechanisms, can also bypass them. Consequently, a security mechanism implemented with reflection must be reflection-proof. A solution is to enforce policies that forbid reflection except for the implementation of these mechanisms. But this situation is not satisfactory because reflection cannot be used anymore for all its other applications, notably monitoring and analysis. Rather than forbidding reflection, or most of its useful applications, we want to control it to ensure that the encapsulation breaches happen only under certain conditions.

In this chapter, we reconcile reflection and object encapsulation via an access control policy to reflective operations. Such policy has to determine when breaking into the encapsulation boundary of an object is legitimate. To this end, we explore the notion of dynamic object ownership [Noble 1999, Gordon 2007] that organizes object graphs around a notion of ownership. We use the object ownership relation to determine access rights to reflective operations on a per-object basis. These access rights are thus based on the dynamic arrangement of objects rather than on static relations between structural entities (*e.g.* classes and packages) as it is the case for visibility modifiers in most languages. Ownership information becomes our basis to decide when it is legitimate for an object to break into the encapsulation boundary of another one using reflection. We show that within a reflective architecture that implements this policy, proxies (a reflectively implemented mechanism) can be used to enforce reflection-proof security policies.

The contributions of this chapter are:

- a description of the problem of object encapsulation violations caused by reflection, a presentation of the existing solutions and their shortcomings (Section 4.1);
- a presentation of an access control policy to reflective operations based on dynamic object ownership (Section 4.2);
- an evaluation of this access control policy in our object-centric MOP (Section 4.3).

4.1 Encapsulation Violation of Reflection

Most reflective operations break object encapsulation. To reconcile reflection and encapsulation we want an access control policy to reflective operations. The purpose of such policy is to decide when an object can legitimately use a reflective operation on another object and thus potentially break the encapsulation of the latter.

In this section we set up the context of the discussion to explain the tension between encapsulation and reflection. To stress this tension we analyse it through the point of view of a security model where object encapsulation is a central requirement: the *Object-Capability Model* (*OCap* model). The following brief introduction to the *OCap* model also introduces some vocabulary.

4.1.1 Object-Capability Model

The Object-Capability Model [Miller 2006, Miller 2003] is a capability-based security model that builds upon the object paradigm. In capability-based security, a capability is an unforgeable reference to a *resource* together with a set of access rights to this resource. Capabilities are *unforgeable i.e.* it is impossible to counterfeit a capability. A capability grants *subjects* holding it the *permission* to invoke operations of the associated resource according to the associated access rights. In a capability system it is impossible to designate a resource without having the permission to access this resource: a capability is at the same time a designation and a permission.

The *OCap* model applies capability-based security to object-oriented programming by treating objects both as subjects and resources. An object is a resource for objects holding a reference to it and a subject (or client) for objects it holds a reference to. In a memory-safe language a capability is encoded as an object reference: the absence of pointer arithmetic ensures that object references are unforgeable. A capability grants a subject the right to send messages to the resource. A capability can only be obtained by:

- *Introduction*: An object can pass one of its capabilities to another one by passing the former as an argument of a message to the latter.
- *Parenthood*: An object obtains the initially unique capability over an object it create.
- *Endowment*: An object can pass some its capabilities to an object it creates.[1]

[1] Endowment can be seen as the combination of introduction and parenthood.

In the context of our wallet example, if *Alice* holds a capability on *Bob*, she can send him messages. On his side, *Bob* encapsulates its capability to its wallet: he is the only one to decide if he gives its wallet away to strangers. That is to say, it is up to the code of *Bob* to take care of exercising its wallet capability himself and not to return or introduce this capability to collaborating objects. For example, if *Alice* were to ask *Bob* to pay her for some item, *Bob* would have to take some coins out of his wallet and give them to *Alice*, but he would not give away his wallet for *Alice* to take these coins herself.

Object encapsulation is crucial to the *OCap* model: a capability only permits a subject object to send messages to the associated resource object, but not to access the capabilities of the resource without its consent. In the introduction we saw that the global availability of object state introspection allows any object to access all the objects indirectly connected to it. This is unacceptable because any capability would bring as much authority than the sum of capabilities in its connected object graph.

The integration of reflection and the *OCap* model in a same language is thus challenging. Consequently *OCap* languages provide limited reflective abilities. For example, *Joe-E* [Mettler 2010], an object-capability subset of *Java*, limits reflection to introspection of public members. Another example is the *E* language [Miller 2006] that limits reflection to the execution and interception of message sends.

An *OCap* language could allow an object to perform many reflective operations on itself. An object inspecting its own state and behavior, instrumenting its own code or altering its own interpretation does not contradict the principles of the *OCap* model as soon as these reflective operations affect only the object itself. Only the reflective operations that an object performs on another object needs to be controlled. More precisely, an object should not be able to perform a reflective operation that produces an effect that could not have been carried out without reflection. This is the *reflection protection principle* formulated by De Meuter *et al.* in the context of the *ChitChat* language [De Meuter 2005]. The reflection protection principle states that, by default, an object does not expose more of itself at the meta-level than it does at the base-level. An object can still expose more at the meta-level than it does at the base-level, but only if it chooses to do so. The set of reflective operations that respect the reflection protection principle depends on the host language. For example in *Java*, the visibility of a field or a method has to be taken into account: reflection protection allows method calls, field readings and field writings but only if the corresponding member is accessible from the class of subject object according to the visibility restrictions. Another example is Smalltalk, where instance variables are private to each object and all methods are public. In this case reflection protection allows every message send but no access to instance variables. The reflection protection principle can be relaxed by allowing the listing of accessible members, even if it is not possible at the base-level. Indeed, listing all the accessible methods of an unknown object is a good way to discover its interface, a facility that is really useful for making generic code and adaptable software. Hence we give the following definition of the reflection protection principle: a reflective API respects the reflection protection principle if only the members that are accessible at the base-level can be accessed at the meta-level. Consequently, a reflective API that respects the reflection protection principle only forbids to access non-visible members. This definition allows for listing of accessible members.

While the reflection protection principle reconciles reflection with the *OCap* model, it also constrains reflection and limits its range of applications. For example the reflection protection principle limits the kind of behavioral variations that a proxy can implement since the proxy can only forwards permitted operations. Behavioral variations that have to forward object state accesses to the target, like the read-only variation that forward state read, are forbidden. We are looking for a more permissive approach to allow such kind of behavioral variations under certain conditions.

4.1.2 Reflection as Separate Capabilities

A first step toward the reconciliation of reflection and the *OCap* model is to encapsulate reflective operations in separate metaobjects. Likewise the capability to send messages to an object and the capability to reflect on this object are kept distinct. Additionally this separation prevent name clashes between the base-level operations and the reflective operations.

To allow fine-grained access control, a reflective architecture should provide metaobjects that grant a subject the right to perform reflection on a single object. This is the case of object-centric MOPs. For example, the reflective API of Java does not fulfills this requirement: in Java, the method `getClass()` returns a reification of the receiver's class (instance of the class `Class`), that allows subjects to inspect and modify any instance they reference. Reflective architectures that expose reflective operations via metaobjects can be divided in two categories: those where metaobjects are accessed directly from objects and those where metaobjects are acceded indirectly via an external provider.

An access control policy to reflective operations also has to face the problem of right propagation. If any subject object can retrieve the metaobject of any object, this metaobject becomes an entry point to the reflective system. The subject can ask this metaobject the class of its referent. If a class gives access to its subclasses and its superclass, all classes of the system become reachable from any object reference. If a class gives access to its instances, all objects of the system become reachable from any object reference. Because of the availability of these reflective operations, any capability grants the authority to perform any reflective operation on any object.

4.1.2.1 Direct Access to Metaobjects

Many reflective architectures grant base-level objects the right to access the metaobject of any other object directly. In this case, the separation of capabilities is not helpful since no access control is performed: the acquisition of a capability over a base-object allows for the acquisition of a capability over the metaobject. When the access to a metaobject is negotiated via message passing, and when the corresponding accessing method can be redefined, objects may restrict the set of reflective operations they provide [De Meuter 2005]. Typically, this kind of solution has two drawbacks.

The first drawback is a lack of principal information: an object has no easy mean to know which subject object is asking for its metaobject. One solution can rely on the different method visibilities provided by the language to grant different subjects different levels of authority. But in most languages method visibilities are based on static criteria, like the package or the hierarchy of the class defining the method. Such solution prevents fine-grained access control by making the assumption that all instances of a class have necessary the same rights. And more often than not, different instances need different access rights to properly encapsulate their respective private state from each other. In the context of our wallet example, restricting the reflective access of someone's wallet to instances of the `Person` class (like *Java*'s `private` modifier) would imply that *Alice* has the permission to access *Bob*'s wallet reflectively. Likewise, restricting the reflective access of linked-list nodes to instances of the `LinkedList` class would imply that any linked list has the permission to access the nodes of any other linked-list. Also, if a malicious program manages to take over one instance, it can then take over all other instances it can access. Such solution also makes the assumption that principal boundaries matches the visibility scopes provided by the language. A more satisfying solution is to take the identity of subjects into account for more fine-grained control. This is what the access controlled policy presented in this chapter does. Of course this is only possible if the language offers a way to know the sender of a message.

A second drawback is that this kind of access to metaobjects forces the repetitive redefinition of metaobject access methods and may lead to duplicated code. Also, developers may either implement over-restricting access policies and prevent the usage of development tools and dynamic analyses or implement over-permissive policies and introduce security breaches. To avoid these repetitive redefinitions, the default accessing methods must provide a sensible default access control policy. This is one of the contributions of this chapter.

4.1.2.2 Indirect Access to Metaobjects

Access to metaobjects can also be done indirectly via an external provider. This is the case in mirror-based reflective architectures [Bracha 2004], where all reflective operations are performed via metaobjects called *mirrors*. A mirror-based reflective architecture follows three design principles:

- Encapsulation: The implementation of reflective operations is encapsulated. It is then possible the substitute one implementation with another, *e.g.* for adapting existing development tools to a different runtime or to provide reflection on remote objects.

- Stratification: The meta-level is totally separated from the base-level. Mirrors are not accessed directly via base-level objects but instead via a *mirror factory*.
- Ontological correspondence: The reflective API describes the reflected language in its entirety and distinguishes between static and dynamic aspects of the language.

In our context, stratification is the most important principle. To perform reflection upon a resource, a subject needs a capability over the resource and a capability over a *mirror factory*. Without access to a mirror factory, a subject cannot use reflection at all. Typically, a default mirror factory creates mirrors that expose all available reflective operations. A capability over the default mirror factory thus grants a subject the right to reflect upon any object it references either directly, or indirectly via transitive encapsulation breaches.

Thanks to the adherence to the *Abstract Factory* design pattern [Gamma 1995], mirror-based reflective architectures make it possible to design custom mirror factories that produce mirrors with less authority. But such mirror factories still need to keep track of access permissions to determine which rights it grants to different subjects. Our policy uses ownership information to keep track of these permissions.

4.2 Access Control Policy to Reflective Operations

An access control policy to reflective operations has to decide when a subject object can legitimately perform a reflective operation on a resource object. Ideally such policy should be permissive enough to retain most of the power of reflection and restrictive enough to prevent abuses. A policy should also be generic enough to be a sensible default for most situations.

Consider the case where the subject and the resource are the same object. In this case we consider all reflective operations legitimate, *i.e.* an object can always access its own metaobject. An object should have the right to decide for itself how it behaves: it should have the right to inspect and alter its own state as it intends. The problematic case is when the subject and the resource are different objects. Specifying access rights on a per-class basis is not a satisfactory solution because all instances of a class would have the same rights. Rather our policy takes the dynamic relations between objects into account. We can observe that the relation between *Alice* and *Bob* is not the same than the relation between *Alice* and her wallet. While *Alice* should not be able to get access to *Bob*'s wallet, she should have a privileged access to her own wallet and then to the credit card it contains. She should be able to use reflection to alter the behavior of her credit card. Likewise she can monitor and limit withdrawals when she lands it to an untrustworthy friend. The relationship between a person and his wallet is stronger than the relationship between persons: while *Alice owns* her wallet, she simply *refers* to *Bob*. The distinction between this *owning* relationship and the standard *referencing* relationship is embodied by the concept of *Object Ownership*.

4.2.1 Object Ownership

Object ownership was originally introduced to control the effects of object aliasing in the context of *Flexible Alias Protection* [Noble 1998]. It was first embodied as a type system with ownership types [Clarke 1998] and was then adapted in the context of dynamically-typed languages with dynamic ownership [Noble 1999, Gordon 2007]. In this paper we consider the latter. Note that we do not use object ownership to control object aliasing but to keep track of reflection permissions on a per-object basis.

The notion of object ownership comes from the observation that objects are rarely autonomous but instead form aggregates. An aggregate object is composed of objects that constitute its *representation* and refers to other objects that constitute its *arguments*. An aggregate owns its representation but simply refers to its arguments. A typical example is a linked-list: the list owns its nodes (its representation) but simply refers to the elements it contains (its arguments).

Figure 4.1 shows the ownership relation in the context of our wallet example. *Alice* and *Bob* refer to each other, and both own their respective wallet and credit card. Such object graph could be the result of the following code.

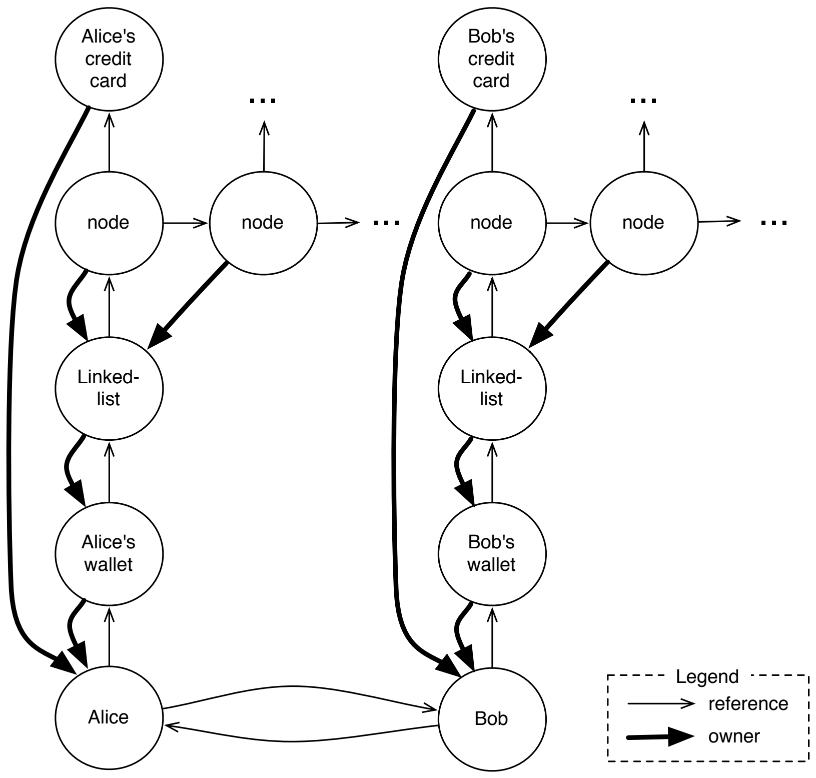


Figure 4.1: An example showing the ownership relation. *Bob* and *Alice* each own their own wallet and credit card. Each wallet owns its respective linked-list and each linked-list own its respective nodes. *Bob* and *Alice* only refer to each other just like each node refers to its contents.

```
Person>>initialize
   wallet := Wallet new.
   wallet add: CreditCard new

alice := Person new.
bob := Person new.
alice friend: bob.
bob friend: alice
```

4.2.2 Access Control to Reflective Operations

We now present our ownership-based access control to reflective operations. Each object refers to its *direct owner*, which by default is the object that instantiated it.[2] This forms the *direct ownership* tree. Depending on the execution model of the host language, the root of this tree is either the first object instantiated or a special object. For example, for *Java* there would need to be a special object to act as the owner of objects instantiated within the `main` method. The *ownership* relation we consider is the transitive reflexive closure of the *direct ownership* relation. In other word, we say that an object *A owns* an object *B* if:

- *A* directly owns *B* (*i.e.* *A* is the direct owner of *B*), or
- there exists an object *C* such that *A* owns *C* and *C* directly owns *B*, or
- *A* and *B* are the same object.

The access control policy is to grant an object the permission to perform any reflective operation on the objects it owns. For example, in the context of Figure 4.1, *Alice* owns her credit card and can alter its behavior to monitor or limit withdrawals. If the subject object does not own an object the policy falls back to the reflection protection principle *i.e.* only the reflective operations that perform an action that could have been carried out without reflection are available. Reflection is not entirely forbidden: the subject object can still perform some reflective operations. In the context of Figure 4.1, *Alice* does not own *Bob* but can still reflectively send messages to *Bob*. But she cannot introspect *Bob* to obtain a capability to his wallet.

4.2.2.1 Root Object as Superuser

Because the direct ownership relation forms a tree, the object at the root indirectly owns all objects. Consequently, this root object has the permission to perform any reflective operation on any object. This root object can thus use development tools and perform dynamic analyses on any object. For example, an application in production should be able to monitor all `hash` messages sent from each hash-table

[2] Other methods are possible to specify the direct owner of an object *e.g.* with annotations.

to determine which `hash` methods are slow or produce too many collisions. To perform such analysis this application should be able to reflect unrestrictedly on all its objects. At the same time, that same application may use a library that it does not trust completely. This library can use reflection but only on its own objects.

4.2.2.2 About Ownership Transitivity

Unlike other notions of object ownership our ownership relation is transitive. This transitivity stems from the power of reflective abilities. Consider three objects *A*, *B* and *C* such that *A* directly owns *B* and *B* directly owns *C*. Even without transitivity, *A* could exercise its full reflective power over *B* to change its behavior at will. Doing so, *A* could easily obtain *B*'s authority to have full reflective power over *C*.

4.2.2.3 Formalization

We can formalize our ownership-based access control to metaobjects by redefining the semantics of MOPLite. In Subsection 2.3.1 an object was encoded as a triplet: $\langle cls, adr_{mo}, ivs\rangle \in \mathcal{C} \times \mathcal{A} \times (\mathcal{I} \rightharpoonup \mathcal{A})$ The first element, cls, is the class of the object, $adrmo$ is the address of the metaobject and ivs is a mapping from instance variables identifiers to addresses. To encode ownership objects are now a quadruplet to add the address of the owner of the object (adr_{own}).

$$\langle cls, adr_{mo}, ivs, adr_{own}\rangle \in \mathcal{C} \times \mathcal{A} \times (\mathcal{I} \rightharpoonup \mathcal{A}) \times \mathcal{A}$$

The direct ownership relation and its transitive reflective closure are respectively noted $\blacktriangleright$ and $\blacktriangleright^*$, *i.e.* $a \blacktriangleright b$ means that the direct owner of (the object pointed by the address) a is (the object pointed by the address) b.

The instantiation rules [*new*] and [*new-mo*] are redefined to set the current object (the one that executes the instantiation) as the owner of the new object. Modifications to the original rules are underlined.

$$\left\langle \left[\begin{array}{c} \langle E[\ cls\,\texttt{.new(}adr^*\texttt{)}\ ], adr, cls\rangle \\ \vdots \end{array}\right], S \right\rangle$$

$$\hookrightarrow \left\langle \left[\begin{array}{c} \langle E[\ adr_{new}\texttt{.init(}adr^*\texttt{) ; }adr_{new}\ ], adr, cls\rangle \\ \vdots \end{array}\right], S' \right\rangle \qquad [\ \textit{new}\ ]$$

$$\text{where } \begin{array}{l} S' = S[adr_{new} \mapsto \langle cls, \texttt{nil}, \{id_{iv} \mapsto S(\texttt{nil}) \mid \forall id_{iv}.\ id_{iv} \sqsubset_p^{hrc} cls\}, \underline{adr}\rangle] \\ adr_{new} \notin \mathrm{dom}(S) \end{array}$$

The access control policy then concerns reduction rules that gives access to metaobjects. The rules [*get-def-mo*] and [*get-mo*] are redefined to add the condition that the current object owns the object being requested its metaobject (*i.e.* $adr_{targ} \blacktriangleright^* adr$).

$$\left\langle \left[\begin{array}{c} \langle E[\ adr_{targ}\texttt{.meta}\ ], adr, cls\rangle \\ \vdots \end{array}\right], S \right\rangle$$

$$\hookrightarrow \left\langle \left[\begin{array}{c} \langle E[\ \text{SETMO}(adr_{targ}, \texttt{DefaultMO.new()})\ ], adr, cls\rangle \\ \vdots \end{array}\right], S \right\rangle [\ \textit{get-def-mo}\ ]$$

$$\text{where } \frac{adr_{targ} \blacktriangleright^* adr}{S(adr_{targ}) = \langle ..., \texttt{nil}, ..., ...\rangle}$$

$$\left\langle \left[\begin{array}{c} \langle E[\ adr_{targ}\texttt{.meta}\ ], adr, cls\rangle \\ \vdots \end{array}\right], S \right\rangle$$

$$\hookrightarrow \left\langle \left[\begin{array}{c} \langle E[\ adr_{mo}\ ], adr, cls\rangle \\ \vdots \end{array}\right], S \right\rangle \qquad [\ \textit{get-mo}\ ]$$

$$\text{where } \begin{array}{l} \underline{adr_{targ} \blacktriangleright^* adr} \\ S(adr_{targ}) = \langle ..., adr_{mo}, ..., ...\rangle \\ adr_{mo} \neq \texttt{nil} \end{array}$$

Finally, a rule [*get-wrapped-mo*] is added for the case the current object does not own the object being requested its metaobject (*i.e.* $adr_{targ} \not\blacktriangleright^* adr$) In that case, the metaobject is wrapped into an instance of the core class `MOWrapper`, defined as follow:

```
class MOWrapper extends Object {
  wrappedMO
  init(mo) { wrappedMO := mo }
  receive(sel,args,cls) { wrappedMO.receive(sel,arg,cls) }
}
```

It should not be possible to use reflection to breach the wrapper encapsulation and gain access to the wrapped metaobject. To ensure this, the owner of a wrapper is the referent of its wrapped metaobject. If an object asks for a metaobject gets a wrapper, it means that it does not own the referent. So if this same object asks for the metaobject of the wrapper, it will get another wrapper wrapping the metaobject of the first wrapper. To ensure that the target object is the owner of the wrapper, a new stack frame is pushed where it is the current object.

$$\left\langle \left[\begin{array}{c} \langle E[\ adr_{targ}\,\texttt{.meta}\ ], adr, cls\rangle \\ \vdots \end{array}\right], S \right\rangle$$

$$\hookrightarrow \left\langle \left[\begin{array}{c} \langle E[\ \texttt{MOWrapper}.new(\texttt{self.meta})\ ], adr_{targ}, \texttt{Object}\rangle \\ \langle E[\ adr_{targ}\,\texttt{.meta}\ ], adr, cls\rangle \\ \vdots \end{array}\right], S \right\rangle [\ \textit{get-mo}\ ]$$

where $\underline{adr_{targ} \not\blacktriangleright^{*} adr}$

4.3 Implementation

We now present the implementation of our ownership-based access control to metaobject in our object-centric MOP. We discuss how ownership is encoded and customized, and present the context *subject-sentitive* method `meta` that gives access to metaobjects.

4.3.0.1 Ownership Encoding

Conceptually, encoding ownership information consists in adding a `directOwner` instance variable to the root class `Object`. In practice, *Pharo*'s virtual machine imposes some limits on the memory layout of some special classes so we use an external table to map objects to their direct owner instead.

To initialize the direct owner of a newly-created object, this object receives the message `initializeDirectOwner` after instantiation. The default implementation of `initializeDirectOwner` determines the direct owner of the newly-created object by traversing the call-stack. The receiver associated with each stack frame receives the message `wantsOwnership:` with the new object as argument: the first receiver that answers `true` become the owner of the new object. The default implementation of `wantsOwnership:` in `Object` unconditionally returns `true` but a class can redefine this behavior if needed.

4.3.0.2 Customisation of Ownership Tree Construction.

The construction of the ownership tree can be customized for different situations. In object-oriented design, some engineering practices, like *Dependency Injection* frameworks and design patterns such as *Factory Method* and *Abstract Factory*, strive for reducing coupling between software components. These practices have in common that instantiation is performed by a third party. If these practices are followed systematically, no object instantiates objects that are part of their representation. Without special care, the resulting ownership tree would be very large and very shallow. In this context, the object that instantiates another object has to be able to let its clients become owner of that newly-created object or to transfer ownership.

For example, a factory can let its clients become the owners of the object it creates by redefining `wantsOwnership:` to answer `false` for the classes it instantiates. For example, a widget factory would implement `wantsOwnership:` as follows.

```
WidgetFactory>>wantsOwnership: anObject
    ^ (anObject class inheritsFrom: Widget) not
```

Another situation where customisation of object ownership is desirable is when an *inversion of control* scheme takes place. This is for exemple the case of dependency injection frameworks where the framework has to be able to transfer ownership of the object it creates. Our implementation allows changing the direct owner of an object. This operation is only available from an object's metaobject, so only an owner of an object can change its direct owner.

Another situation where the ownership tree construction needs to be customized is when a class wants all its instances to be owned by the same object. This is the case of immutable objects denoting values, like numbers, characters, points, etc, that are unconditionally owned by the object `nil`, the root of the ownership tree. This saves some space by not using the ownership map for these objects.

4.3.0.3 Context-Sensitive Method `meta`

Our ownership-based access control policy grants an object the permission to access the metaobjects of the objects it owns. When a subject requests the metaobject of a resource it does not own, the policy is to restrict the available reflective operations. Only the reflective operations that perform an action that could have been carried out without reflection are allowed, following the reflection protection principle. In the context of Pharo, where all instance variables are object-private and all methods are public, these restrictions only allow message reception. In our implementation, wrapper objects encapsulate metaobjects to enforce these restrictions. The method `meta` implements the access control to metaobjects. This method is context-sensitive: it checks whether the sender of the message is an owner of the receiver or not. If the sender is not an owner the metaobject is wrapped.

These access rules extends to metaobjects returned by reflective operations. For example, if the metaobject of an object *A* is about to return the metaobject of another object *B* that *A* does not own, the metaobject of *A* wraps the metaobject of *B* before returning it.

Alternative Design. Our access control policy does not need to be implemented with metaobject wrappers. The restrictions could be implemented in the metaobject directly. We choose the wrapper approach because it has two advantages. First, the implementation is simpler because only the method `meta` is context sensitive. If the restrictions were implemented in the metaobjects directly, each method would have to check whether the corresponding reflective operation is permitted or not. This would imply code duplication and slower execution. Second, an object can choose to pass the metaobject of an object it owns to an external object it trusts. If the restrictions were implemented in the metaobjects, they would apply to the external object and prevent delegation of reflective rights as capabilities.

4.3.1 The Policy at Work

We now show our policy at work, first with proxies, a reflective mechanism that can be used as an access control mechanism, then in the context of a reflective tower. Proxies shows that our policy enables the reflective implementation of a security mechanism and prevents reflection to be used to bypass this mechanism. Reflective towers shows that our policy can control reflection at any meta-level.

Proxy-based Behavioral Intercession. Proxies are good candidates to implement access control. A proxy can interpose an arbitrary policy between a client and a resource and implement *OCap* patterns like caretakers, facets or membranes [Miller 2006, Van Cutsem 2013].

The creator (*i.e.* the direct owner) of a proxy has to be sure that no object apart its owners can bypass the policy the proxy enforces. In Figure 4.2 *Alice* asks *Bob* to lend her his credit card. Because *Bob* does not trust *Alice* entirely, he creates a proxy for his credit card to limit withdrawals to a certain amount. *Alice* hopes to bypass the withdrawal policy imposed by the proxy. To do so she asks the proxy for its metaobject to then access the metaobject of the proxy's target (*i.e.* the credit card). But since *Alice* does not own the proxy, she obtains a wrapper on the proxy's metaobject instead. The wrapper allows her to send messages to the proxy (as she can do at the base-level) but do not permit her to access the state of the proxy's metaobject to obtain a reference to the metaobject of *Bob*'s credit card.

But the owner of a proxy does not have to own the target. In that case the proxy owner can only access a wrapper of the target's metaobject and only message sends interceptions can be passed to the wrapper. This is enough to implement capability patterns such as caretakers, facets and membranes.

Reflective Tower. When performing reflection upon the meta-level, one attains the meta-meta-level (or meta-level 2). This is useful to record which meta-level operations are performed on an object (meta-tracing for optimization), to query the meta-level behavior for analysis, to debug the implementation of the meta-level, etc.

More generally one can reflect on the meta-level n at the meta-level $n + 1$. This virtually infinite stack of meta-levels is called a *reflective tower* [Smith 1984]. Each level interprets operations performed at the level below. The tower is not actually infinite: the host language interpreter takes over level reification whenever operations at a level n are interpreted according to the default semantics.

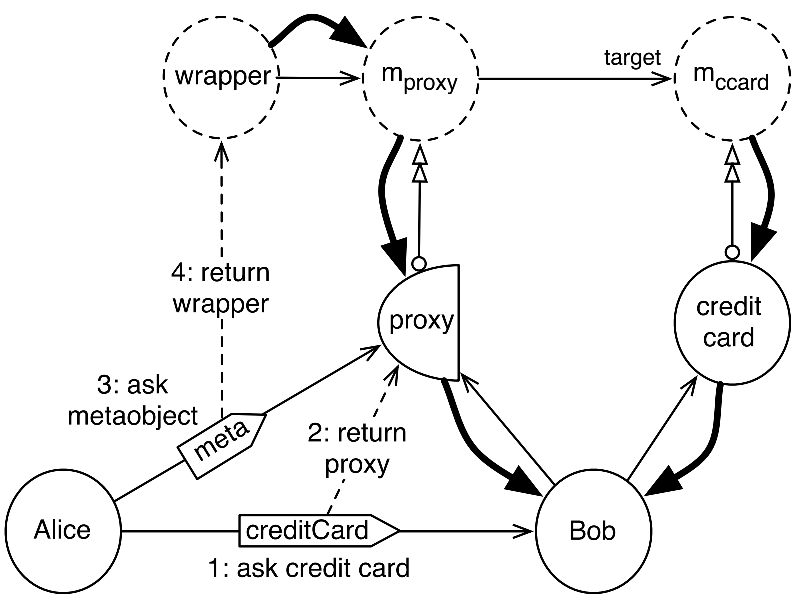


Figure 4.2: Proxy protection: (1) *Alice* asks *Bob* for his credit card, (2) *Bob* creates a proxy for his credit card to limit withdrawals and returns it, (3) *Alice* asks for the metaobject of the proxy to leak the credit card (target of the proxy), (4) but since she does not own the proxy she obtains a wrapper on the metaobject of the proxy and cannot leak the credit card.

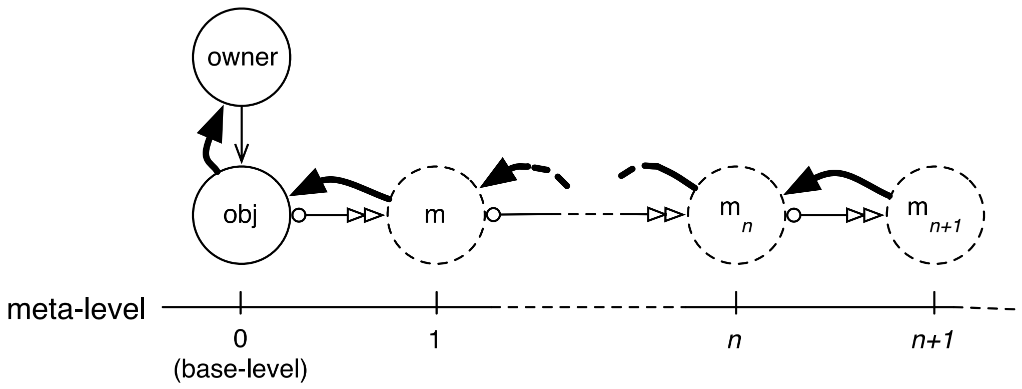


Figure 4.3: A chain of metaobjects forming a reflective tower with their respective owner. The owner of the object can reflect on this object at any level.

Reflective towers appear naturally in the context of an object-centric MOP. Each base-level object is the first floor of its own reflective tower. The metaobject of a base-level object belongs to meta-level 1. But like other objects, the behavior of a metaobject is also defined by another metaobject, which belongs to meta-level 2. The reflective tower is embodied by a virtually infinite chain of metaobjects as depicted in Figure 4.3.

The ability to jump from one meta-level to the meta-level above should not jeopardize the access control policy. That is to say that if the access control policy states that an object *A* has restricted access rights to the metaobject of another object *B*, *A* must not be able to circumvent this restriction by manipulating the metaobject of that metaobject. To ensure that, we customize ownership tree construction so that the direct owner of a metaobject is always its referent. Thereby, a metaobject at meta-level n owns its metaobject at meta-level $n + 1$. Consequently, an object owns all the metaobjects of its tower. It follows that only the owners of an object have the authority to reflect on this object at any level.

4.4 Related Work

Object Ownership. A lot of research has been done on object ownership through ownership types [Clarke 1998, Clarke 2001] and dynamic object ownership [Noble 1999, Gordon 2007]. Originally, object ownership was devised to control the effect of object aliasing. Later, many different ownership systems have been used for many other applications [Clarke 2013]: concurrency control, memory management, security, etc.

Since our access control policy relies on ownership information, it would be interesting to leverage this information with other applications. The first application that comes to mind is object alias control since it is the original application of object ownership. So an interesting question is to know if an effective alias control discipline can be compatible with our access control policy. The precise design of such alias control discipline remains an open question but we give some trails.

Such an alias control discipline has to rely on a transitive ownership relation. *Flexible alias protection* [Noble 1999, Clarke 2001] enforces two rules to control the effect of aliasing.

- The *no representation exposure* rule prevents the clients of an aggregate to access its representation.
- The *no argument dependence* rule prevents an aggregate to depend on the mutable state of its arguments.

With our ownership-based access-control to metaobjects, the *no representation exposure* rule cannot be enforced when the client is an owner, at least at the meta-level. Indeed, an owner can use reflective operations to access the representation of the aggregate, and transitively the representation of that representation. The *no representation exposure* rule can still be enforced at the base-level but this contradicts with the reflection protection principle.

Secure MOP. Caromel *et al.* [Caromel 2001b, Caromel 2001a] presents concerns about MOP and security in the context of component-based applications in *Java*. These works do not concern the access control to reflective operations but the security implications of implementing a MOP within the security framework of *Java*. The security framework of *Java* is based on inspecting the call stack to determine whether the execution of a sensible operation is permitted or not. In this context, the type of MOP used greatly influences the necessary propagation of permissions from meta-level code to base-level code. This is problematic because the implementation of a MOP typically requires many permissions. The authors show that within a proxy-based run-time MOP, it is possible to capture the set of permissions before reflective calls and restore them after. MOP implementation gets more permission than base-level code even though the call stack contains stack frames from different levels.

4.5 Conclusion

We explored the problem of encapsulation violation caused by reflective operations and its implications on the *OCap* model. The tension between the need for object encapsulation on the one hand and the need of reflection on the other hand led us to the conclusion that we need a way to track when breaking into an encapsulation boundary is legitimate. To this end we have explored the concept of dynamic object ownership that has been originally used to tame object aliasing. Instead of object aliasing, we showed how this notion of object ownership can be used to design an access policy to reflective operations. Thanks to this access control policy, owners of an object can perform any reflective operations on that object. An object that does not own another target object can only perform reflective operations whose effect could have been carried out without reflection. This simple policy reconcile reflection and security in the context of multiple interacting software components.

We implemented this policy in the context of a prototype MOP. In this context, we showed that this policy can ensure that domain-level access-control policies, implemented reflectively with proxies, cannot be bypassed using reflection at any meta-level.

CHAPTER 5

Scoped Extension Methods

Contents

Previous chapters studied the usages of reflection for adaptability, the encapsulation problems it brings and proposed an access control policy to reflective operations based on object-ownership that prevents harmful encapsulation breaches. This chapter studies the tension between encapsulation and another mechanism for adaptability: extension methods.

An extension method is a method declared in a package for a class declared in another package. Extension methods allow developers to adapt classes they do not own to their need. Adding convenience methods to standard library classes is a typical use case. This mechanism is particularly popular in dynamically-typed languages.

However, in typical implementation, extension methods are globally visible. Because any developer can define extension methods for any class, conflicts occur when two extension methods with the same signature are defined for the same class. Implementations can forbid conflicts but this constraint poses problems in practice. Indeed, when considering package co-evolution in large projects, the addition of one extension method can invalidate the project configuration. Typical implementations allows these conflicts. In this case only one extension method is visible and overwrites the other. Conflicts can also occur across class hierarchy, introducing potentially erroneous overriding relationships. Because they concern whole class hierarchy these kind of conflict is more likely to occur.

To avoid conflicts, some implementations limit the visibility of an extension method to a particular scope. However, their semantics have not been fully described and compared. In addition, these solutions typically rely on a dedicated and slow method lookup algorithm to resolve conflicts at run time.

This chapter first introduces extension methods and gives some common use cases. We then show some problems caused by the typical global visibility of extension methods, and conclude that extension methods must be local to their users. The core of this chapter is an analysis of different local extension methods mechanisms for dynamically-typed languages. We compare four approaches to local extension methods: Ruby refinements, Groovy categories, Classboxes, and Method Shelters. We study their implication on encapsulation and formalize their semantics in the context of MOPLITE. We show that lexically-scoped extension methods yields less conflicts, respects encapsulation and are easier to reason about. Finally, we propose an efficient implementation technique for lexically-scoped extension methods that leverages the lookup-algorithm of single-dispatch languages using a variant of name-mangling.

5.1 Introduction to Extension Methods

Extension methods are a popular feature in object-oriented languages. An extension method is a method that a developer adds to a class he does not own, *e.g.* a class from the standard library of the language or a framework he uses. Variants of extension methods are available in many dynamically-typed languages: they are known as *open classes* in Ruby, *categories* in Objective-C and Groovy, and *extension methods* in Smalltalk and Pharo.

Nevertheless, in most existing implementations extension methods are globally visible. This causes two problems: *accidental overwrites* and *accidental overrides*. An *accidental overwrite* happens when two developers define an extension method with the same signature in the same class: in this case a conflict occurs and one method overwrites the other. An *accidental override* happens when two developers define an extension method with the same signature in two classes related by inheritance: one method overrides the other.

Another common problem is the absence of dependency declaration between extension methods and their callers. Together with the global visibility of extension methods, this promotes the emergence of hidden dependencies that are difficult to track, especially in a dynamically-typed language.

One way to solve these problems is to assign each extension method a particular scope. Variants of scoped extension methods have already been discussed in the literature with the Classbox model [Bergel 2003, Bergel 2005b, Bergel 2005a] and the Method Shelter model [Akai 2012] and been implemented in Ruby since version 2.1 and in Groovy. These variants, however, have different semantics that must be well understood by the developers. There is no clear description and comparison of their semantics as well as pros and cons of their impact on applications. In addition, these variants rely on dedicated method lookup algorithms to resolve conflicts at run time and tend to have a negative impact on performances.

Other solutions have been proposed in the context of statically-typed languages [Clifton 2000, Warth 2006, Ducournau 2007] but are not portable to dynamically-typed languages as they rely on static type information. The difficulty in dynamically-typed languages is to determine *method families* at call sites, *i.e.* the set of methods that can be activated by a particular message send.

5.1.1 Usage of Extension Methods

We show different use cases of extension methods with examples taken from *PetitParser*, a parser combinator library for Smalltalk [Renggli 2010]. In such a library, a parser combinator accepts one or several parsers and produces a new composed parser. Examples of combinators include "`,`" to sequence two parsers, "`star`" to repeat a parser zero or more times and "`not`" to negate a parser.

As Syntactic Sugar. In addition to these combinators, *PetitParser* defines convenient `asParser` extension methods to some core classes. These extension methods create parsers depending on the receiver (see Figure 5.1). For example, the `asParser` extension method defined in the class `Character` returns a parser that

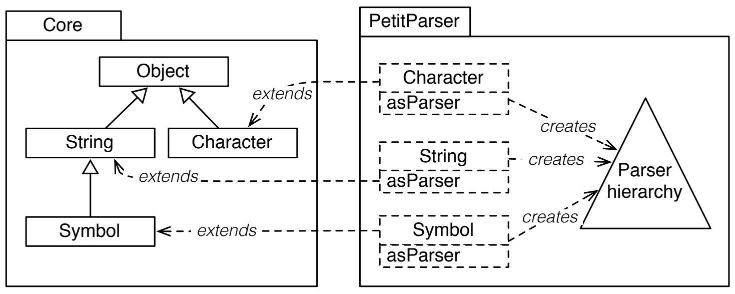


Figure 5.1: The PetitParser parser combinator library defines `asParser` extension methods on core classes to create various kinds of parsers.

accepts the receiver character. Together with combinators, these extension methods give a readable DSL-like syntax. For example, the following expression returns a parser object that accepts regular expressions of the form `ab*`.

```
$a asParser , $b asParser star
```

Here, the `asParser` extension method act mainly as syntactic sugar. The next expression returns an identical parser.

```
(PPLiteralObjectParser on: $a) asParser , (PPLiteralObjectParser on: $b) asParser star
```

To Improve Extensibility. By changing slightly the previous example, we can see that extension methods can also improve code quality. Consider the following code:

```
MyParser>>one: a thenMany: b
   ↑ a asParser , b asParser star.

MyParser>>id
   ↑ self
      one: #uppercase
      thenMany: #letter

MyParser>>int
   ↑ self
      one: ('1' to: '9')
      thenMany: #digit
```

In the `MyParser` class, the `one:thenMany:` method takes as parameter two objects that can be converted into parsers and returns a new parser. The `id` and `int` methods use that first method to build custom parsers. The method `id` sends the message `one:thenMany:` with two symbols (`uppercase` and `letter`) while the method `int` sends the same message with an interval and a symbol. The method `one:thenMany:` does not have to care about the class of its arguments. The only requirement is that arguments understand the `asParser` message. Here, extension methods improve extensibility and modularity by making unrelated classes polymorphic. Any developer can add the method `asParser` to any class and then pass instances of this class to the `one:thenMany:` method.

To Adapt Classes Interfaces. The Adapter pattern [Gamma 1995] adapts the interface of an existing class to work with other classes without modifying its source code. Its classic realization consists in wrapping instances of the adapted class into instances of an adaptor class. When there is no name conflicts between the provided and required interfaces, extension methods offers an alternative without relying on an adapter class. Instances of the adapted class can be used directly as they do not need to be wrapped with an adapter object. Client code doesn't need to be updated to explicitly wraps object when needed.

To Handle Unanticipated Changes. Extension methods are also useful to handle unanticipated changes. For example, if a package defines a class hierarchy not prepared for the Visitor design pattern [Gamma 1995] another package can add `acceptVisitor:` extension methods.

Monkey-Patching. If a third-party library or framework has a bug, developers can create an overwriting extension methods to correct the bug, a technique known as *monkey patching* [1]. While occasionally useful, monkey patching is often considered a bad practice in developer communities (as the name implies).

[1]The term *monkey patching* is sometimes used as a synonym of globally-visible extension methods. Here we use a narrower definition that refers to one way of using extension methods.

5.1.2 Problems of Globally Visible Extension Methods

Most implementations of extension methods, such as the ones of various Smalltalk dialects, Ruby (before the introduction of *refinements*) and Objective-C, make extension methods globally visible. This can lead to undeclared dependencies, accidental overwrites and accidental overrides.

Undeclared Dependencies. Once an extension method is loaded, it is visible from everywhere. The method can be called from any class of any loaded package without any form of declaration. This means that an application can work correctly in the developer's environment and fail once deployed because the application depends on an extension method from a non-loaded package. The absence of declaration favors the emergence of these hidden dependencies.

Accidental Overwrites. Extension methods defined by different packages may conflict in two different ways. The first kind of conflict arises when two packages each define an extension method with the same signature in the same class. In this case, one extension method replace the other. We call this situation an *accidental overwrite* (this is unlike monkey patching where the overwrite is intentional).

An example is given in Figure 5.2. A package `SimpleLog` declares an extension method `log` for the class `Object`. This package is a logging framework that records the string representation of an object in a log file. The package `ObjectLog` declares another extension method `log` for the class `Object`. This latter package is another logging framework that serializes objects in a log file for later analysis. Because these two extension methods conflict the two logging frameworks cannot be loaded at the same time. Even though these name clashes happen sparingly in practice, they are also difficult to anticipate, especially when considering package coevolution in large projects using dozens of packages.

Accidental Overrides. The second kind of conflict arises when a method overrides another method defined higher in the same class hierarchy. Figure 5.3 gives two examples of such overrides with extension methods.

To the left, a regular method `log` in package `Math` accidentally overrides an extension method in its superclass declared in package `Logger`. While `Logger`'s

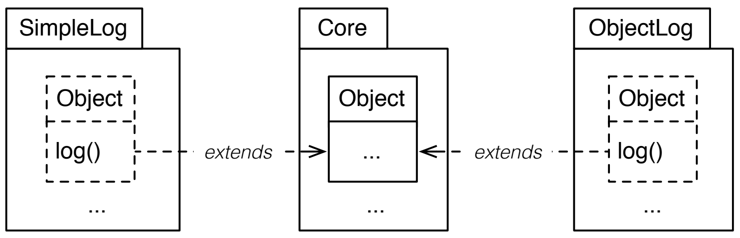


Figure 5.2: An exemple of accidental overwrite. Two packages each declare an extension method `log` for the class `Object`.

extension method `log` prints the receiver object in some log file, `Math`'s extension method `log` computes the logarithm of a number. Users of package `Logger` expects that when they send a message `log` to an object, the extension method of `Logger` is taken into account. However, `Number`, as a subclass of `Object` overrides the `log` method in package `Math`.

To the right, an extension method in package `Math` overrides another extension methods in package `Logger`. In this situation the packages `Math` and `Logger` are unaware of each other: none of them know that `Math`'s extension method overrides `Logger`'s.

These accidental overrides are more insidious than accidental overwrites. First, as they involve a whole class hierarchy they potentially happen more often. Then, an accidental overwrite is easily noticeable because the client packages are likely to break upon the first invocation of the overwriting method. Accidental overrides are much less noticeable because they affect only instances of the class defining the overriding method. When multiple parties can enhance the interface of any class, one party should not be able to override the methods defined by an unrelated party it is not aware of.

A big task of programming is to put word on program entities according to the terms of the business domain. Large programs usually involve multiples concerns and domains, each coming with its own terminology. Even inside the business domain of a single program several terms may have different meanings in different contexts. Large programs involve many different concerns and domains with different terminologies and overlapping terms. We call an override "accidental" when it is not intended by the programmer. Another way to put it is to say that accidental overrides happen when two or more concerns with overlapping termi-

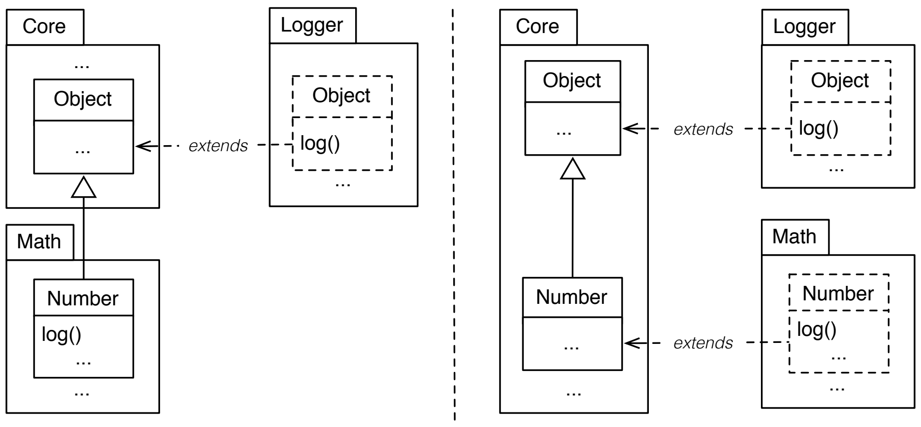


Figure 5.3: Two examples of accidental overrides. To the left, a regular method accidentally overrides an extension method. To the right, an extension method accidentally override another extension methods.

nologies spread over the same class hierarchy. In the context of extension methods, the probability of an accidental override is large because any package can declare an extension method for any class. When considering package coevolution, this becomes really problematic. These accidental overrides are a form of interference between packages.

To sanitize extension methods, accidental overrides must be limited, the visibility of extension methods also needs to be limited and dependencies need to be declared. In the next section, we study several scoped extension methods mechanisms.

5.2 Scoped Extension Methods

Because extension methods with global visibility exhibit the above-mentioned problems, several implementations propose a narrower visibility. We briefly describe four of these solutions. Depending on the solutions, the scope of activation of extension methods is either lexical or dynamic. In solutions where the scope of activation is lexical, the set of extension methods that are active at a given point is determined statically. In solutions with a dynamic scope of activation, the set of extension methods active at a given point is contextual. Dynamic scoping is necessary to support a property called *local rebinding* [Bergel 2003, Bergel 2005b], that allows extension methods to override other methods in a contextual manner. First we present the local rebinding property, its strenght and its weaknesses. Then, we show three solutions that expose the local rebinding property: *Classboxes* [Bergel 2003, Bergel 2005b], *Method Shelters* [Akai 2012] and Groovy's *categories*. Finally, we present Ruby's *refinements* where extension method activation is determined lexically.

In the following, we use the following terms:

Package. We call package the language-specific unit of deployment that gathers definitions of classes and other modular constructs from the language. Different packages are potentially maintained by different parties. A package also declares dependencies toward other packages by importing some definitions.

Extension method. An extension method is a method that is defined in a package for a class defined in another package.

Class extension. A class extension is a named set of extension methods that apply to the same class. We do not consider addition of instance variables.

Extension. Finally, an *extension* is a named set of extension methods that may apply to different classes.

5.2.1 Local Rebinding

Local rebinding is a property of a method-lookup algorithm first discussed in the context of the *Classbox* model [Bergel 2003, Bergel 2005b] and then in the context of the *Method Shelter* model [Akai 2012]. This property permits extension methods to override other methods in a contextual manner, *i.e.* an active extension method takes precedence over regular methods, even for indirect calls. This is similar to traditional method overrides in a subclass except that with traditional overrides, only the instances of the subclass are affected by the redefinition. With local rebinding, the same objects behave differently in different contexts. In Figure 5.4, the `MyExtendedApp` package redefines the `printIndentation(int)` method to print spaces instead of tabs. When invoked from within this package, this redefinition

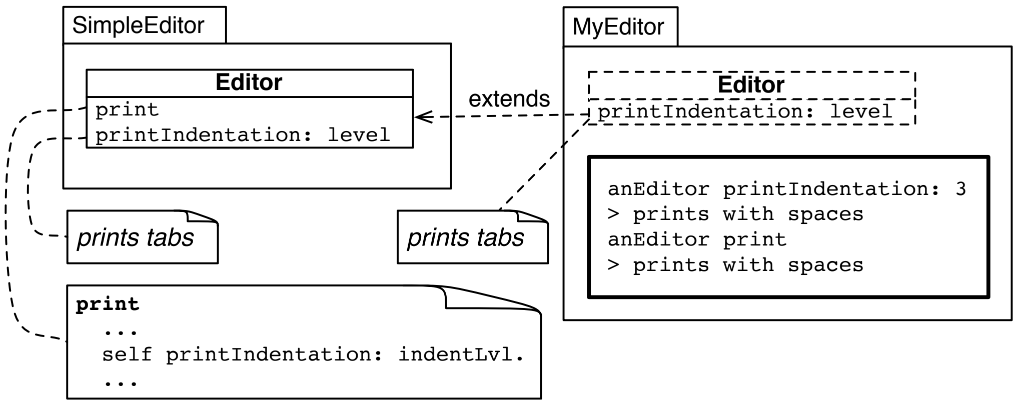


Figure 5.4: With local rebinding, changes made by an extension method are applied in case of indirect calls.

is taken into account, even in indirect calls: when invoking the `print()` method defined in the `MyApp` package, the redefined version of `printIndentation(int)` will be executed and not the one defined in the `MyApp` package.

With local rebinding, the lookup algorithm dispatches to different methods in different contexts. More precisely, when the signature of an extension method *e* matches the one of a method *m* of the extended class, local rebinding ensures that *e* overrides *m* during the dynamic extent of message sent by importers of *e*. The method lookup algorithm has to access this contextual information to determine the active extension methods. Two ways to implement such method lookup algorithms are by inspecting the call stack or by storing the set of active extension methods in a thread-local variable.

5.2.2 Local Rebinding Weaknesses

As it mimics the behavior of method overriding in a class hierarchy, local rebinding semantics seems to be quite natural. However, this property has drawbacks. First it cause encapsulation problems because the behavior of an object can be changed without it consent. Then accidental overrides can still occur, but in a contextual manner, making them even more difficult to track and understand for developers.

5.2.2.1 Potential Breach of Encapsulation

Beyond augmenting a class behavior, the local rebinding property can be used to change the behavior of a class without subclassing. With subclassing, the behavior of instances of the superclass is unchanged. Contrarily, with local rebinding, an extension method that overrides a regular method affects the extended class instances. Instances of the extended class behave differently depending on which extension methods are active in the current execution context (*i.e.* the current state of the call stack). These invasive behavior modifications by third parties can corrupt objects and raise a tension with object encapsulation.

5.2.2.2 Accidental Contextual Overrides

While the scope of extension methods is not global anymore, accidental overrides can still happen with local rebinding. Indeed, when considering all the active extension methods that are imported in the context of each method in a call chain, several extension methods may conflict with each other or with regular methods. A solution providing the local-rebinding property has to define a priority between these conflicting methods. The chosen one effectively overrides the others but only in certain dynamic conditions. We call these situations *contextual override* and we say that the method that has priority contextually overrides the others. When such overrides is not intended by the involved developer we call them *accidental contextual overrides*.

Because it is determined dynamically, the extension methods that are active at a given point is hard to anticipate, preventing local reasoning. For example, the control flow typically depends on the structure of the involved object graph. If objects in a graph are instances of classes located in different packages, a common situation, and if some of these packages import conflicting extension methods, the method lookup indirectly depends on the object structure and the consequent control flow.

Consider the example depicted in Figure 5.5. A `Collections` package defines common collections and an abstract class `Collection`. Two packages `ReadOnly` and `Record` each define a collection decorator.

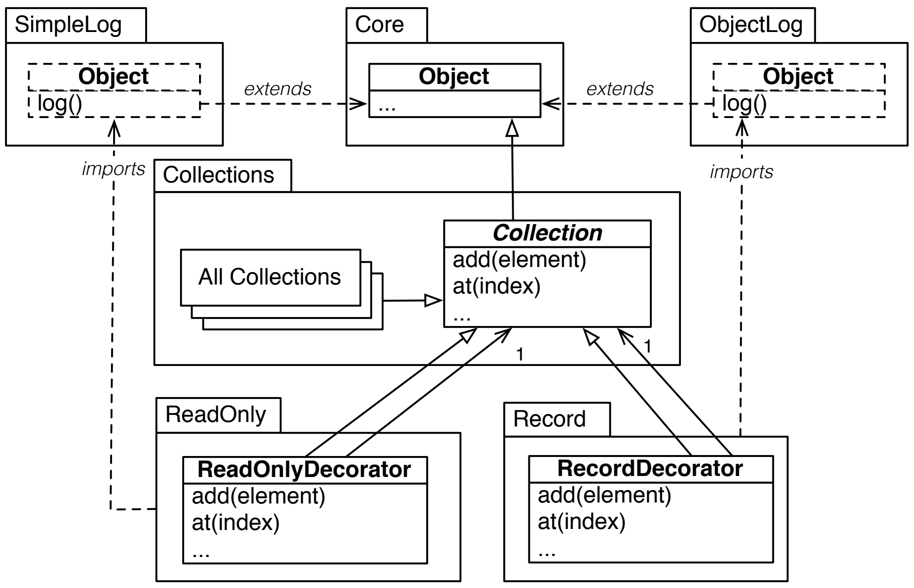


Figure 5.5: Decorating collections.

The read-only decorator disables all operations that mutate the decorated collection. When one of these operations is invoked, the read-only decorator logs the attempt using the logging facility of the `SimpleLog` package and throws an error. The record decorator just logs the operations done on the decorated collection using the logging facility of the `ObjectLog` package for latter analysis as shown below in pseudo-code.

```
ReadOnlyDecorator>>at: index
   ↑ decoree at: index

ReadOnlyDecorator>>add: element
   'Adding element is forbidden' log.
   ReadOnlyError signal

RecordDecorator>>at: index
   { 'accessing'. decoree. index } log.
   ↑ decoree at: index

RecordDecorator>>add: element
   { 'adding'. decoree. element } log.
   ↑ decoree add: index
```

If a client application uses both decorators together, one `log` method is likely to contextually override the other. This is the case when one decorator decorates the other. In this case the order of composition matters because it impacts the call stack and thus the extension methods that are active when looking up `log`.

```
list := #(1,2,3,4).
"case 1"
(ReadOnlyDecorator on: (RecordDecorator on: list)) at: 3.
"case 2"
(RecordDecorator on: (ReadOnlyDecorator on: list)) add: 5
```

In case 1, a read-only decorator decorates a record decorator that decorates a list. When sending the `at(3)` message to the read-only decorator, first its `at()` method transfers the request to the record decorator. The `at()` method of the record decorator then tries to log this operation. At this point two method activations are at the top of the call-stack: first an activation of the `at()` method of record decorator, then an activation of the `at()` method of the read-only decorator.

```
2. RecordDecorator at: 2
1. ReadOnlyDecorator at: 2
```

Since each package defining the `at()` method imports a different `log()` extension method, the lookup algorithm must decide which one to select. A similar situation occurs with case 2 with another call order.

```
2. ReadOnlyDecorator at: 5
1. RecordDecorator at: 5
```

This shows how the structure of an object graph, by influencing the control-flow, also influences the behavior of individual objects. We now study two strategies to select a method in case of ambiguities: *bottom-up local rebinding* and *top-down local rebinding*.

5.2.3 Bottom-up Local Rebinding

The first strategy gives precedence to extension methods imported by callers (*i.e.* appearing first in the call stack). We refer to this strategy as *bottom-up local rebinding*. This is the strategy of the Classbox and Method Shelters models.

In the context of Figure 5.5, this means that the `log()` method of the `SimpleLog` package is selected in Case 1 and the `log()` extension method of the `ObjectLog` package is selected in Case 2. This strategy implies that *client code may override other extension methods defined in any package*. As the developer of a package, your methods can be overridden by a package that is indirectly using yours. Consequently, this forces a developer to know the implementation of all the packages it uses (even indirectly) to prevent himself from creating accidental contextual overrides. This raises a tension with information hiding at the package-level and precludes local reasoning.

5.2.3.1 Classboxes

A classbox is a modular construct, akin to a package, that defines classes and class extensions. A classbox can define at most one class extension per imported class. This prevents useful ways to group related extension methods as we will see in Subsection 5.3.1. A classbox can import class extensions from other classboxes. Classboxes have been devised to facilitate handling of unanticipated changes. To handle unanticipated changes, a developer of a client classbox can push modifications to other classboxes. Used sparingly, classboxes allows developers to customize the implementation of external packages. However, if used extensively, accidental contextual overrides are likely to occur.

5.2.3.2 Method Shelters

The Method Shelters model [Akai 2012] builds upon the Classbox model and adds the ability to protect some extension methods from accidental contextual overriding. A method shelter consists of an *exposed chamber* and a *hidden chamber*. Each chamber declares imports toward other method shelters. Importing a method shelter brings the extension methods of its exposed chamber into the importing chamber. Only methods imported or declared in the exposed chamber of a shelter can be contextually overridden by other method shelters.

In Figure 5.6, two definitions of division (`/`) over integers coexist without accidental contextual override. The default `/` method of the `FixNum` class defines euclidian division. The `math` shelter redefines `/` as exact division: the method returns a rational number. The `average` shelter imports the `math` shelter in its hidden chamber. The `avg` method of `Array` uses the exact division of the `math` shelter to compute the average of an array of integers. Finally a `client` shelter imports the `average` shelter and computes the average of an integer array: the computation results in a rational number. The `client` shelter is oblivious of the fact that the `average` shelter uses the `math` shelter. From its point of view, `/` still refers to the standard euclidian division.

When only exported chambers are used, method shelters are similar to classboxes. This means that the same conflict problems happen. The developer of a method shelter has to choose which extension methods can be contextually overridden or not. To protect his code against unwanted overrides, the developer would likely adopt a defensive attitude and put all the extension methods he can in the hidden chamber. However, if the method shelter has to offer some facilities to client code via extension methods, they must be declared in its exposed chamber. In this case, these exposed methods are overridable. The method shelter model could be extended to support extension methods that are both importable by third parties and non-overridable.

5.2.4 Top-Down Local Rebinding

The second strategy gives priority to extension methods imported by callees. With this priority strategy, an extension method can be overridden in a called method. In the context of Figure 5.5, this means that the `log()` method of the `ObjectLog` package is selected in Case 1 and the `log()` extension method of the `SimpleLog` package is selected in Case 2. We refer to this strategy as *top-down local rebinding*. This is the strategy of *Categories* in Groovy.

5.2.4.1 Groovy Categories.

Groovy developers can define scoped extension methods in *categories*. A category defines a named extension that can be put into the scope of a block of code using the `use` keyword. When a `use` block is entered, the category is activated by pushing it onto a thread-local stack variable. This extension is popped from the thread-local stack of active extensions when the block is exited. Upon method lookup, a method redefined in a category takes priority over the original method in the extended class. In case of conflict between two extension methods in two categories, the method

```
shelter MathShelter do
  class Fixnum
    def /(x)
      Rational(self,x)
    end
  end
end

shelter ClientShelter do
  import AverageShelter
  def calc
    p([1,2,3,4,5,6,7,8,9,10].avg)          prints "(11/2)"
    p(55/10)                               prints "5"
  end
end

shelter AverageShelter do
  class Array
    def avg
      sum = self.inject(0){|r,i| r + i}
      sum / self.size
    end
  end
  hide
  import MathShelter
end

shelter_eval ClientShelter do
  calc
end
```

Figure 5.6: Example taken from [Akai 2012]. Method shelters provide the ability to control which method can be overridden: extension methods declared or imported after `hide` cannot be overridden by client shelters

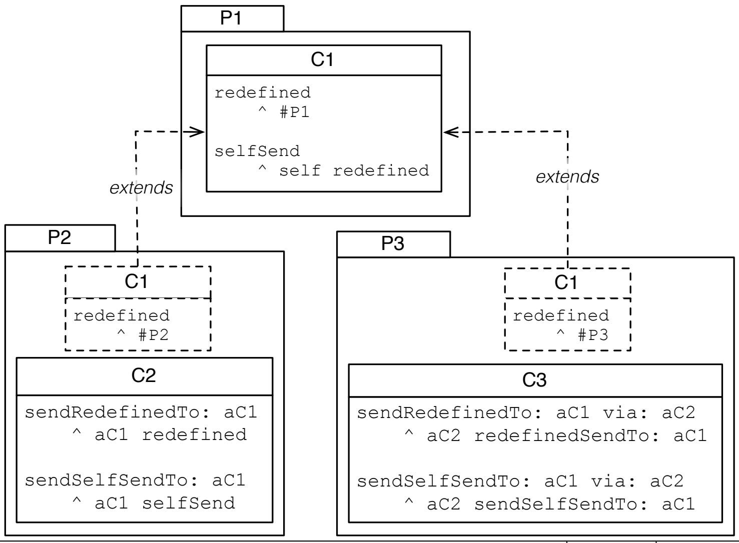
defined in the lastly-activated category (the one that is nearest to the top of the stack) is selected. Moreover, a `use` block can activate several categories. If there is conflicting methods in these categories the first definition hides the others.

All in all, we believe that local-rebinding brings more problems than solutions. Accidental contextual overriding asides, local rebinding violates encapsulation. Indeed, the developer of a class has no guaranty that this class instances will behave as he intend. Extension methods can override its own methods and can possibly lack private state or invalidate invariants. Lexical activation of extension does not have this problem. We believe that lexical activation has a simpler and more predictable behavior.

5.2.5 Lexical Extension Activation

The previous section presented different models providing local-rebinding and their issues. Here we present scoped extension methods with a lexical scope of activation. This kind of scoped extension methods is provided by *refinements* in Ruby. A similar solution is available in the *SmallScript* language under the name *selector namespaces*. Unfortunately, the lack of documentation for *selector namespaces* prevents us from analyzing its properties in details.

Since its first versions, Ruby supports extension methods, under the name of *open classes*. Ruby classes can indeed be *reopened* to add and change methods. Such modifications are globally visible. To tackle the problems of global visibility discussed in Subsection 5.1.2, Ruby 2.1 introduced scoped class extensions under the name of *refinements*. Only the modules or classes importing a refinement can call its extension methods. Unlike classboxes, refinements do not support local rebinding: all method calls are resolved lexically. If a class uses a refinement, this refinement is also active in the scope of the subclasses, even when the subclasses are defined in another package. This propagation of visibility provides some common facilities to subclasses, a feature that may be useful in frameworks where an abstract class of the framework is subclassed by users. Also, developers who subclass an external class should be aware of the refinements that are active in that class. Surprisingly, while the sequence of active refinements can be determined statically, the implementation of refinements does the resolution dynamically with



```
redefined
    ^ #P1

selfSend
    ^ self redefined
```

```
redefined
    ^ #P2
```

```
sendRedefinedTo: aC1
    ^ aC1 redefined

sendSelfSendTo: aC1
    ^ aC1 selfSend
```

```
redefined
    ^ #P3
```

```
sendRedefinedTo: aC1 via: aC2
    ^ aC2 redefinedSendTo: aC1

sendSelfSendTo: aC1 via: aC2
    ^ aC2 sendSelfSendTo: aC1
```

	LR↑	LR↓	Lexical
`aC2 sendRedefinedTo: aC1`	#P2	#P2	#P2
`aC2 sendSelfSendTo: aC1`	#P2	#P2	#P1
`aC3 sendRedefinedTo: aC1 via: aC2`	#P3	#P2	#P2
`aC3 sendSelfSendTo: aC1 via: aC2`	#P3	#P2	#P1

Figure 5.7: A summary of the different method lookup algorithms.

a dedicated and slower method lookup. This choice may be due to other implementation constraints.

Figure 5.7 presents a synthesis that shows the difference between bottom-up local rebinding (column "LR↑"), of top-down local rebinding (column "LR↓") and lexical extension activation (column "Lexical").

5.3 Analysis

The previous section presented different solutions to scope extension methods to their users. This section presents an analysis of the design space. We first discuss import granularities then we formalize the different method lookup algorithms.

5.3.1 Declaration of Dependencies

Once extension methods are local to their users it is mandatory for the users to declare which extension methods they bring into scope. Hence, all the existing solutions here solve the problem of hidden dependencies. These dependencies are usually declared with some form of import statement. Such statement establishes a dependency between a user (the *importer*) and a set of extension methods (the *importee*). The choice of the granularity at both ends of the dependency answers to the questions: "What is imported?" and "Where is it imported?".

Importee Granularity. Many different granularities can be considered for the importee side. Importing extension methods one by one is tedious: the solutions presented here offer means to group related extension methods together. One possible grouping is at the class-extension level (used by Classboxes for example): *i.e.* extension methods are grouped by the class they extend. This kind of grouping is simple but can not specify a set of related methods applying to different classes (such as the `asParser` methods presented in Subsection 5.1.1). Also, with classboxes and method shelters this class-centric grouping can not specify different sets of methods for the same class. Being able to make different groups for the same class can be useful: one group for a public API while another group is not meant to be exposed because it is implementation details. Another possible grouping is at the extension level (used by Method Shelters, Refinements and Categories): *i.e.* extension methods are grouped under a named extension and can affect different classes. This kind of grouping is more powerful: (1) an extension can specify a set of related methods in different classes (such as the `asParser` methods), (2) different extensions can specify different sets of methods for the same class, and (3) the previous class-based grouping can be realized with an extension whose methods all belong to the same class.

Importer Granularity. The granularity at the importer side determines what code is affected by the imported extension methods. With Classboxes for example, a class extension is imported and visible for all methods in the importing Classbox. With Groovy Categories, extension methods are activated during the execution of an importing block. With Ruby Refinements, imported extension methods are visible in the importing class and all its subclasses. We believe that the method, class and package-level are all valuable importers granularities and that a solution can support several or all of them.

5.3.2 Lookup Formalization

We presented several models of scoped extension methods in Section 5.2. To study the different design choices of each model, we give an abstract specification of a method lookup algorithm for scoped extension methods in the context of MOPLITE. To this end we extend MOPLITE syntax with extensions as follow (changes to original syntax are underlined).

$$\begin{array}{rcl} p \in \mathcal{P} & ::= & (\mathit{cls} \underline{\mid \ \mathit{extension}})^{*} \ \mathit{exp} \\ \mathit{cls} \in \mathcal{C} & ::= & \textbf{class} \ \mathit{id} \ \textbf{extends} \ \mathit{id} \ \underline{\textbf{uses} \ \mathit{id}^{*}} \ \{ \ \mathit{id}^{*} \ \mathit{meth}^{*} \ \} \\ \mathit{ext} \in \mathcal{E} & ::= & \textbf{extension} \ \mathit{id} \ \{ (\mathit{id} \ \mathit{meth}^{*})^{*} \ \} \end{array}$$

A program now consists of a list of class or extension definition, and is still followed by a main expression. A class definition now has a new **uses** clause that lists the extensions it imports. These imports could be declared at other granularities: for a single method, for a whole class hierarchy, for a whole package, *etc*. We choose the class level for simplicity. A new syntax is also introduced for extensions. An extension has a name and a list of extension methods for different classes.

The [*send*] and [*super-send*] rules of MOPLITE used an auxiliary function lookup to perform the method lookup algorithm. To model the context-sensitiveness of the lookup algorithms with local rebinding, the lookup function now takes a new parameter: a sequence of classes.

$$\mathrm{lookup}_p : \mathcal{I} \times \mathbb{N} \times \mathcal{C} \times \mathcal{C}^{*} \rightharpoonup \mathcal{M} \times \mathcal{C}$$

Each class correspond the current class of each stack frame present at lookup time from bottom (oldest frame) to top (newest frame). The abstract machine of MOPLITE consists of a stack and a heap. The stack frames are triplets: the first element is an evaluation context, the second is the current object (the binding of **self** in a frame) and the third, that is of interest here, is the current class. The current class is the one where the method that corresponds to the stack frame is defined. It was originally used for the lookup for super sends and now it is also used to retrieve the list of active extensions. In the following stack, the sequence of classes from which the list of active extensions is retrieved is: $(cls_1, cls_2, ..., cls_{n-1}, cls_n)$.

$$\left\lfloor \begin{array}{c} \langle ..., ..., cls_n \rangle \\ \langle ..., ..., cls_{n-1} \rangle \\ \vdots \\ \langle ..., ..., cls_2 \rangle \\ \langle ..., ..., cls_1 \rangle \end{array} \right\rfloor$$

To better distinguish between the different kinds of lookup algorithms, we divide the lookup in two steps. The first step, the function activeExts, determines the sequence of active extensions from the sequence of classes. The second step, the function select, selects a suitable method to be executed among the sequence of active extensions determined by thefeat first step.

$$\begin{aligned} &\mathrm{lookup}(cls, id, n, cls'^*) = \mathrm{select}(cls, id, n, \mathrm{activeExts}(cls'^*)) \\ &\quad \text{where} \\ &\quad \mathrm{activeExts}\colon \mathcal{C} \to \mathcal{E}^* \\ &\quad \mathrm{select}\colon \mathcal{C} \times \mathcal{I} \times \mathbb{N} \times \mathcal{E}^* \rightharpoonup \mathcal{M} \end{aligned}$$

We can now describe different versions of the activeExts and the selection functions separately. We call the different versions of activeExts *active extensions strategies* and the different versions of select *method selection strategies*.

5.3.3 Active Extensions Strategies

We now review the different active extension strategies. In the context of local rebinding, the lookup has to consider the chain of callers to find if one imports an extension with an overriding extension method. The extension activation is dynamically-scoped. This means that the lookup algorithm traverses the call stack or uses a thread-local variable to determine active extensions. The call-stack can be traversed bottom-up giving priority to callers imports, or top-down, giving priority to callees imports. Without local rebinding, the extension activation is said to be lexical. For each strategy, we consider that a global extension `global` that contains all regular methods is implicitly imported by default.

5.3.3.1 Bottom-up Local Rebinding.

We first consider the extension activation strategy of bottom-up local rebinding as exemplified by Classboxes and also by Method Shelters to a certain extent. The selection of active extensions for method shelters is more refined as it stops searching if one of the shelter is imported in a hidden chamber. Here is the definition of the $\mathrm{activeExts}_{\mathrm{lr}\uparrow}$ that computes the active extensions following the this strategy:

$$\mathrm{activeExts}_{\mathrm{lr}\uparrow}(< c_1, \ldots, c_n >) = \mathrm{imports}(c_1) \frown \ldots \frown \mathrm{imports}(c_n) \frown < \texttt{global} >$$

Here $\mathrm{imports}$ returns the sequence of extensions a class uses and "$\frown$" denotes concatenation of sequences. The function concatenates the imports of each class with `global` at the end. As a result of this bottom-up approach, extensions imported in the classes of the oldest stack frames come first. Precedence is thus given to extensions imported by calling code over extensions imported by called code.

5.3.3.2 Top-down Local Rebinding.

Now we consider top-down call-stack traversal as exemplified by Groovy categories. Here is the definition of the $\mathrm{activeExts}_{\mathrm{lr}\downarrow}$ that computes the active extensions following this strategy.

$$\mathrm{activeExts}_{\mathrm{lr}\uparrow}(< c_1, \ldots, c_n >) = \mathrm{imports}(c_n) \frown \ldots \frown \mathrm{imports}(c_1) \frown < \texttt{global} >$$

The function concatenates the imports of each class in reverse order with `global` at the end. As a result of this top-down approach, extensions imported in the classes of the newest stack frames come first. Precedence is given to extensions imported by called code over extensions imported by calling code.

5.3.3.3 Lexical Extension Activation.

We finally consider the lexical extension activation strategy as exemplified by Ruby refinements. The call-site determines active extensions alone. It means that the sequence of active extension is known statically. The active extensions are the ones imported by the calling method, that is the last element of the sequence cls^*.

$$\text{activeExts}_{\text{lex}}(< c_1, \ldots, c_n >) = \text{imports}(cls_1) \frown < \texttt{global} >$$

Choosing one of these three active extensions strategies (bottom-up local rebinding, top-down local rebinding, lexical) determines which method extensions are active during a message send. The next step of the lookup is to choose a method among these extensions.

5.3.4 Method Selection Strategy

Once the sequence of active extensions are determined according to one of the previous strategies, the second step is to select one method from all these extensions. One strategy is to lookup for a method in the first active extension throughout the hierarchy and then continue with following extensions. We refer to this strategy as *hierarchy-first method selection strategy*. Another solution is to lookup the method in the receiver class for each active extension in order and then continue to the superclass. We refer to this strategy as *extensions-first selection strategy*. The choice of the method selection strategy has a big impact for the accidental override situation depicted in Figure 5.3. Indeed, given a sequence of active extensions, these strategies determine whether in a hierarchy two extension methods with the same name from different extensions have an override relationship or not.

5.3.4.1 Extensions-First Method Selection Strategy

This first method selection strategy searches for a suitable method in each active extension before searching in the superclass of the receiver class. This is the strategy used by all solutions presented in Section 5.2.

$$\text{select}_{\text{ext}}(cls, id, n, ext^*) = \begin{cases} \text{lookupInClass}(cls, id, ext^*) & \text{if defined} \\ \text{select}_{\text{ext}}(\text{superclass}(cls), id, ext^*) & \text{if superclass}(cls) \text{ defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

The function $\text{select}_{\text{ext}}$ first looks if an extension in ext^* defines a method for the provided class and signature using the function lookupInClass. If no method is found (*i.e.* $\mathrm{lookupInClass}(cls, id, n, ext^*)$ is undefined), $\mathrm{select_{ext}}$ continues recursively with the superclass of cls if it exists. Otherwise, it is undefined if $\mathrm{superclass}(cls)$ is undefined. The function lookupInClass searches for the first suitable method defined for a given class in a given sequence of extensions. It is defined as follow:

$$\mathrm{lookupInClass}(cls, id, n, <>) \text{ is undefined}$$

$$\mathrm{lookupInClass}(cls, id, n, < e_1, \ldots, e_n >) = \\ \begin{cases} \mathrm{method}(cls, id, n, e_1) & \text{if defined} \\ \mathrm{lookupInClass}(cls, id, n, < e_2, \ldots, e_n >) & \text{otherwise} \end{cases}$$

With the extension-first method selection strategy, a method can be overridden in extensions with higher priority in that method class or any subclass.

5.3.4.2 Hierarchy-First Method Selection Strategy

The other solution to select a method given a sequence of extensions is to first lookup for the whole hierarchy of the receiver class in the context of the first extension and then consider the other extensions. As we will see later, this strategy limits accidental overrides compared to the extension-first strategy. It is defined as follow.

$$\mathrm{select_{hrc}}(cls, id, n, <>) \text{ is undefined}$$

$$\mathrm{select_{hrc}}(cls, id, n, < e_1, ..., e_n >) = \begin{cases} \mathrm{lookupInExtension}(cls, id, n, e_1) & \text{if it is defined} \\ \mathrm{select_{hrc}}(cls, id, n, < e_2, ..., e_n >) & \text{otherwise} \end{cases}$$

The function $\mathrm{select_{hrc}}$ first looks if the first extension defines a method in the class c or its superclasses from thanks to the function lookupInExtension. If no method is found, it continues recursively with the remaining scopes if there is some. The function lookupInExtension searches for the first method defined in the hierarchy of a given class in a given extension. It is defined as follow:

$$\text{lookupInExtension}(cls, id, n, ext) =$$
$$\begin{cases} \text{method}(cls, id, n, ext) & \text{if defined} \\ \text{lookupInExtension}(\text{superclass}(cls), id, n, ext) & \text{if superclass}(cls) \text{ defined} \\ \text{is undefined} & \text{otherwise} \end{cases}$$

With hierarchy-first method selection, a method can be overridden in extensions with higher priority in the whole hierarchy of that method class. Now we study which method selection strategies has the less risk of accidental overrides.

5.3.4.3 Estimation Accidental Overrides Risks

We now estimate and compare the risks of accidental overrides for the two method selection strategies. Let consider an arbitrary message $mess = (cls_{lookup}, id, n, ext^*)$ with the signature (id, n) with the sequence of active extensions ext^* and the lookup beginning the class cls_{rcv}. Let $meth = \text{lookup}_p(ext_i, id, n, cls_{def})$ be that method. This method is declared in the extension ext_i, the *i*-th extension of ext^*(possibly `global` if it is a regular method) for the class cls_{def} (*i.e.* cls_{lookup} or one of its superclasses). Now let consider the addition of an arbitrary method $new = \text{lookup}_p(ext_j, id, n, cls_{new})$ with the same signature (id, n). We want to model the set of method locations where this new method would cause an accidental override, *i.e.* the set of method locations that would cause $mess$ to dispatch new instead of $meth$. Since the method new has the same signature than $meth$, a method location only consists of a class and an extension. If new overrides $meth$ and are defined in the same extension ext_i, this override is intentional, so we only consider locations where $j \neq i$. We call this set of locations the *accidental overriding space* (AOS).

AOS of Extension-First Strategy. For extension-first method selection, new is an accidental override of $meth$ if: (1) new is defined for a subclass of cls_{def} in an extension ext_j in ext where $j \neq i$, or (2) new is defined for cls_{def} in an extension ext_j where $j < i$. If we note $\text{super}_p^{-1+}(cls)$ all the subclasses of a class cls (transitive closure of the inverse of super_p), we have:

$$\mathrm{AOS_{ext}}(mess) =$$
$$\{(cls_{new}, ext_j) \mid (cls_{new} \in \mathrm{super}_p^{-1+}(c_{def}) \wedge i \neq j) \vee (cls_{new} = cls_{def} \wedge i < j)\}$$

The size of $\mathrm{AOS_{ext}}(mess)$ is then given by:

$$|\mathrm{AOS_{ext}}(mess)| = |\mathrm{super}_p^{-1+}(cls_{def})| \times (|ext^*| - 1) + (i - 1)$$

AOS of Hierarchy-First Strategy. For hierarchy-first method selection, new accidentally overrides $meth$ if new is defined in any class in cls_{def} hierarchy in an extension e_j in ext where $j < i$. If we note $\mathrm{super}_p^{+}(c)$ all the superclasses of a class c, we have:

$$\mathrm{AOS_{hrc}}(mess) =$$
$$\{(cls_{new}, ext_j) \mid cls_{new} \in (\mathrm{super}_p^{-1+}(cls_{def}) \cup cls_{def} \cup \mathrm{super}_p^{+}(cls_{def})) \wedge i < j\}$$

The size of $\mathrm{AOS_{hrc}}(mess)$ is then given by:

$$|\mathrm{AOS_{hrc}}(mess)| = (|\mathrm{super}_p^{-1+}(cls_{def})| + |\mathrm{super}_p^{+}(cls_{def})| + 1) \times (i - 1)$$

Comparison of $|\mathrm{AOS_{ext}}|$ and $|\mathrm{AOS_{hrc}}|$. We can now compare the AOS of each method selection strategy. We ask ourself when the hierarchy-first strategy is better than the extension-first strategy *i.e.* when $|\mathrm{AOS_{hrc}}(mess)| \leq |\mathrm{AOS_{ext}}(mess)|$.

To do so we must have an idea of the average number of subclasses and superclasses a class has. We take these numbers from Pharo, where the average number of subclasses of a class is 8.82 and the average number of superclasses of a class is 3.83. With these numbers our inequality reduces to $1.43i - 0.43 \leq |ext^*|$. Remember that i ranges from 1 to $|ext^*|$.

We know for which values of i hierarchy-first strategy has less risk to cause accidental overrides than extension-first strategy. For $|ext^*|$ ranging from 1 to 10, the table below shows these values of i.

$\lvert ext^*\rvert$	1	2	3	4	5	6	7	8	9	10
$i \leq$	1	1	2	3	3	4	5	5	6	7

For example, when 5 extensions are active, the hierarchy-first strategy has less risk to cause an accidental override than extension-first strategy whenever $meth$ belongs to one of the first 3 active extensions. In this table we see that there are more values of i for which hierarchy-first strategy is better. This means that extension-first strategy is better only when $meth$ belongs to the last extensions, *i.e.* the ones that have smaller priority. In addition, hierarchy-first strategy has the advantage that an accidental method override happen only for extension with a higher priority. With extension-first strategy, accidental overrides can also happen for extension with a lower priority. These reasons shows that the hierarchy-first strategy is better to limit accidental overrides.

5.3.5 Synthesis

In this analysis we found that the design of scoped extension methods has a big impact on accidental overrides. It is thus important to make the good design decisions to not pose class encapsulation problems. Indeed we called these overrides "accidental", but they can also be malicious, *e.g.* voluntarily corrupting a class behavior to gain access to protected operations or break fundamental invariants. We determined that while local-rebinding improves code adaptability it causes too much encapsulation problems. On the other hand, lexical extension methods cannot modify the behavior of object in a contextual manner like local-rebinding does but are easier to reason about. Also we saw that the active extension strategy is not the only design choice. The method selection strategy also has impact on accidental overrides. We determined that the hierarchy-first strategy is better than the extension-first strategy. Finally the granularity of the import relationship has a consequence on expressivity and segregation of extension methods in meaningful groups. For the importee end, we saw that being able to define extension, named groups of extension methods is the best solution. For the other importer end, we think that multiple solutions are possible but a good solution would combine several granularity: package-level, class-level and method-level.

	Importee granularity	Importer granularity	Extension activation strategy	Method selection strategy
Classboxes	one class extension per class per package	package	bottom-up local rebinding	extensions-first
Method Shelters	one extension per package	package	controlled bottom-up local rebinding	extensions-first
Categories	many extensions per package	block of code	top-down local rebinding	extensions-first
Refinements	many extensions per package	class and its subclasses	lexical	extensions-first
Conch	many extensions per package	class, package or method	lexical	hierarchy-first

Figure 5.8: Comparison of the different approaches to scoped extension methods

5.3.5.1 Comparison of the Presented Solutions.

Figure 5.8 presents a comparison of the solution we have presented according the the criteria we have discussed previously: the importee granularity, the importer granularity, the extension activation strategy, and the method selection strategy. In addition, we add our solution (presented in Section 5.4) to the comparison.

5.4 Our Solution: Conch

In the previous sections we concluded that the combination of lexical activation and hierarchy-first strategies limits accidental overrides. Now we present Conch that follows the conclusions of this analysis. Conch is prototyped for Pharo. We believe our solution is close to the *selector namespace* mechanism of the *SmallScript* language, but we could find no code or article to precisely verify it. We know that selector namespace features lexical extension methods but we don't know which method selection strategy is used.

5.4.1 Context

Like in other languages of the Smalltalk family, using extension methods is idiomatic in Pharo. Pharo is thus a good candidate to study how much extension methods are used in a real system. In Pharo 3[2], 4.7% of all methods are extension methods, 16.7% of all classes and traits are extended, 48.1% of all packages define an extension method and 31.7% of all packages define a class or a trait that is extended by another package. These numbers demonstrate that extension methods are widely used in practice. Consequently, even though the number of accidental overrides is limited by coding conventions (like prefixing method names with the extending package name), it would be more preferable to prevent them by design.

The implementation given here is built with certain constraints in mind. Absence of static type information prevents the compiler to know the receiver class of a message send. Consequently, the message may dispatch to an extension method or to a regular method. Also, since extension methods are widely used, we want as little performance loss as possible. Moreover, the migration to the new model must be feasible without rewriting every existing package from scratch. Our implementation relies on the default method lookup of Pharo instead of changing it in the virtual machine. We briefly present this method lookup.

Smalltalk-80 Method-Lookup. The Pharo method lookup algorithm is inherited from Smalltalk-80. As a single-dispatch language, this lookup doesn't take the class of the arguments into account. The signature of a method only consists of a selector (the name of the method that also encode the number of parameter) instance of the class `Symbol`. Symbols are globally unique string: at most one symbol object exists for any sequence of characters.

The methods of a class are stored in a dictionary that maps method selectors to method objects that contain bytecode. When a message with a given selector is sent to an object, the lookup algorithm searches for the associated method as follows: first, the current class is set to the class of the receiver object. If the method dictionary of the current class contains a method for the given selector, the lookup stops and returns that method. Otherwise, the lookup pursues

[2]build #860

in the current class superclass. If no method is found in the hierarchy, the message `#messageNotUnderstood:` is sent to the receiver object with a reification of the original message as argument. Overriding `#messageNotUnderstood:` permits classes to answer to any message, a technique used to implement generic proxy objects as we saw in Chapter 3.

5.4.2 Model And Implementation

Conch is based on lexically scoped extensions with a hierarchy-first method selection strategy to minimise accidental overrides (see Figure 5.8). Packages gather classes and extensions and extensions can be imported at the package, class or method level.

An extension can have a parent extension. The child extension can declare methods that are polymorphic with the one declared in its parent: the messages sent by client code that uses the parent extension can dispatch to methods declared in the child extension. A child extension cannot override an extension method declared in its parent extension. This mechanism is useful when one wants to enhance an external extension with new methods. For example, if a developer creates a DSL of predicates to match objects upon various criteria, he can also issue a compatibility package with PetitParser. This compatibility package declares a child extension of the PetitParser extension (that declares the `asParser` extension methods). This child extension declares its own version of `asParser` in predicate classes to convert predicate objects into parsers. Client code that imports the PetitParser extension can then use the extension of the compatibility package without changes.

5.4.2.1 Selector Mangling Implementation.

The key point of this implementation is to distinguish a method name from its selector. The name is what the developer types in source code while the selector is the object used to look into class method dictionaries at run time. An extension is akin to a namespace: it maps a method name to a unique selector object. Instead of `Symbol` the selector of extension methods are instances of a new class `Selector`. Symbols are still used as selectors for regular methods.

Before an extension method m is compiled, its abstract syntax tree (AST) is transformed. The transformation changes the selector of the AST method node from a symbol to the associated selector object of m's extension e. If e has a parent extension, the selector associated with m's name in the parent extension is used. If no selector object is associated with m's name in e or its parent (*i.e.* m is the first method of its name), a new selector object is generated and installed in e.

When compiling any method (extension method or regular method), the selector of each AST message node is transformed as follows. Each extension that is visible from the method is queried for the selector associated with the message name in order of priority. Because several extensions can declare extension methods with the queried name, this yields an ordered list of selectors objects. If this list contains only one selector object, we replace the name of the message node (a symbol) with that selector object. If this list contains several selectors objects, we replace the name of the message node with a special selector object, instance of the class `DispatchSelector`. Dispatch selectors are used to resolve dynamically the ambiguities that exist at compile-time. The dispatch selector takes the list of selector objects to be looked-up at run time. Once a method is found, it then installs a new hidden entry in the method dictionary of the corresponding class that maps the dispatch selector to the found method to improve performances. We call these such new entry a *method alias*. Likewise, the next time the same dispatch selector is looked-up for the same class, the dynamic resolution is no longer necessary. To implement the hierarchy-first method selection strategy, each selector of this list must be looked-up one by one in the entire hierarchy of the receiver class before continuing with following extensions.

Self/Super Optimisation. In case the message node is a self send, we know that the method lookup will start in the class of the receiver. Consequently we consider only the selector objects of extensions that define an extension method for either the class of the receiver, one of its superclass, or one of its subclasses. In case the message send is a super send, we know that the method lookup will start in the superclass of m's class. Consequently we consider only the selector objects of extensions that define an extension method for either the superclass of the class of m, or one of its superclasses.

Backward Compatible Error Handling Hook. When the method-lookup fails the virtual machine sends the message `#doesNotUnderstand:` to the receiver with a reification of message as argument. We use a selector object (*i.e.* not a symbol) `#retry:` instead, in order to preserve the behavior of classes that override `#doesNotUnderstand:`. A method with this same selector object `#retry:` is defined in `ProtoObject`:

```
ProtoObject>>retry: aMessage
  ^ aMessage selector retryFor: self withMessage: aMessage
```

The `#retryFor:withMessage:` method of the class `Symbol` sends the normal `#doesNotUnderstand:` message:

```
Symbol>>retryFor: anObject withMessage: aMessage
  ^ anObject doesNotUnderstand: aMessage
```

This permits classes that redefine `#doesNotUnderstand:` to behave as intended. The method `#retryFor:withArguments:` of the class `Selector` is defined as follows:

```
Selector>>retryFor: anObject withMessage: aMessage
  (anObject class lookupSelector: self name) ifNotNil: [ :method |
    self installAliasesFor: method.
    ^ anObject perform: selector withArguments: aMessage arguments ].
  ^ anObject doesNotUnderstand: aMessage
```

This method first checks if the receiver class implements or inherits a regular method that has the same name. This is how we implement the implicitly imported global extension that contains all regular methods. If a method is found, method aliases are installed and the next lookups of this selector in that class hierarchy will succeed directly and be as fast as normal message send.

For dispatch selectors, the situation is similar. At run time, when a message with a dispatch selector is sent for the first time, the lookup will fail because no method has this dispatch selector as selector. The method `#retryFor:withMessage:` of the class `DispatchSelector` is defined as follow:

```
DispatchSelector>>retryFor: anObject withMessage: aMessage
  self selectors do: [ :each |
    (anObject class lookupSelector: each) ifNotNil: [ :method |
      self installAliasesFor: method.
      ^ anObject perform: selector withArguments: aMessage arguments ] ].
  ^ super retryFor: anObject withMessage: aMessage
```

This method searches the first selector understood by the receiver's class and sends it to the receiver with the message arguments using the reflective send method `#perform:withArguments:` after installing the corresponding method aliases. If none of the selector is understood by the receiver's class we fallback with `Selector`'s behavior.

To summarize, Conch is an implementation of lexically-scoped extensions methods for Pharo that requires no modification of the virtual machine. It solves the problem of ambiguous call-sites thanks to dispatch selectors. Thanks to method aliases, the amortized performance cost is zero.

5.5 Related Work

We now compare our work with related work: Bergel's module taxonomy, scoped extension methods in statically-typed languages and the `new` modifier of C#.

5.5.1 Module Taxonomy

Bergel *et al.* [Bergel 2005a] present a taxonomy of module systems using a module calculus consisting of a small set of operators over environments and modules. Using these operators, they specify a set of module combinators that capture the semantics of Java packages, C# namespaces, Ruby modules, selector namespaces, gbeta classes, classboxes, MZScheme units, and MixJuice modules. Even if the paper covers Classboxes, their semantics does not capture the local rebinding lookup stack traversal.

5.5.2 Solutions for Scoped Extension Methods in Statically-Typed Languages.

In this chapter we have limited our research to solutions in dynamically-typed languages. There exists other solutions in the context of statically-typed languages. These solutions are not portable to dynamically-typed languages because they rely on static type information. In C#, extension methods are essentially syntactic sugar over static methods whose first argument type is the extended type. Scala supports a mechanism similar to extension methods with *implicit classes*.

MultiJava [Clifton 2000] is a Java extension that support open classes and multiple dispatch. The open-class solution proposed by MultiJava is close to our proposition for dynamic languages. MultiJava protects from accidental overrides: a method m overrides an extension method m' only if m' is imported in the file that declares m.

Expanders [Warth 2006] are another language construct that support scoped extension methods in the context of *eJava*, a Java extension. An expander is a class extension that can be brought into the lexical scope of a compilation unit. It allows classes to be updated with new methods, fields and interfaces. An expander can override the extension methods defined in another expander. This solution enables intended overrides while preventing accidental ones. An extension method cannot override a regular method. By importing expanders, client code adapts some classes to its particular need.

Module Refinement. PRM is a statically-typed language supporting class refinement [Ducournau 2007]. In PRM, a module is a class hierarchy, i.e. a set of classes ordered by specialisation. It is a reuse unit which can be compiled separately and then linked to other modules to produce a final executable. A module depends on a set of other modules (called supermodules) and can refine classes imported from them. A class refinement can be one of the four atomic mechanisms: (1) adding a property, i.e., the definition of a newly introduced method or attribute; (2) redefining a property; (3) adding a superclass; (4) generalizing a property, i.e., defining a property in superclasses of the class which introduced it in the supermodules.

5.5.3 C# `new` Method Modifier.

C# is the one of the rare languages that offers a way to control accidental overrides. C# allows the programmer to qualify a method with the keyword `new` to declare that while the newly defined method has the same name as the one in a superclass, it is not an override, *i.e.* it is used for a different concept. As such all calls in the superclass that would invoke a method with the same name will not consider that new method.

5.6 Conclusion

Globally-visible extension methods can lead to conflicts: accidental overrides and overwrites. These conflicts pose class encapsulation problems that can lead to subtle bugs or be exploited by malicious parties. In this chapter we propose studied various solutions that propose to scope extension methods in dynamically-typed languages: Classboxes, Ruby Refinements, Method Shelters, and Groovy Categories. We saw that the semantics of scoped extension methods has a big impact on accidental overrides, and concluded that the combination of lexical extension methods with the hierarchy-first method selection strategy gives the best results. We proposed Conch , a solution for scoped extension methods that follows these conclusions, and described its implementation for Pharo that incurs little performance overhead by leveraging a traditional class-based method lookup algorithm.

5.6.1 Future Work

Our implementation is based on first-class and unforgeable selectors. We believe this kind of selector deserve further investigation.

When used only at the language level, like in our implementation, these selectors can also be used to control the visibility of methods in a number of different ways. Hence, these selectors can serve as a foundation to unify method visibility with scoped extension methods. Scoping extension methods lexically is indeed an advanced form of method visibility.

But these first-class selectors could also be used at the application level. Indeed, the difference between these first class selectors and name mangling is that we use specific objects to represent scoped selectors. Unlike a mangled name, these objects are unforgeable and can be used as a form of right amplification. A first-class selector is a capability to send messages to instances of specific classes. For example, a client object with a reference to a subject object cannot send a specific protected message to that object unless it also has a reference to the corresponding selector object. With a reference to that selector object in hand, the client can reflectively send the protected message. In this context, selector objects are a way to assign and delegate permissions to send messages to instances of specific classes, thereby blurring the boundary between static and dynamic permission assignments.

CHAPTER 6

Conclusion

Contents

This last chapter summarizes the contributions made by this dissertation and point to directions for future works.

6.1 Contributions

This thesis studied the tension between adaptability and encapsulation in the context of reflective object-oriented languages. After motivating the need for software that is both adaptable and encapsulated, we presented the tension between these two goals under two perspectives: a behavioral perspective with the tension between behavioral reflection and object encapsulation and a structural perspective with the tension between extension methods and class encapsulation.

6.1.1 Reflection

We presented reflective concepts and reflective architecture in object-oriented languages. We presented and formalized the reflective architecture that is used in this thesis that takes the form of an object-centric MOP. Such architecture enables behavioral variations on a per-object basis. Behavioral variations can focus on monitoring and adaptive aspects, thereby improving adaptability but also on encapsulation and security aspects, thereby improving security. We gave a formalization of a simple object-centric MOP used to formalize the mechanisms presented in this thesis.

We studied proxies in details and their realization in an object-centric MOP. We saw the effect of delegation is important for composition of behavioral variations and for the `self`-problem. Proxies can form chains to compose their behavioral variations: different parties can add their own behavioral variation without being aware of other behavioral variations already active for the same target object. We can for instance trace and profile an execution by using tracing proxies and profiling proxies. Adapting objects during an execution will not affect other objects in the system (partial reflection [Tanter 2003]). Also, objects are wrapped selectively and a behavioral variation is enabled only for the proxy. A variation is enabled for clients who possess a reference to the proxy while other clients may have a reference to the target or to another proxy implementing another behavioral variation. It is up to the creator of the proxy to decide whether to pass the proxy or the target. Additionally, with specific wrapping rules it is possible to propagate a behavioral variation to isolate an object graph with membranes or a portion of execution with control flow propagation. Since the propagation is written reflectively, it can be customized to achieve various forms of scoping.

We explored the problem of encapsulation violation caused by reflective operations and its implications on the *OCap* model. The tension between the need for object encapsulation on the one hand and the need of reflection on the other hand led us to the conclusion that we need a way to track when breaking into an encapsulation boundary is legitimate. To this end we have explored the concept of dynamic object ownership that has been originally used to control object aliasing. Instead of object aliasing, we showed how this notion of object ownership can be used to design an access policy to reflective operations. Thanks to this access control policy, owners of an object can perform any reflective operations on that object. An object that does not own a target object has only access to limited reflective abilities. This simple policy reconciles reflection and security in the context of multiple interacting software components. We implemented this policy in the context of a simple object-centric MOP. In this context, we showed that this policy can ensure that reflectively implemented domain-level access-control policies cannot be bypassed using reflection.

6.1.2 Extension Method

Finally we studied different extension method mechanisms and the implication of their semantics on class encapsulation. We proposed a framework to formally study various solutions that proposed to scope extension methods in dynamically-typed languages: Classboxes, Ruby Refinements, Method Shelters, and Groovy Categories. We saw that the semantics of scoped extension methods has a big impact on accidental overrides, and concluded that the combination of lexical extension methods with the hierarchy-first method selection strategy gives the best results. We proposed Conch, a solution for scoped extension methods that follows these conclusions, and described its implementation for Pharo that incurs little performance overhead by leveraging the traditional lookup algorithm of single-dispatch class-based languages.

6.2 Future Works

We gave tracks to future works throughout the dissertation that we summarize here.

6.2.1 Ownership for Alias Control

Originally, object ownership has been devised to control the effect of object aliasing. Later, many different ownership systems have been used for many other applications [Clarke 2013]: concurrency control, memory management, security, etc.

Since our access control policy relies on ownership information, it would be interesting to leverage this information with other applications. The first application that comes to mind is object alias control since it is the original application of object ownership. So an interesting question is to know if an effective alias control discipline can be compatible with our access control policy.

This seems difficult because one principle of the original ownership-based alias control (known as the *owner-as-dominator* discipline) is *"no representation exposure"*. This means that external objects can not reference the representation objects of an aggregate. The transitivity of our ownership relation contradicts with this principle. An alias control discipline has to rely on a transitive ownership relation to be compatible with our access control policy to metaobjects.

6.2.2 Unification of Method Visibility with Extension Methods

The distinction between a method name and a method selector used in our Conch implementation can also be used to control the visibility of methods in a number of different ways and hence can serve as a foundation to unify method visibility with scoped extension methods. The visibility modifiers of a language are fixed and doesn't necessarily meet developers needs. Also visibility modifiers often come with assumptions about the relationship with between visibility and overriding. For example, in Java if a method in a superclass is visible then a method with the same signature is necessarily and override. An infrastructure that allow developers to defines method visibilities that meets the specific modularity and security requirements of the application under development, such as *encapsulation policies* [Schärli 2004], is really valuable. If the same infrastructure is used for extension methods, this would offer an expressive way to deal with encapsulation.

6.2.3 First-class Selectors for Assignment and Delegation of Permissions

The difference between our implementation and name mangling is that we used specific objects to represent scoped selectors. Unlike a mangled name, these objects are unforgeable. They can thus be used as a form of right amplification. For example, a client object with a reference to a subject object cannot send a specific protected message to that object unless it also has a reference to the corresponding selector object. With a reference to that selector object, the client can reflectively send the protected message to that subject object. Selector objects can then be used as a way to assign and delegate permissions to send messages to instances of specific classes, thereby blurring the boundary between static and dynamic permission assignments

Bibliography

[Akai 2012] Shumpei Akai and Shigeru Chiba. *Method Shelters: Avoiding Conflicts Among Class Extensions Caused by Local Rebinding*. In Proceedings of the 11th International Conference on Aspect-Oriented Software Development (AOSD'12), 2012.

[Aksit 1993] Mehmet Aksit, Ken Wakita, Jan Bosch, Lodewijk Bergmans and Akinori Yonezawa. *Abstracting Object Interactions Using Composition Filters*. In Proceedings of the ECOOP'93 Workshop on Object-Based Distributed Programming, 1993.

[Ancona 1999] Massimo Ancona, Walter Cazzola and Eduardo B Fernandez. *Reflective Authorization Systems: Possibilities, Benefits, and Drawbacks*. In Secure Internet Programming – Security Issues for Mobile and Distributed Objects. Springer, 1999.

[Apel 2008] Sven Apel, Thomas Leich and Gunter Saake. *Aspectual Feature Modules*. Transactions on Software Engineering, vol. 34, 2008.

[Aracic 2006] Ivica Aracic, Vaidas Gasiunas, Mira Mezini and Klaus Ostermann. *An Overview of CaesarJ*. Transactions on Aspect-Oriented Software Development (TAOSD), vol. 1, 2006.

[Arnaud 2010] Jean-Baptiste Arnaud, Marcus Denker, Stéphane Ducasse, Damien Pollet, Alexandre Bergel and Mathieu Suen. *Read-Only Execution for Dynamic Languages*. In Proceedings of the 48th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Europe'10), 2010.

[Arnaud 2013a] Jean-Baptiste Arnaud. *Towards First Class References as a Security Infrastructure in Dynamically-Typed Languages*. PhD thesis, Université de Lille, 2013.

[Arnaud 2013b] Jean-Baptiste Arnaud, Stéphane Ducasse and Marcus Denker. *Behavior-Propagating First Class References For Dynamically-Typed Languages*. Science of Computer Programming, vol. 98, 2013.

[Arnaud 2015] Jean-Baptiste Arnaud, Stéphane Ducasse, Marcus Denker and Camille Teruel. *Handles: Behavior-Propagating First-Class References for Dynamically-Typed Languages*. Science of Computer Programming, vol. 98, 2015.

[Bergel 2003] Alexandre Bergel, Stéphane Ducasse and Roel Wuyts. *Classboxes: A Minimal Module Model Supporting Local Rebinding*. In Proceedings of the 4th Joint Modular Languages Conference (JMLC'03), 2003.

[Bergel 2005a] Alexandre Bergel, Stéphane Ducasse and Oscar Nierstrasz. *Analyzing Module Diversity*. Journal of Universal Computer Science, vol. 11, 2005.

[Bergel 2005b] Alexandre Bergel, Stéphane Ducasse and Oscar Nierstrasz. *Classbox/J: Controlling the Scope of Change in Java*. In Proceedings of the 20th International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'05), 2005.

[Bettini 2007] Lorenzo Bettini, Sara Capecchi and Elena Giachino. *Featherweight Wrap Java*. In Proceedings of the 22nd Symposium on Applied Computing (SAC'07), 2007.

[Bracha 2004] Gilad Bracha and David Ungar. *Mirrors: Design Principles for Meta-level Facilities of Object-oriented Programming Languages*. In Proceedings of the 19th International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'04), 2004.

[Büchi 2000] Martin Büchi and Wolfgang Weck. *Generic Wrappers*. In Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP'00), 2000.

[Caromel 2001a] Denis Caromel, Fabrice Huet and Julien Vayssière. *A Simple Security-Aware MOP for Java*. In Proceedings of the 3rd International Conference on Reflection (REFLECTION'01), 2001.

[Caromel 2001b] Denis Caromel and Julien Vayssière. *Reflections on MOPs, Components, and Java Security*. In Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01), 2001.

[Clarke 1998] David G. Clarke, John M. Potter and James Noble. *Ownership Types for Flexible Alias Protection*. In Proceedings of the 13th International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'98), 1998.

[Clarke 2001] David G. Clarke, James Noble and John M. Potter. *Simple Ownership Types for Object Containment*. In Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01), 2001.

[Clarke 2013] Dave Clarke, Johan Östlund, Ilya Sergey and Tobias Wrigstad. *Ownership Types: A Survey*. In Aliasing in Object-Oriented Programming – Types, Analysis and Verification. Springer, 2013.

[Clifton 2000] Curtis Clifton, Gary T. Leavens, Craig Chambers and Todd Millstein. *MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java*. In Proceedings of the 15th International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'00), pages 130–145, 2000.

[De Meuter 1998] Wolfgang De Meuter. *Agora: The Story of the Simplest MOP in the World — or — The Scheme of Object-Orientation*. In Prototype-based Programming – Concepts, languages and applications. Springer, 1998.

[De Meuter 2005] Wolfgang De Meuter, Eric Tanter, Stijn Mostinckx, Tom Van Cutsem and Jessie Dedecker. *Flexible Object Encapsulation for Ambient-Oriented Programming*. In Proceedings of the 1st Dynamic Languages Symposium (DLS'05), 2005.

[Denker 2008] Marcus Denker, Mathieu Suen and Stéphane Ducasse. *The Meta in Meta-Object Architectures*. In Proceedings of the 46th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Europe'08), 2008.

[Ducasse 1999] Stéphane Ducasse. *Evaluating Message Passing Control Techniques in Smalltalk*. Journal of Object-Oriented Programming (JOOP), vol. 12, 1999.

[Ducournau 2007] Roland Ducournau, Floréal Morandat and Jean Privat. *Modules and Class Refinement: A Meta-Modeling Approach to Object-Oriented Programming*. Rapport technique, LIRMM, Université Montpellier II, 2007.

[Eugster 2006] Patrick Eugster. *Uniform Proxies for Java*. In Proceedings of the 21th International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'06), 2006.

[Felleisen 1992] Matthias Felleisen and Robert Hieb. *The Revised Report on the Syntactic Theories of Sequential Control and State*. Theoretical Computer Science, vol. 103, 1992.

[Ferber 1989] Jacques Ferber. *Computational Reflection in Class-Based Object-Oriented Languages*. In Proceedings of the 4th International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'89), 1989.

[Friedman 1984] Daniel P. Friedman and Mitchell Wand. *Reification: Reflection without Metaphysics*. In Proceedings of the 3rd Symposium on LISP and Functional Programming (LFP'84), 1984.

[Gabriel 2006] Richard P Gabriel and Ron Goldman. *Conscientious Software*. In Acm Sigplan Notices, volume 41, 2006.

[Gamma 1995] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. Design patterns: Elements of reusable object-oriented software. Pearson Education, 1995.

[Gordon 2007] Donald Gordon and James Noble. *Dynamic Ownership in a Dynamic Language*. In Proceedings of the 3rd Dynamic Languages Symposium (DLS'07), 2007.

[Harrison 1993] William Harrison and Harold Ossher. *Subject-Oriented Programming (A Critique of Pure Objects)*. In Proceedings of the 8th International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'93), 1993.

[Hayes 1997] Barry Hayes. *Ephemerons: A New Finalization Mechanism*. In Proceedings of the 12th International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'97), 1997.

[Hirschfeld 2008] Robert Hirschfeld, Pascal Costanza and Oscar Nierstrasz. *Context-Oriented Programming*. Journal of Object Technology, vol. 7, 2008.

[Hölzle 1995] Urs Hölzle. *Adaptive optimization for SELF: reconciling high performance with exploratory programming*. PhD thesis, Stanford University, 1995.

[Kiczales 1991] Gregor Kiczales, Jim des Rivières and Daniel G. Bobrow. The art of the metaobject protocol. MIT Press, 1991.

[Kniesel 1999] Günter Kniesel. *Type-Safe Delegation for Run-Time Component Adaptation*. In Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP'99), 1999.

[Kristensen 1996] Bent Bruun Kristensen. *Object-Oriented Modeling with Roles*. In Proceedings of the 2nd International Conference on Object Oriented Information Systems (OOIS'96), 1996.

[Lieberman 1986] Henry Lieberman. *Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems*. In Proceedings of the 1st International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'86), November 1986.

[Maes 1987a] Pattie Maes. *Computational Reflection*. PhD thesis, Vrije Universiteit Brussel, 1987.

[Maes 1987b] Pattie Maes. *Concepts and Experiments in Computational Reflection*. In Proceedings of the 2nd International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'87), 1987.

[Marr 2015] Stefan Marr, Chris Seaton and Stéphane Ducasse. *Zero-Overhead Metaprogramming*. In Proceedings of the 36th Annual Conference on Programming Language Design and Implementation (PLDI'15), 2015.

[Martin 2003] Robert Cecil Martin. Agile software development: Principles, patterns, and practices. Prentice Hall PTR, 2003.

[Martinez Peck 2014] Mariano Martinez Peck, Noury Bouraqadi, Luc Fabresse, Marcus Denker and Camille Teruel. *Ghost: A Uniform and General-Purpose Proxy Implementation*. Journal of Object Technology, vol. 98, 2014.

[Masuhara 2003] H. Masuhara, G. Kiczales and C. Dutchyn. *A Compilation and Optimization Model for Aspect-Oriented Programs*. In Proceedings of the 12th International Conference on Compiler Construction (CC'03), 2003.

[McAffer 1995] Jeff McAffer. *Meta-level Programming with CodA*. In Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95), 1995.

[McCullough 1987] Paul L. McCullough. *Transparent Forwarding: First Steps*. In Proceedings of the 2nd International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'87), 1987.

[Mettler 2010] Adrian Mettler, David Wagner and Tyler Close. *Joe-E: A Security-Oriented Subset of Java*. In Proceedings of the 17th Annual Network & Distributed System Security Conference (NDSS'10), 2010.

[Miller 2003] Mark S. Miller and Jonathan S. Shapiro. *Paradigm Regained: Abstraction Mechanisms for Access Control.* In Proceedings of the 8th Asian Computing Science Conference (ASIAN'03), 2003.

[Miller 2006] Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control.* PhD thesis, Johns Hopkins University, 2006.

[Mittal 1986] Sanja Mittal, Daniel G. Bobrow and Kenneth M. Kahn. *Virtual Copies – At the Boundary Between Classes and Instances.* In Proceedings of the 1st International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'86), 1986.

[Mostinckx 2007] Stijn Mostinckx, Tom Van Cutsem, Stijn Timbermont and Eric Tanter. *Mirages: Behavioral Intercession in a Mirror-based Architecture.* In Proceedings of the 3rd Dynamic Languages Symposium (DLS'07), 2007.

[Mostinckx 2009] Stijn Mostinckx, Tom Van Cutsem, Stijn Timbermont, Elisa Gonzalez Boix, Éric Tanter and Wolfgang De Meuter. *Mirror-Based Reflection in AmbientTalk.* Software: Practice and Experience, vol. 39, 2009.

[Noble 1998] James Noble, Jan Vitek and John Potter. *Flexible Alias Protection.* In Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP'98), 1998.

[Noble 1999] James Noble, David Clarke and John Potter. *Object Ownership for Dynamic Alias Protection.* In Proceedings of the 32th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific'99), 1999.

[Ostermann 2002] Klaus Ostermann. *Dynamically composable collaborations with delegation layers.* In Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP'02), 2002.

[Parnas 1972] David L. Parnas. *On the Criteria To Be Used in Decomposing Systems into Modules.* Communications of the ACM, vol. 15, 1972.

[Pascoe 1986] Geoffrey A. Pascoe. *Encapsulators: A New Software Paradigm in Smalltalk-80*. In Proceedings of the 1st International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'86), 1986.

[Renaud 2001] K Renaud. *JavaCloak: Considering the Limitations of Proxies for Runtime Specialisation*. In Proceedings of the 6th Annual Conference of the South African Institute of Computer Scientists and Information Technologists (SAICSIT'01), 2001.

[Renggli 2010] Lukas Renggli, Stéphane Ducasse, Tudor Gîrba and Oscar Nierstrasz. *Practical Dynamic Grammars for Dynamic Languages*. In Proceedings of the 4th Workshop on Dynamic Languages and Applications (DYLA'10), June 2010.

[Ressia 2012] Jorge Ressia. *Object-Centric Reflection*. PhD thesis, University of Bern, 2012.

[Ressia 2014] Jorge Ressia, Tudor Gîrba, Oscar Nierstrasz, Fabrizio Perin and Lukas Renggli. *Talents: an Environment for Dynamically Composing Units of Reuse*. Software: Practice and Experience, vol. 44, 2014.

[Riechmann 1997] Thomas Riechmann and Franz J Hauck. *Meta Objects for Access Control: Extending Capability-Based Security*. In Proceedings of the 6th New Security Paradigms Workshop (NSPW'97), 1997.

[Riechmann 1998] Thomas Riechmann and Jürgen Kleinöder. *Meta Objects for Access Control: Role-Based Principals*. In Proceedings of the 3rd Australasian Conference on Information Security and Privacy (ACISP'98), 1998.

[Schärli 2003] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz and Andrew P. Black. *Traits: Composable Units of Behavior*. In Proceedings of the 17th European Conference on Object-Oriented Programming (ECOOP'03), 2003.

[Schärli 2004] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz and Roel Wuyts. *Composable Encapsulation Policies*. In Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP'04), 2004.

[Smith 1984] Brian Cantwell Smith. *Reflection and Semantics in Lisp*. In Proceedings of the 11th Symposium on Principles of Programming Languages (POPL'84), 1984.

[Smith 1996] Randall B. Smith and Dave Ungar. *A Simple and Unifying Approach to Subjective Objects*. Theory and Practice of Object Systems (TAPOS) – Special Issue on Subjectivity in Object-Oriented Systems, vol. 2, 1996.

[Strickland 2012] T.S. Strickland, S. Tobin-Hochstadt, R.B. Findler and M. Flatt. *Chaperones and Impersonators: Run-Time Support for Reasonable Interposition*. In Proceedings of the 27th International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'12), 2012.

[Tanter 2003] Éric Tanter, Jacques Noyé, Denis Caromel and Pierre Cointe. *Partial Behavioral Reflection: Spatial and Temporal Selection of Reification*. In Proceedings of the 18th International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'03), 2003.

[Tanter 2008] Éric Tanter. *Expressive Scoping of Dynamically-Deployed Aspects*. In Proceedings of the 7th International Conference on Aspect-Oriented Software Development (AOSD'08), 2008.

[Tanter 2009] Éric Tanter, Johan Fabry, Rémi Douence, Jacques Noyé and Mario Südholt. *Expressive Scoping of Distributed Aspects*. In Proceedings of the 8th International Conference on Aspect-Oriented Software Development (AOSD'09), 2009.

[Teruel 2015] Camille Teruel, Erwann Wernli, Stéphane Ducasse and Oscar Nierstrasz. *Propagation of Behavioral Variations with Delegation Proxies*. Transactions on Aspect-Oriented Software Development (TAOSD), vol. 12, 2015.

[Van Cutsem 2010] Tom Van Cutsem and Mark S. Miller. *Proxies: Design Principles for Robust Object-oriented Intercession APIs*. In Proceedings of the 6th Dynamic Languages Symposium (DLS'10), 2010.

[Van Cutsem 2013] Tom Van Cutsem and Mark S. Miller. *Trustworthy Proxies – Virtualizing Objects with Invariants*. In Proceedings of the 22th European Conference on Object-Oriented Programming (ECOOP'13), 2013.

[Viega 2000] John Viega, Paul Reynolds and Reimer Behrends. *Automating Delegation in Class-Based Languages*. In Proceedings of the 34th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS USA'00), 2000.

[von Löwis 2007] Martin von Löwis, Marcus Denker and Oscar Nierstrasz. *Context-Oriented Programming: Beyond Layers*. In Proceedings of the 15th International Conference on Dynamic Languages (ICDL'07), 2007.

[Warth 2006] Alessandro Warth, Milan Stanojević and Todd Millstein. *Statically Scoped Object Adaptation with Expanders*. In Proceedings of the 21th International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'06), 2006.

[Warth 2011] Alessandro Warth, Yoshiki Ohshima, Ted Kaehler and Alan Kay. *Worlds: Controlling the Scope of Side Effects*. In Proceedings of the 25th European Conference on Object-Oriented Programming (ECOOP'11). LNCS, 2011.

[Welch 1999] Ian Welch and Robert Stroud. *Dalang – A Reflective Java Extension*. In Proceedings of the OOPSLA'99 Workshop on Reflective Programming in C++ and Java., 1999.

[Wernli 2014] Erwann Wernli, Oscar Nierstrasz, Camille Teruel and Stéphane Ducasse. *Delegation Proxies: The Power of Propagation*. In Proceedings of the 13th International Conference on Modularity (MODULARITY'14), 2014.

[Zimmermann 1996] Chris Zimmermann. Advances in object-oriented metalevel architectures and reflection. CRC Press, 1996.