Run-time Variability with First-class Contexts

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Erwann Wernli
von Zürich

Leiter der Arbeit:
Prof. Dr. O. Nierstrasz
Institut für Informatik und angewandte Mathematik

Run-time Variability with First-class Contexts

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Erwann Wernli
von Zürich

Leiter der Arbeit:
Prof. Dr. O. Nierstrasz
Institut für Informatik und angewandte Mathematik

Von der Philosophisch-naturwissenschaftlichen Fakultät angenommen.

Bern, 31.10.2013

Der Dekan:
Prof. Dr. S. Decurtins

This dissertation can be downloaded from `scg.unibe.ch`.

Acknowledgments

I warmly thank the following persons who directly, or indirectly, contributed to this work:

First, I express my gratitude to Oscar for his support throughout the years. His expertise in articulating ideas has been invaluable.

I am very thankful to Manuel for his interest in my research. This was motivational. I also thank him for accepting to be the external reviewer.

I thank my friends Niko, for his cheerful enthusiasm and deep love of puzzles, Fabrizio, for bringing fun to any discussion, Jorge, for many useful reflections, and Mircea, for many whimsical discussions about research and life.

I am grateful to Toon, for his compelling low-tech known-how, Lukas, for his impressive involvement in the community, Doru, for his tips about research and presenting, Adrian K., for his thought-provoking nature, and Adrian L., for early help with my research.

I am indebted to David and Pascal for their contributions to the technical implementations. I am also grateful to Camille and Stéphane for discussions about proxies.

Thanks to Jan, Andrea, Boris, Andrei, Haidar, and Nevena, for reading drafts of my work and bringing fresh energy to the group during the last stage of my thesis.

I am grateful to Iris for her flawless organization and administrative support.

Special thanks to Slavisa, for encouraging me to take this academic detour, Maxime, for early guidance during the application process, and Gwenaëlle, Heinz, Jacqueline, for cheering me up when it was needed.

Above all, I want to thank Lilith for sharing with me the many adventures of life.

Abstract

Software must be regularly updated to keep up with changing requirements. Unfortunately, to install an update, the system must usually be restarted, which is inconvenient and costly. In this dissertation, we aim at overcoming the need for restart by enabling run-time changes at the programming language level.

We argue that the best way to achieve this goal is to improve the support for encapsulation, information hiding and late binding by *contextualizing* behavior. In our approach, behavioral variations are encapsulated into context objects that alter the behavior of other objects locally.

We present three contextual language features that demonstrate our approach. First, we present a feature to evolve software by scoping variations to threads. This way, arbitrary objects can be substituted over time without compromising safety. Second, we present a variant of dynamic proxies that operate by delegation instead of forwarding. The proxies can be used as building blocks to implement contextualization mechanisms from within the language. Third, we contextualize the behavior of objects to intercept exchanges of references between objects. This approach scales information hiding from objects to aggregates. The three language features are supported by formalizations and case studies, showing their soundness and practicality. With these three complementary language features, developers can easily design applications that can accommodate run-time changes.

Contents

1 Introduction **1**
- 1.1 First-class Contexts . . . 3
- 1.2 Thesis . . . 4
- 1.3 Contributions . . . 4
  - 1.3.1 Active Context . . . 4
  - 1.3.2 Delegation Proxies . . . 5
  - 1.3.3 Dynamic Ownership . . . 5
- 1.4 Outline . . . 6

2 Related Work **7**
- 2.1 Temporal Variability . . . 7
- 2.2 Active Variability . . . 11
- 2.3 Structural Variability . . . 15
- 2.4 A Taxonomy of Complexity . . . 18
- 2.5 Goals and Roadmap . . . 22
- 2.6 Conclusions . . . 23

3 Dynamically Updating Software with Active Context **25**
- 3.1 Introduction . . . 25
- 3.2 Running Example . . . 26
  - 3.2.1 The Problem with Updates . . . 27
  - 3.2.2 Lifecycle of an Incremental Update . . . 28
- 3.3 First-class Context . . . 30
  - 3.3.1 User-defined Update Strategy . . . 30
  - 3.3.2 Object Representations . . . 32
  - 3.3.3 First-class Classes . . . 33
  - 3.3.4 Threads and Contexts . . . 34
- 3.4 Implementation . . . 35
  - 3.4.1 Concurrency and Garbage Collection . . . 36
  - 3.4.2 State Relocation . . . 39
  - 3.4.3 Special Language Constructs . . . 39
  - 3.4.4 Further Details . . . 41
- 3.5 Validation . . . 42
  - 3.5.1 Evolution . . . 42
  - 3.5.2 Run-time Characteristics . . . 43

3.6 Discussion . . . 47
3.6.1 Portability . . . 47
3.6.2 Performance . . . 47
3.6.3 Development Effort . . . 48
3.6.4 Type Safety . . . 48
3.7 Related Work . . . 49
3.7.1 Class Redefinition . . . 49
3.7.2 Layers . . . 50
3.7.3 Additional Related Work . . . 50
3.8 Conclusions . . . 51

4 Contextualizing Behavior with Delegation Proxies 53
4.1 Introduction . . . 53
4.2 Delegation Proxies . . . 54
4.2.1 Propagation . . . 56
4.2.2 Closures . . . 58
4.2.3 Forwarding . . . 60
4.2.4 Transparency . . . 60
4.3 Examples . . . 61
4.3.1 Lazy Values . . . 61
4.3.2 Membranes . . . 62
4.3.3 Layers ⋆ . . . 63
4.3.4 Interceptors ⋆ . . . 64
4.3.5 Object Versioning ⋆ . . . 65
4.3.6 Read-only Execution ⋆ . . . 65
4.3.7 Dynamic Scoping ⋆ . . . 66
4.4 Semantics . . . 67
4.4.1 Identity Proxy . . . 70
4.4.2 Propagating Identity Proxy . . . 72
4.5 Implementation . . . 73
4.5.1 Performance . . . 75
4.5.2 Static Typing . . . 75
4.6 Related Work . . . 76
4.7 Conclusions . . . 78

5 Scaling Information Hiding with Dynamic Ownership 81
5.1 Introduction . . . 81
5.2 Filters and Crossing Handlers . . . 82
5.2.1 Default Policy . . . 85
5.3 Using Filters . . . 86
5.3.1 Iterators . . . 86
5.3.2 Read-only References . . . 87
5.3.3 Access Modifiers . . . 88
5.3.4 Sandboxing . . . 89

5.3.5 First-class State . . . 89
5.4 Using Crossing Handlers . . . 91
5.4.1 Defensive Copying . . . 91
5.4.2 Remoting . . . 92
5.4.3 Synchronization . . . 93
5.5 Security . . . 93
5.5.1 Ownership Transfer . . . 93
5.5.2 Reflection . . . 94
5.6 Semantics . . . 95
5.6.1 Ownership and References . . . 95
5.6.2 Topics and Filters . . . 96
5.6.3 Paths . . . 96
5.6.4 Accessibility . . . 97
5.6.5 Validity of References . . . 97
5.6.6 Instantiation . . . 98
5.6.7 Aliasing . . . 98
5.6.8 Ownership Transfer . . . 99
5.7 Implementation . . . 100
5.7.1 Closures and `self` . . . 100
5.7.2 Primitive Types . . . 100
5.7.3 Control Flow . . . 101
5.7.4 Ownership Transfer . . . 101
5.7.5 First-class Classes . . . 101
5.8 Experiments . . . 101
5.8.1 Adapting the Web Server . . . 102
5.8.2 Performance . . . 104
5.9 Discussion . . . 106
5.10 Related work . . . 107
5.11 Conclusions . . . 109

6 Conclusions **111**
6.1 An Extended Toolbox . . . 111
6.2 Strengths and Weaknesses . . . 113
6.3 Open Questions . . . 114

1 Introduction

Software systems must regularly modify their behavior to adapt to changing requirements or changing configurations. Such changes frequently entail the restart of the software system, which is both inconvenient and costly. Instead, we aim in this dissertation at supporting *run-time changes*, *i.e.*, changes without a restart. The term *run-time change* covers both unanticipated changes like new requirements, and anticipated changes like configuration changes. When the distinction is necessary, we speak of *run-time evolution* for the former category and *run-time adaptation* for the latter.

The support for run-time changes is closely related to the modularity of the software system, and the underlying support for modularity by the language: in a modular software system, units make few assumptions about each other [130] and can be easily replaced without entailing global modifications. This makes the overall system easier to change.

The following modularity principles are particularly relevant to enable run-time changes:

Encapsulation. A programming language with support for *encapsulation* enables elements that belong together to be bundled into cohesive units. A common example of encapsulation is the concept of class in most object-oriented languages. A class is a unit that encapsulates both behavior and state. Other mechanisms for encapsulation exist. For instance, some languages enable classes to nest within others [29], some rely on the concept of package [71], or some provide the concept of trait to encapsulate pure behavior [144].

Information hiding. A programming language with support for *information hiding* enables units to hide their implementation details. This

prevents units from depending too much on each others. Instead, they communicate via stable interfaces that expose clear abstractions. A common example of information hiding is the use of access modifiers to restrict the visibility of methods. Another common example is the policy regarding state accesses. Some languages make the state of an object accessible only to the object itself, while others are more permissive.

Late binding. A programming language with support for *late binding* resolves symbolic names between units as late as possible. For instance, if a unit depends on another unit named "engine", this dependency should be resolved only when the "engine" is actually accessed. A common example of late binding is dynamic dispatch: the method that will be executed is resolved dynamically at invocation time. The more late bound a language is, the easier it is to change applications. For instance, if class names are resolved to actual classes when first used, it isn't necessary to recompile the whole system to change a single class and the change can take effect upon the next restart. If class names are resolved each time the symbol is used, the software system gains in dynamism and classes can be changed at run time.

Together, these three principles favor an organization of code where it is easy to change design decisions [130]. Unfortunately, the actual support for encapsulation, information hiding and late binding found in mainstream programming languages is still insufficient to make run-time changes practical:

On one hand, statically-typed languages have a type discipline that supports strong information hiding, but that restricts the ability for late binding. The behavior of an object can be changed only if it preserves the type signature. On the other hand, dynamically-typed languages support advanced forms of late binding, but the lack of static types restricts their ability to enforce information hiding. The behavior of an object can change over time, but this can compromise safety. In both categories of languages, the unit that encapsulate behavior is the class. Coordinating changes below the class level, *e.g.*, individual methods, or above the class level, *e.g.*, behavior crosscutting multiple classes, is challenging.

Faced with these difficulties, developers must rely on convoluted design strategies to make software changeable at run time. They must devise ad hoc approaches to detect, load and activate changes using various programming languages features (*e.g.*, Java class loaders or proxies) and programming idioms (*e.g.*, strategy or visitor design patterns).

1.1 First-class Contexts

The main obstacle towards run-time changeability is the lack of support for encapsulating behavioral variations and activating them. When it exists, the support for behavioral variations affects whole classes of objects, and variations are activated permanently and globally. We argue that the best way to enable run-time changes is instead to *contextualize* variations.

In our approach, behavioral variations are encapsulated into *context* objects that alter the behavior of other objects locally in time and space: a context object can either scope variations to *dynamic extents* or *structural extents*. In the first case, the behavioral variation will affect only objects accessed during the evaluation of method on the context object. In the second case, the behavioral variation will affect a certain set of objects reachable from the context object. We call these two forms of scoping *active variability* and *structural variability*.

A context object is a regular object and can implement business logic. If the context object is used solely to contextualize variations and does not contain any business logic, it is called a *first-class context*.

Figure 1.1 and Figure 1.2 depict the two forms of variability. In Figure 1.1, the context object varies the behavior of the object that is accessed during the evaluation of the method `switch`; in Figure 1.2, the context object varies the behavior of the object it references.

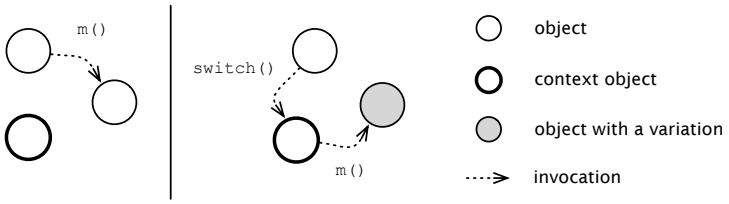


Figure 1.1: Active variability.

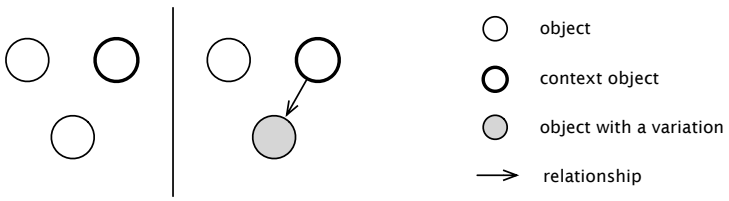


Figure 1.2: Structural variability.

1.2 Thesis

We formally state our thesis as follows:

> *First-class context objects implementing active and structural variability enable run-time changes by improving the support for encapsulation, information hiding and late binding.*

1.3 Contributions

In the next chapters, we show how first-class contexts implementing active and structural variability improve the support for encapsulation, information hiding and late binding in the following ways:

Encapsulation Context objects enable the encapsulation of behavioral variations spanning multiple objects into first-class entities.

Information hiding Context objects can make existing methods fail depending on the caller, thus preventing illegal invocations.

Late binding Context objects provide a fine level of control to rebind behaviors locally in time and space, instead of globally and permanently.

The discussion is organized around three language features — active context, delegation proxies, and dynamic ownership — that form the core contributions of the thesis. Below is a brief overview of the design, implementation and validation of each feature. Figure 1.3 presents an overview of their respective benefits from the modularity point of view.

1.3.1 Active Context

Active context is an approach to unanticipated run-time evolution. Behavioral variations that are immediate and global are a threat to type safety. Instead, we introduce first-class contexts to represent versions of the software and enable existing behaviors to be overridden on a per-thread basis rather than globally. Individual threads run in distinct contexts which results in an incremental switch from the old to the new behavior without compromising type-safety. The state is transferred between versions using bidirectional transformations. Active contexts avoids the need for awkward patterns to support unanticipated changes.

We have used several existing web servers as case studies. We show that run-time evolution can be achieved with an acceptable performance penalty and an acceptable development overhead, both for dynamically-typed and statically-typed languages.

The approach improves the support for *encapsulation* (by encapsulating software versions into contexts) and *late binding* (by rebinding incrementally the behaviors of the objects).

We realized a prototype for Java (`github.com/ewernli/theseus`) and a full implementation for Smalltalk (`http://ss3.gemstone.com/ss/Theseus.html`). This approach has been published in workshops [169, 170], a conference [168], and a journal [172].

1.3.2 Delegation Proxies

Delegation proxies are building blocks to implement contexts that encapsulate fine-grained crosscutting behavioral variations. They complement active contexts, which operate at the class level and are not appropriate to rebind the methods of an object selectively.

Delegation proxies are proxies that operate with delegation instead of forwarding. This way, a behavioral variation can propagate in a dynamic extent. By default, delegation proxies support homogenous variations, *i.e.*, multiple methods have the same variation. Support for heterogenous variations, *i.e.*, methods have specific variations, can be built on top of homogenous variations.

We show that delegation proxies enable run-time adaptations similar to dynamic aspects. Additionally, we show that the adaptions can help implement recovery blocks and immutable objects, which both improve safety.

The approach improves the support for *late binding* (`self` is late bound to the delegation proxy) and *encapsulation* (crosscutting variations can be encapsulated using delegation proxies).

We realized an implementation for Smalltalk (`http://ss3.gemstone.com/ss/Amop.html`). This approach has been submitted for publication in a conference.

1.3.3 Dynamic Ownership

To be replaceable, objects must communicate through stable interfaces. We show with dynamic ownership that structural variability with first-class contexts can be used to improve information hiding. With dynamic ownership, aggregates can hide their implementation details.

The approach structures objects in a tree and the behavior of an object becomes dependent on the relative positions of the sender and the receiver in the tree. The behavioral variation is executed when an incoming reference to an aggregate is established. The variation can fail, or take some alternative actions, *e.g.*, copy the object defensively. The owner of an object can be changed at run time.

	Act. Context	Del. Proxies	Dyn. Own.
Encapsulation	Encapsulate software versions into contexts	Encapsulate crosscutting variations	Encapsulate cross-cutting aliasing policy
Information Hiding			Scale information hiding from objects to aggregates
Late Binding	Rebind incrementally the behaviors of the objects	`self` is late bound to the delegation proxy	

Figure 1.3: Improvements of the language features

We demonstrate with a case study on an existing web server that dynamic ownership is applicable to existing systems and can help spot unwanted dependencies. We also evaluate the performance overhead and show it is acceptable.

The approach improves the support for *information hiding* (by scaling information hiding from objects to aggregates) and possibly *encapsulation* (by encapsulating crosscutting aliasing policy like defensive copying).

We realized a prototype for Smalltalk (`http://ss3.gemstone.com/ss/DynOwn.html`). This approach has been published at a symposium [171].

1.4 Outline

The main chapters of this thesis present each one language feature. The thesis is thus organized as follows:

Chapter 2 gives an overview of the related work under the perspective of temporal, active and structural variability.

Chapter 3 presents *Active Context*, an approach to unanticipated run-time evolution.

Chapter 4 presents *Delegation Proxies*, an approach to implement active variability.

Chapter 5 presents *Dynamic Ownership*, a flexible approach to enforce information hiding.

Chapter 6 concludes the dissertation and outlines future work.

2 Related Work

In this chapter, we survey the state of the art in enabling run-time changes. First we review existing approaches depending on their mechanics in terms of variability. In addition of active and structural variability, we introduce the term *temporal* variability to refer to globally and permanent changes. Second, we organize the existing approaches into a taxonomy that characterizes the complexity of the changes they support. Finally, we identify three levels of complexity that we aim at improving the support for.

2.1 Temporal Variability

Temporal variability refers to global and permanent variations of the behavior of objects over time.

Classic Objects Several of the classic design patterns [65] aim at enabling temporal behavioral variations. This is notably the case of the following patterns: the state design pattern, which "allows an object to alter its behavior when its internal state changes"; the strategy design pattern, which "allows one of a family of algorithms to be selected on-the-fly at runtime"; the decorator pattern, which "dynamically adds/overrides behavior in an existing method of an object". The popularity of these patterns is a clear witness of the need for temporal variability.

Reclassification The ability to change the behavior of an object after it was created is called *reclassification*. The ability to reclassify objects has been explored since the early days of object-orientation, since it is common

for objects to have changing states or phases. In statically-typed languages, general reclassification is however a loophole that compromises type safety. Also, if the fields of two classes differ, the semantics of reclassification become inevitably complicated. These two problems explain why the support for general reclassification never became mainstream.

To be practical, reclassification must be constrained. In Smalltalk, objects can be reclassified using `primitiveChangeClass`. This operation fails if the classes have incompatible layouts.

With *Wide Classes* [148], objects can be *widened* from a superclass to a subclass, and later on *shrunk* from a subclass to a superclass. Widening and shrinking are more disciplined than general reclassification. Still, the ability to shrink objects makes wide classes unsuitable for static typing. Subclasses can have additional fields that can be initialized during widening.

In the *Fickle* language [53, 46], a *root* class can be reclassified into one of its pre-defined *state* subclasses. State subclasses cannot define new fields. Object layouts are not affected by reclassification. Developers must annotate methods with the list of reclassifications they might perform. Type safety is checked with a type and effect system. This guarantees type safety from within one thread of execution. In the presence of multiple threads of executions, type safety could still be compromised. To ensure type safety, the type and effect system is leveraged to delay operations until when they will provably succeed.

Typestate Reclassification can help to represent an object during the different phases of its lifetime. For instance, the behavior of a file that can be either open or closed can be captured in two distinct state subclasses. The transitions between states are however not modeled, nor constrained. An early work that explicitly modeled state transitions was Pernici's *Objects with Roles Model* [131] . The *Fugue* programming language [47] enables the expression of state-related object invariants that are statically enforced. Bierhoff and Aldrich built up on Fugue to propose a more lightweight approach to enforcing legal transitions of states [20]. The *Plaid* programming language made transitions explicit, as well as the state, which are modeled as first-class entities [154]. The language is dynamically-typed and does not enforce legal transitions statically.

Object Swapping The ability to swap two objects (all references to the two objects are exchanged but identity is retained) could be considered the dual of reclassification. If an instance of the "destination" class is used to swap an object, the object effectively appears to change its class while retaining its identity. Inversely, reclassifying two objects and updating their states can appear as a swap of two objects if coordinated correctly. A principled approach to object swapping was proposed by Costanza in the

programming language Gilgul [43], which decouples the concepts of reference and identity. The language provides static type safety by constraining swapping to objects of similar types or subtypes.

Swapping can be pushed to the next level by enabling whole components/services to be swapped. COM and Enterprise Java Beans (EJB) are two mainstream component models. Components/services are loosely coupled and communicate via interfaces; they are hosted in an application server and can be redeployed dynamically. In recent years, OSGi gained traction as a stand-alone component model that enables dynamic reloading. Certain approaches enable components to be manipulated programmatically, *e.g.*, Oriol's LuckJ [126].

Object Specialization Rather than reclassifying objects between existing classes, objects can have their own behavior that can be specialized per-instance. Beck's *Scriptable Objects* [14] describe a pattern to do on top of a class-based language, namely Smalltalk. Essentially, the idea is to dynamically generate an anonymous class per object. The class can then be dynamically modified to alter the behavior of its unique instance. This is similar to *eigenclasses* in Ruby. The granularity of the specialization is in this case the method.

The granularity of the specialization can be increased while retaining the concept of per-object specialization. Ressia *et al.* proposed *talents* [140], a technique to specialize objects using trait-like units of behavior. Bettini *et al.* proposed a similar construct for Java [18, 19], building on Smith and Drossopoulou's earlier work [149]. In these approaches, the specialization replaces existing behaviors.

Instead, the specialization can merely override the behavior to "decorate" the original behavior. In *Featherweight Wrap Java* [17], objects can be dynamically extended with decorators. Büchi and Weck proposed *Generic Wrappers* [31], an extension of Java where objects can be specialized with decorators at time of creation (but not later).

Prototype-based languages like Self [161] rely on delegation rather than inheritance to reuse behaviors. Object specialization is at the heart of such languages. Objects can naturally replace or override the behavior of their prototype. Conversely, reclassification translates to a trivial change of the prototype of an object. Dynamic proxies [162, 60, 41] can to some extent be used to implement prototype-like specialization. However, since they operate by forwarding instead of true delegation, this results in problems with self-sends [78].

Class Redefinition An alternative way to change the behavior of objects at run time is to redefine classes themselves. In Smalltalk, all changes happen reflectively via class redefinitions. Naturally, redefining classes poses

similar problems of type safety and object layout compatibility as reclassification. In Smalltalk, new fields in the class definition are initialized to `nil`. Since the language is dynamically typed, run-time type errors raise exceptions as usual. This is the common approach taken by other dynamically-typed language as well.

Many approaches have been proposed to enable class redefinitions in statically-typed languages like Java. They are inherently trade-offs between type safety and flexibility. Early versions of the JVM did not support any class redefinition facility. Dynamic class loading was however possible via class loaders which can be used to simulate dynamic changes in a very limited way [97]. Modern versions of the JVM enable the redefinition of a method as long as its signature doesn't change. Such features were first researched by Dmitriev in PJama [52]. Less constraining approaches include the Dynamic Code Evolution VM (DCEVM) [174], JRebel [87], Javaleon [72], JVolve [153] and Javadaptor [138]. These approaches supersede older approaches like JDrums [5], the rewriting technique of Orso *et al.* [127] or the Dynamic Virtual Machine (DVM) of Malabarba [105].

The DCEVM and JVolve are custom Java virtual machines. JRebel, Javaleon and Javadaptor use code transformation to enable class redefinitions. These recent approaches differ along several dimensions: 1) the level of type safety, 2) the flexibility in changes supported, 3) the constraints about the timing of changes, 4) the flexibility in custom object conversions, 5) the time of object conversions. With respect to the flexibility of changes supported, Gustavsson established a classification of changes in Java [73] with their frequencies. A similar study was performed for C [117].

DCEVM. The DCEVM performs changes instantly. It supports layout changes but no custom conversion (like Smalltalk). Run-time type errors raise `NoSuchMethodError`. It supports arbitrary changes to the class and interface hierarchy. When the change is installed, a full garbage collection is forced and converts the objects eagerly.

JRebel. JRebel is a proprietary software that enables arbitrary changes, except modifying the class hierarchy and the addition or removal of interfaces. It operates by code rewriting. It is meant to avoid redeployment during developments and thus does not support custom object conversions.

JVolve. JVove supports arbitrary changes, except changes to the class hierarchy and the addition or removal of interfaces. It converts objects eagerly during a forced garbage collection. To be type safe, it constrains the timing of updates using temporal *safe points*. It supports custom object conversions.

Javaleon. Javaleon supports arbitrary changes to classes, the class hierarchy and the interfaces. It converts objects lazily and enables custom

conversions. Type compatibility across versions is achieved by wrapping objects with adapters. Adapters will preserve type safety, but not necessary the semantics, if old code accesses new objects (*e.g.*, if a method is renamed). Javaleon is limited to NetBeans components.

JavAdaptor. JavAdapor supports similar capabilities as Javaleon, without being limited to NetBeans components. It converts objects eagerly using JVMTI's `referringObjects` facility.

For the dynamic updates of productions systems, type safety is not enough. A type-safe update might lead to functional errors if the logic of collaborating objects is changed during an ongoing collaboration. The timing of updates to guarantee update safety beyond type safety has been the subject of intensive research. Ginseng [79, 118, 116] and Kitsune [76] are two approaches for C systems. In these approaches, developers specify update points when the update can be installed. Such *safe points* correspond to quiescent program states where no method that will be changed is currently active. UpStare [104] avoids timing constraints by enabling active methods to be updated using stack reconstruction. No such technique exists for object-oriented systems at the moment. POLUS [34] also avoids quiescent update points by enabling both old and new code to coexist; the system keeps old and new data coherent using eager bidirectional conversions.

Object Migration Dynamic updates of production systems require the use of convertors to specify the migration of objects from their old layout to the new layout. Specifying such convertors is cumbersome and error prone. Transformations are usually simple (field addition, renaming, suppression, type conversion) and complex transformations are occasional [73].

Static analyses of the evolution of Java system classes reveal that almost all changes to object layouts can be automatically generated [132] to accommodate mismatches during serialization and deserialization. Using dynamic analysis, the heap of the old and new programs after similar scenarios can be compared to infer object converters automatically [103].

If the conversion of a parent object depends on a child object, the conversions must be orchestrated so that first the parent is converted, then the child. If objects are converted lazily when accessed, such ordering might be broken, leading to conversion errors. Boyapati *et al.* solved this problem using ownership types [24].

2.2 Active Variability

Active variability refers to behavioral variations that are activated in the *dynamic extent* [156] of a method invocation.

Classic Objects Active variability lets the behavior of an object depend on the client (direct and indirect) accessing it. Without native support for active variability, various design patterns and programming techniques can be used to simulate it.

The visitor design pattern [65] externalizes the logic of a family of collaborating objects into an auxiliary object, *i.e.*, the visitor. Clients of objects in the family must provide a concrete visitor to use them. A client can thus adapt the behavior of the objects in the family to its needs.

In the context of user interfaces, the model-view-controller pattern [93] decouples the graphical presentation of objects from their states. The model contains state, and the views contain presentation logic.

A more recent design pattern is the Ambient Context pattern [146], which "makes a dependency available to every [object] without polluting their [interface]". The shared dependency is made available to all objects via a global resolver that returns a per-thread instance of the dependency.

Nested Diagnostic Context [74] is a popular pattern (notably popularized in Log4j) which uses dynamic variables to add contextual information to diagnostic messages. This way, an object can for instance specify the current transaction it is involved in while logging its activity.

Lastly, an object can implement active variability by inspecting the stack and adapting its behavior depending on its direct and indirect callers. This is for instance possible in Java using `Thread.getStackTrace(...)` or Smalltalk using `thisContext`.

Roles Within an object system, objects of the same class might serve different roles to other objects. For instance, a person can be the assistant or the manager of another person. The concepts of "assistant" and "manager" are roles. They aren't intrinsic properties of a person, but instead properties of the relationship between two entities. Similarly, the concepts of "adolescent" and "adult" are not intrinsic properties of a person but states or phases of a person [151].

In class-based languages, it is tempting to model both entities and roles as classes. This is the classical *is-a* form of inheritance. Unfortunately, roles and entities are not subsets of each other, which inevitably leads to incoherences in the design. For instance, if a system has two entities *Person* and *Organization*, and two roles *Customer* and *Supplier*, it is impossible to correctly capture the desired relationships between the four concepts with only classes [151].

Certain languages make roles explicit either as names on relationships [61] or explicit entities [94]. Benefits of the second approach are the support for role-specific state and the ability to dynamically acquire or lose roles; the drawback is the possible confusion between a role and its target object, since their identities differ. Objects may simultaneously play multiple roles to different clients.

Coercion and Lifting A role can be seen as a wrapper around its target object. The challenge is to transparently and consistently represent sets of collaborating objects with their appropriate roles depending on the client accessing them. By coordinating the application of roles to multiple objects, roles can model cross-cutting slices of behavioral. For instance, presentation logic that cross cut multiple objects can be organized into roles.

One prominent language supporting the coordination of explicit roles is Object Team [77]. In Object Team, an object can be *played by* a role, and related roles (*i.e.*, cross-cutting slices of behavior) are organized into *teams*. The implicit representation of an object with the corresponding role is called *lifting* (the opposite is called *lowering*). This kind of polymorphism is called translation polymorphism [84].

A similar form of polymorphism is enabled in Scala where implicit type conversions are called *views*[1]. Warth's Expanders [167] enable the implicit translation of collaborating objects to apply cross-cutting slices of behavior. In both approaches, base objects and translated objects have distinct entities. Bergel's Classboxes [15] enable similar client-specific adaptations, but without resorting to distinct identities. Unlike with Expanders, the compilation is not modular and all views must be known at compile time.

Context-oriented Layers The cross-cutting slice of behavior expressed with roles, expanders and classboxes isn't explicitly manipulable; it is applied based on syntactic rules. In context-oriented programming [80], the slice is explicit and can by activated dynamically. In ContextL [42], a variant of Lisp developed by Costanza and Hirschfeld, the slice of behavior is called a layer, and can be it can be dynamically activated in the control flow, scoping variations to threads. Numerous variants of this approach were later on explored, giving birth to ContextS (Smalltalk), ContextJ (Java), ContextPy (Python), ContextJS (Javascript), and ContextG (Groovy) [7, 143]. These variants differ in the way variations are encoded (layer-in-class or class-in-layer), how layers are activated and composed, whether state can be specific to a layer, and whether layers can be reflectively accessed [7].

Other approaches enable global activations of layers. These approaches are thus closer to *temporal* variability than *active* variability. For instance, Löwis *et al.* support thread-based and global layer activations, as well as

[1] `http://www.scala-lang.org/node/130`

dynamic variables [165]. The Ambience programming language [68] and Context Traits [69] similarly support global layer activations. Explicit layer activations was shown to be useful to support non-functional requirements like mobility and multi-tenancy [160], or the modularization of graphical presentation logic [8].

Aspects Aspect-oriented programming [91] also aims at the encapsulation of cross-cutting slices of behavior. Aspects represent generic variations (*i.e.*, advices) that can be applied in multiple places (*i.e.*, join points). The technique can be extended to support dynamic activations of aspects [135], *e.g.*, to enable and disable logging at run time. Dynamic aspects correspond to temporal variability. However, dynamic aspects can also implement active variability if join points can be conditionally activated depending on the control flow (*e.g.*, using AspjectJ's `cflow` operator). For instance, access permissions in Java (using `doPriviledged( ... )`) can be seen as an aspect [159] implementing active variability: accesses to sensitive resources trigger a check which inspects the stack to authorize or deny the access. The main difference between dynamic aspects and layers is that aspects best represent generic variations to apply in multiple places, while layers embody specific variations to apply in unique places. Apel *et al.* refer to both forms of variations as homogeneous concerns and *heterogenous concerns* [6]). Both help the multi-dimensional decomposition of concerns.

Subjectivity Harrison and Ossher [75] decouple object identity from behavior and state. In their approach, different "subjects" can view objects with different behaviors and states. The only intrinsic property of an object is its identity, which is shared across subjects. Behavior and state are extrinsic properties of objects, that depend on the current subject activation.

Ungar and Smith developed Us [150], an extension of the language Self with a notion of "perspective". Perspectives are explicit and can be composed in a layered fashion to provide different views of objects. State and behavior are unified in Self with slots. In Us, the value of a slot can be redefined by a layer. All message sends are parametrized with a perspective, which by default is the `here` perspective.

Predicate Dispatch Predicate dispatch enables methods to be dispatched based on arbitrary predicates [58]. Methods can be dispatched based on the types of their arguments and based on the state of the arguments (including the possible relationships between objects). Predicate dispatch thus generalizes multi-methods [113] and pattern matching. Layers, dynamic AOP and subjectivity can be seen as forms of predicate dispatch where methods are dispatched based on the type of the receiver and additional dynamic variables.

Environments A dynamic binding is a binding that exists only during the evaluation of a given expression and its sub-expressions [112]. Variables bound this way are called dynamic variables. Dynamic binding is usually contrasted with lexical binding, which is the norm for modern languages (it is however interesting to note that exception handling is a disguised form of dynamic binding common to most languages). The mapping of names to values used to resolve bindings is called the *environment* of an expression. How bindings "propagate" is defined by the programming language semantics. For instance, languages can implement closures either by copying the environment at the time of creation (*e.g.*, Java) or by referencing the original environment (*e.g.*, Smalltalk). The ability to control the environment and the related bindings gives a very high expressive power to a language. Tanter extended Lisp [156] with the ability to explicitly control how bindings propagate with user-defined policies. In Piccola [119], the environment is explicit. This enables a very flexible form of modularity where functions can be loaded and rebound to different names. Dependency Injection [136] is the most popular form of environment control. Essentially, the configuration of the dependency injection framework represents a set of bindings.

2.3 Structural Variability

Structural variability refers to variations of the object behaviors based on relationships between objects.

Classic Objects The behavior of an object naturally depends on the behavior of its dependent objects. Objects thus inherently provide support for structural variability. This structural variability is unidirectional: the behavior of a parent object will depend on the behavior of the object it references, but not the opposite. Also, this structural variability is explicit: the mere referencing has no effect unless the referenced object is effectively accessed. Other forms of structural variability exist, where relationships between objects implicitly impact behavior.

Meta-objects A meta-object is an object which defines the behavior of one or more base objects. In languages with first-class classes, a class is a meta-object that defines the behavior of its instances [90]. In reflective systems, base objects and meta-objects coexist in the same object space and are causally connected. Smalltalk [67] is a reflective system with first-class classes, meta-classes, and activation frames. CLOS [48] and Ruby [108] are other dynamic languages with reflective architectures. Supporting a causal connection between base and meta-objects is usually not possible

in statically-type languages, for the same reasons already presented in the section about classes redefinitions.

Reflection is split into two categories of facilities: structural and behavioral reflection. The difference between the two is best understood if we consider an application running on top of an interpreter. Structural reflection enables the application to reflect on itself; behavioral reflection enables the application to reflect on the interpreter running it, thus changing the semantics of the language. Arguably, the distinction between the two is blurry, since a behavioral change can be simulated with a structural change that rewrites the sources of the application. A weaker variant of full reflection is called load-time reflection, which essentially supports changes only when classes are initially loaded. Javassist [35] is a popular framework for load-time structural reflection in Java.

Since reflection is costly, it is interesting to limit it in space (*i.e.*, on which syntactic entities reflection is enabled) and time (*i.e.*, when reflection is enabled). This approach was first proposed in Reflex [158], for Java, and then Reflectivity [49], for Smalltalk. In Ressia's object-centric reflection [139], a meta-object can be associated with an individual object to alter its behavior. Dynamic proxies can be used to simulate meta-objects by intercepting all accesses to a given target object. In Smalltalk, method wrappers [54] are equally convenient to intercept messages sent to instances of a specific class.

A meta-objects with a causal connection with its base object can be seen as a form of structural variability (since it defines the behavior of the base objects it relates to) and at the same time a mechanism to achieve temporal variability (since the meta-object can be modified to change the behavior of its based objects).

Virtual Classes A virtual class [102] is a nested class that can be overridden by subclasses of the outer class. For instance, a class `Graph` can nest virtual classes `Edge` and `Node` to enable the specialization of graphs with different types of edges and nodes. A virtual class is a member of an object, *e.g.*, `aGraph.Edge`. When instantiating new objects, class names are bound to their class definitions taking into consideration the relationship between a class and its enclosing object. If an edge creates a node, it will create a node that matches the definition in its enclosing graph.

Both virtual classes and parametric polymorphism enable genericity and advanced code reuse. Pros and cons of either approach depend on the specific design problem to solve [30].

Two classes `aGraph.Edge` and `anotherGraph.Edge` have distinct types, unless it can be proven that `aGraph` and `anotherGraph` are instances of the same class. Redefining `Edge` or `Node` in a subclass of `Graph` does not replace the previous definition, but rather enhances it with *furtherbinding* [57].

The ability to virtualize superclasses provides mixin-like capabilities to the language. For instance, if graphs must be specialized into weighted graphs (with weighted edges and weighted nodes), there are two possibilities: the specialization of classes `Edge` and `Node` with an additional `weight` attribute, or the rebinding of the superclass of `Edge` and `Node`, usually `Object`, to a class `WeightedObject` which defines a common `weight` attribute.

With virtual classes, an object like `aGraph` acts as a repository of compatible classes. A set of compatible classes is called a *family* and virtual classes enable *family polymorphism* [56]. Furthermore, if virtual classes can inherit from other virtual classes, this enables *class hierarchy inheritance* [57], essentially solving the problem of parallel class hierarchies.

With hierarchy inheritance, classes in two inheritance hierarchies do not need necessary to map one-to-one. Classes can be added in the second inheritance hierarchy as long as they don't break the original inheritance hierarchy. Taking the example of a compiler, if the class `PlusOp` inherits from `Expr` in the original hierarchy, the new hierarchy can introduce an intermediate class `BinaryOp` between `Expr` and `PlusOp` [57].

Several mainstream languages support various flavors of virtual classes and class hierarchy inheritance:

BETA. The most recent incarnation of the language, gbeta [102, 57, 56], supports family polymorphism and class hierarchy inheritance. It defines combinators to compose hierarchies. For instance, the `Graph` family could be specialized into `WeightedGraph` and `ColouredGraph` and both specializations can be combined into a class `ColouredWeightedGraph`. The language is statically-typed.

Newspeak. Newspeak [29] is a language inspired by Smalltalk, Self and BETA. It supports family polymorphism and class hierarchy inheritance. Unlike BETA, a virtual class redefined in a subclass does not enhance the corresponding class in the superclass (*i.e.*, it does not necessary furtherbind it). Instead, developers can control whether they want to replace or enhance the base virtual class. The module system of Newspeak unfolds entirely from the use of virtual classes. The language is dynamically-typed.

J&. J& [123] supports virtual classes that are bound to classes, not objects. This facilities type checking but enables limited family polymorphism only. For instance, a structure that must contain a list of edges and a list of nodes of compatibles types can not be parametrized with a graph object that acts as a repository, as in Newspeak and gbeta. The approach also limits cross-family inheritance.

Scala. Scala does not support virtual classes nor class hierarchy inheritance. Instead, Scala has virtual types and self types [124]. With

these, it is possible to simulate virtual classes in certain cases. The class `Graph` could for instance be specialized into `WeightedGraph` [2]. Virtual types do not support the trivial addition of methods or fields as is the case with virtual classes.

Java. Java does not support virtual classes nor class hierarchy inheritance. It supports *inner classes* and *static nested classes* [71]. Instances of the former are bound to an enclosing object whereas instances of the latter are just like instances of regular classes. Static nesting serves only as namespacing.

Delegation Layer In Ostermann's Delegation Layers [128], objects are organized in a tree (using class nesting) and can be wrapped with an explicit slice of behavior. When a wrapped object accesses one of its children, the child is wrapped as well. A wrapped object represents thus a consistent "view" of the original object, including its nested children. For instance, wrapping `aGraph` with a coloring layer produces the object `aCouloredGraph`; when the object `aColouredGraph` is manipulated, the variation propagates to the nodes and its edges. Unlike context-oriented layers, delegation layers do not activate the variation based on the control flow, but based on the relationships between objects in the tree.

Ownership Objects must expose interfaces that hide their internals. To prevent accesses to an internal object, developers must design interfaces with care. Such a syntactic approach is limited in the face or reflection (any object can be accessed) and developer mistakes (such as exposing a getter by mistake). To enforce the encapsulation of a system of objects, objects can be structured in a ownership tree which captures semantically the desired accessibility. Accessibility restrictions can then be enforced statically [39, 121, 23] or dynamically [70] at the language level. The restriction to apply depends on the relative positions of the caller and callee in the ownership relationship.

2.4 A Taxonomy of Complexity

The mechanisms presented previously were categorized based on a very coarse-grained taxonomy of how they work. We now refine this taxonomy to account for a better overview of the complexity of the changes they enable.

[2] `http://docs.scala-lang.org/tutorials/tour/explicitly-typed-self-references.html`

Our refined taxonomy uses the following dimensions:

1. *Type-preserving vs. Type-modifying* A mechanism is type-modifying if it can change the signature of existing methods, otherwise it is type-preserving. Accordingly, the addition of methods is considered type-preserving.

2. *Quiescent vs. Busy* A mechanism is quiescent if the system must be quiescent for changes to be safe, otherwise it is busy.

3. *Anticipated vs. Unanticipated* A mechanism supports *anticipated* variability if the the number of possible object behaviors is enumerable, *i.e.*, it could be computed at compile time. A mechanism supports *unanticipated* variability if variations can be dynamically loaded, *i.e.*, the number of variations cannot be computed at compile time.

4. *Stateless vs. Stateful* A mechanism is stateful if it supports changing the state of objects as well as behavior, otherwise it is stateless.

5. *Fine-grained vs. Coarse-grained* A mechanism is fine-grained if it supports changing one class/object at a time, and coarse-grained if it can impact multiple classes/objects (depending on whether it is class-centric or object-centric).

6. *Object-centric vs. Class-centric* A mechanism is class-centric if it operates on classes, and object-centric if it operates on objects.

The five first dimensions of the taxonomy capture the complexity of the changes supported by the mechanism. The last dimension about object or class centrism helps characterize the mechanisms, but does not represent a dimension of the complexity. For each complexity dimension in the list, the option on the left is simpler to support than the option on the right. For instance, supporting stateless changes is simpler than supporting stateful changes.

The simplest changes to support are thus fine-grained, stateless, anticipated, quiescent, and type-preserving. This is what objects enable, without any language extension. For instance, the strategy design pattern falls in this category. Conversely, the most complex changes to support are coarse-grained, stateful, unanticipated, busy, and type-modifying. This is what advanced techniques for dynamic updates enable. These techniques are not language features, though. In between these two extremes, there is a gradient of techniques and mechanisms with various levels complexity.

We synthesize the related work in the following main lines of research: Classic Objects, Aspects, Reflection, Contextual Layers, Global Layers, Roles, Object Adaptations, Family Polymorphism. The description and positioning of these lines of research are as follows (see Figure 2.1):

	Objects	Aspects	Reflection	Contextual Layers	Global Layers	Roles	Object Adaptations	Family Polymorphism	Active Context	Delegation Proxies	Dynamic Ownership
Type-**p**reserving vs. Type-**m**odifying	P/M.	P	P/M	P	P	P	P/M	-	P/M	P	P
Quiescent vs. **B**usy	Q	Q	Q	B	Q	B	Q	-	B	B	Q
Anticipated vs. **U**nanticipated	A/U	A	U	A	A	A	U	-	U	A	A/U
State**l**ess vs. State**f**ul	F	L	F	L	L	F	F	-	F	L	L
Fine-grained vs. **C**oarse-grained	F	C	F/C	C	C	F	F	-	C	C	C
Object-centric vs. **C**lass-centric	O	C	C	C	O	O	O	-	C	O	O

Figure 2.1: Comparison of the main lines of research.

Classic Objects. This line of research aims at understanding the expressiveness of objects alone, possibly with design patterns. Objects alone enable run-time changes by encapsulating the logic that must vary into objects that can be swapped dynamically, *e.g.*, the strategy design pattern. This technique is typically type-preserving, but it could be type-modifying in dynamic languages. For the replacement to be consistently applied, the system must usually reach quiescence. With objects alone, only anticipated changes are possible. If the language supports dynamic class loading and reflective instantiations, unanticipated changes are possible. State is not transferred between objects to swap, unless developers define a contract to do so. The unit of replacement is the object. This technique does not provide any guarantee beyond the object level. It is object-centric, and fine-grained.

Aspects. This line of research covers static and dynamic aspects. Aspects enable type-preserving homogenous variations applied in multiple join points. Dynamic aspects enable global adaptions to the system at run time. In this case, aspects are usually first-class. Some approaches enables aspects to be loaded dynamically, and thus enable unanticipated variations (*e.g.*, Previtali's approach to dynamic updates [33]). Aspects are stateless. Using special operators, advices that consider the control flow can be defined, *e.g.*,`cflow`. However, since aspects are activated globally, they can compromise consistency in multithreaded systems that are busy. The impact of aspects is coarse-grained since they can alter the behavior of multiple objects and classes. Aspects are usually class-oriented.

Reflection. This line of research covers traditional class-based meta-object protocols that reify classes as first-class entities. Classes are causally connected to base objects that reflect modifications to classes. In statically-typed languages, reflection is limited and can't change existing method signatures; in dynamically-typed languages, reflection can change the layout and signature of classes. Reflection can be used to evolve the system dynamically, but it assumes that the system is somehow in a quiescent state (this is for instance an assumption of the Smalltalk image model). Reflec-

tion enables unanticipated changes since arbitrary code can be compiled dynamically. When object layouts are changed, new fields are initialized with default values. Reflection does not support state transfer. Reflection operates usually at the class level. The unit of change is the class and reflection is usually fine-grained; however, certain systems enable changes to multiple classes at once (*e.g.*, Java and VisualWorks).

Contextual Layers. This line of research covers context-oriented programming with layers that are activated into dynamic extents [42, 143, 7, 165, 160]. Since the variation is not global, layers can be safely used in busy multi-threaded systems. This line of research subsumes subjective programming. Contextual layers enable type-preserving heterogenous variations, where the impact of a layer can be specified on a per-class basis (independently of the class-in-layer or layer-in-class discipline). Layers are first-class but cannot be usually dynamically loaded. Thus, they enable anticipated variations only. They usually do not support state variations. The effect of a layer is scoped to the dynamic extent of an expression. Layers are coarse-grained and impact multiple objects and classes.

Global Layers. This line of research covers context-oriented programming with layers that are globally activated and deactivated [69, 68]. Similarly to contextual layers, global layers enable type-preserving heterogenous, anticipated, stateless variations. Unlike contextual layers, the variation is activated globally based on events. Global layers will not compromise type safety in busy multi-threaded system, but could compromise consistency nevertheless. Also, unlike contextual layers, global layers impact objects rather than classes (*e.g.*, Context Trait Gonz13a). Multiple objects can be modified at once, so the mechanism is coarse-grained.

Roles. This line of research covers techniques to adapt individual objects based on roles. Roles resemble contextual layers, except that they provide a view on one single object, not multiple objects. They enable heterogenous variations. They are dynamic and the behavior of an object changes depending on which object accesses it. Therefore, roles are fine-grained and object-centric. Roles are usually known at compile time and enable thus only anticipated variations. The variation can be stateful. The variation is scoped to the client of the object; the role characterizes the relationship between the two.

Object Adaptations. This line of research covers techniques to adapt individual objects. It covers object-specific meta-objects, object specialization,

decorators, type states. Object adaptations are fined-grained and object-centric. Object adaptations are stateful and both behavior and state of an individual object can be adapted. For reflective techniques, classes can be loaded dynamically to enable unanticipated variations. The variation is global, which can compromise consistency in busy multithreaded systems. Usually, the variation is type-preserving, but it could be used to alter method signatures as well.

Family Polymorphism. This line of research covers techniques to generate variants of collaborating objects, without leading to parallel class hierarchies. These techniques usually leverage virtual class nesting to do so. They enable heterogenous variations, where the behavioral variation is specific to a class of object. New fields can be introduced into existing classes. The techniques are static in the sense that object behavior can't change after creation. Since these mechanisms are static, the criteria do not really apply and the cells are marked with a dash.

2.5 Goals and Roadmap

To evolve and adapt software at run time, support for unanticipated coarse-grained changes is required. As the classification shows, mechanisms to enable such changes are missing. Let us consider changes combining type or layout modifications as "advanced" changes; inversely, let us consider changes that preserve types and layouts as "simple". The support for unanticipated, coarse grained changes is as follows:

- Advanced changes when the system is busy – Lacunary. No technique covers this complexity of changes.
- Basic changes when the system is busy – Lacunary. Contexual Layers do not enable loading layers dynamically.
- Advanced changes when the system is quiescent – Lacunary. With classic objects, it is hard to make guarantees when multiple objects must be changed. Reflection does not support state transfer.
- Basic changes when the system is quiescent – Satisfying. With reflection, developers can update multiple method bodies when the system is quiescent.

We have argued in the previous chapter that active and structural variability helps introduce run-time changes by improving encapsulation, information hiding and late binding. Our goal in the following chapters will be to demonstrate this claim by improving the support for the three gaps

that we identified. We will show how different language features — namely active contexts, delegation proxies, and dynamic ownership — help address these gaps. Figure 2.1 positions the features in the classification.

2.6 Conclusions

We have surveyed existing mechanisms to create, replace and adapt the behavior of objects at various levels of scale and with various levels of dynamism. We observed that only few techniques provide the ability to encapsulate behavioral variations into an entity that can be loaded and activated at run time. These are necessary conditions to enable unanticipated run-time changes. Additionally, the need to change object layouts is frequently ignored and unsupported. The outcome of our analysis was the identification of three forms of run-time changes that lack support: advanced changes when the system is busy, basic changes when the system is busy, and advanced changes when the system is quiescent.

Without proper support for run-time changes, developers must combine existing mechanisms with ad hoc design strategies. Such techniques enable changes but provide little guarantee about their correctness. For instance, developers can combine class loaders with a careful design to enable the replacement of objects dynamically [97]. Without the necessary care, this technique can lead to memory leaks though. It is hard to guarantee that such leaks won't happen.

We set out to apply our approach to the three areas of deficiencies we identified and devise contextual language features to cover the gaps. In the next chapter, we present a feature that addresses the first gap. We introduce first-class contexts that encapsulate behavioral variations and state migrations, enabling this way incremental changes to the system without requiring quiescence.

3

Dynamically Updating Software with Active Context

3.1 Introduction

Software systems must be periodically updated to keep up with changing requirements. Without the ability to update software at run time, the software system must be redeployed and restarted for changes to take effect, which entails costs and might be intolerable for highly available systems. The key challenge for dynamically updating systems like web servers is to ensure consistency and correctness while maximizing availability.

Some form of dynamic updating can be supported at the language level with design patterns (state, strategy, visitor), reflection and in the case of Java, class loaders [97]. Design patterns and class loaders are however suitable only if developers can anticipate in advance units that can change or not. Reflection is on the other hand inherently unsafe: running threads might run both old and new code in an incoherent manner and old methods on the stack might presume type signatures that are no longer valid, possibly leading to run-time type errors. Also, reflective capabilities do not usually enable the atomic installation of multiple changes nor a precise control of how the state is migrated. These techniques support dynamic updates only *in the small*, *i.e.*, at the object-level.

For dynamic updates *in the large*, *i.e.*, involving multiple objects of different classes with state transfer, developers have to resort to external mechanisms. These mechanism interrupt the application to perform a global update of both the code and the state of the program when the program is quiescent. Unfortunately, quiescent global update points may be difficult to

reach for multi-threaded systems [116, 153]. Also, even with quiescence, a global update might not be possible due to the nature of the change, for example it would fail to update anonymous connections to an FTP server that mandates authentication after the update: the missing information cannot be provided *a posteriori* [118]. The problem of dynamic updates in the large, at the language level, is thus open.

In this chapter, we propose to solve this problem by taking a contextual approach: we represent software versions explicitly with first-class context objects. The context encapsulates the behavioral variations to introduce. Also, it does not activate variations globally, but instead implements active variability and scopes the variations on a per-thread basis. Different threads run different versions of the software, providing version consistency at the thread level. Instead of a global update, the update is *incremental*. It progressively switches from one context to another.

The update scheme is not fixed, and can be tailored to the nature of the application. For instance, the update of a web application can be rolled out on a per-thread, or per-session basis. In the latter case, visitors always see a consistent version of the application. However, different visitors might see different versions of the application until the update completes. Such a scheme would not be possible with a global update: one would need to wait until all existing sessions had expired before starting new ones. The overall consistency of the data is maintained by running bidirectional transformations to synchronize the representations of objects shared across contexts. We show that the number of such shared objects is significantly smaller than the number of objects local to a context, and that this strategy fits well with the nature of the event-based systems we are interested in. We call our approach *Active Context* and its implementation *Theseus*[1].

This chapter is organized as follows. First, we present our approach informally with the help of a running example in Section 5.2. We present our model in detail in Section 3.3 and our implementation in Section 5.7. We validate our approach in Section 3.5 and demonstrate that it is practical. We put our approach into perspective in Section 5.9 and we compare it with related work in Section 5.10 before we conclude in Section 5.11.

3.2 Running Example

To illustrate our approach let us consider the implementation of one of several available Smalltalk web servers[2]. Its architecture is simple; a web server listens to a port, and dispatches requests to so-called services that ac-

[1] In reference to Theseus' paradox: if every part of a ship is replaced, is it still the same ship?

[2] See `http://www.squeaksource.com/WebClient.html` (The name is misleading since the project contains both an HTTP client *and* server)

cept requests and produce responses. For the sake of our running example, let us assume that the server keeps count of the total number of requests that have been served. Figure 5.10 illustrates the relevant classes.

3.2.1 The Problem with Updates

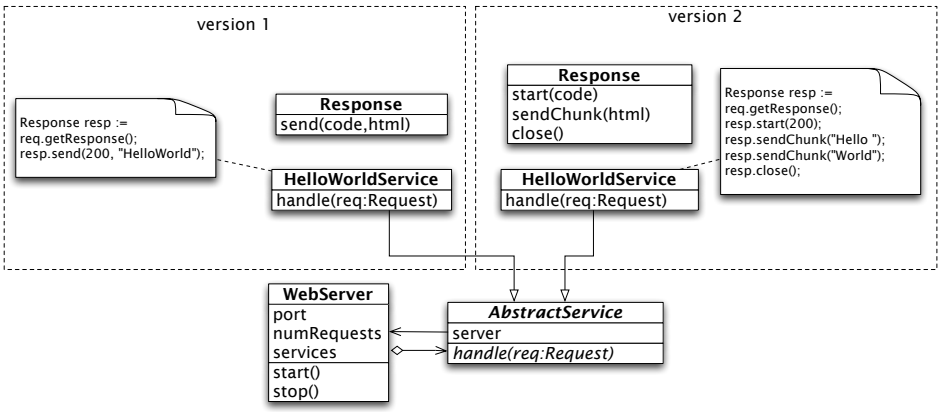


Figure 3.1: Design of the web server and a simple behavioral update

Let us consider the evolution of the Response API, which introduces chunked data transfer[3], also depicted in Figure 5.10. Assume that instead of sending "Hello World" over the wire we need to produce a sensible answer that takes some time. Installing such an update *globally* raises several challenges. First, both the `HelloWorldService` and `Response` classes must be installed together: *How can we install multiple related classes atomically?*

Second, the methods impacted by the update can be modified or added only when no request is being served: *When can we guarantee that the installation will not interfere with the processing of ongoing requests?*

Rather than performing a global update, it would be more appealing to do an *incremental* update, where ongoing requests continue to be processed according to the old code, and new requests are served using the new code. Note that the granularity of the increment might differ depending on the update. We could imagine that the modification of a check-out process spanning multiple pages would imply that the increment be the web session rather than the web request. Our solution to enable incremental updates is to reify the execution *context* into a first-class entity.

Not only the *behavior* but also the *structure* of classes can change. Fields can be added or removed, and the type of a field can change. As a matter of fact, in a subsequent version of the project[4], the developers added a

[3]See version 75 of the project.

[4]See version 82 of the project.

field `siteUrl` to the `WebServer` class. Unfortunately, the server is an object shared between multiple requests, and each service holds a reference back to the server. If the object structure is updated globally while different versions of the code run to serve requests, old versions of methods might access fields at the wrong index. While the problem for field addition can be solved easily by ensuring new fields are added at the end, we need to consider type changes as well. For instance, one could imagine that in the future newest versions will store the `siteUrl` as an `HttpUrl` rather than a `String`. Therefore, the general problem remains: *How can we ensure consistent access to objects whose structure (position or type of fields) has changed?*

Our solution to ensure consistent access is to keep one representation of the object per context and to synchronize the representations using bidirectional transformations. Once there is no reference any longer to a context, it is garbage-collected together with the corresponding representations of objects in that context.

3.2.2 Lifecycle of an Incremental Update

Let us consider the addition of the field `siteUrl` in the `WebServer` class in more detail. The following steps describe how an *incremental* update can be installed with Theseus, an implementation of our approach, while avoiding the problems presented above.

First, the application must be adapted so that we can "push" an update to the system and activate it. Here is how one would typically adapt an event-based server system, such as a web server.

0. *Preparation.* First, a global variable `latestContext` is added to track the latest execution context to be used. Second, an administrative page is added to the web server where an administrator can push updates to the system; the uploaded code will be loaded dynamically. Third, the main loop that listens to incoming requests is modified so that when a new thread is spawned to handle the incoming request, the latest execution context is used. Fourth, the thread that listens to incoming connections in a loop is modified so that it is restarted periodically in the latest context. Note that the listening socket can be passed to the new thread without ever being closed.

After these preliminary modifications the system can be started, and now it supports dynamic updates. The life cycle of an update would be as follows:

1. *Bootstrap.* After the system bootstraps, the application runs in a default context named the *Root* context. The global variable `latestContext` is initialized to refer to the *Root* context. At this stage only one context exists and the system is similar to a non-contextual system.

2. *Development.* During development, the field `siteUrl` is added to `WebServer` and other related changes are installed.

3. *Update preparation.* The developer creates a class called `UpdatedContext`, which specifies the variations in the program to be rolled out dynamically. This is done by implementing a bidirectional transformation that converts the program state between the *Root* context and the *Updated* context. Objects will be transformed one at a time. By default, the identity transformation is assumed, and only a custom transformation for the `WebServer` class is necessary in our case.

4. *Update push.* Using the administrative web interface, the developer uploads the class `UpdatedContext` as well as the other classes that will be required by the context. The application loads the code dynamically. It detects that one class is a context and instantiates it. Contexts are related to each other by a ancestor-successor relationship. The ancestor of the newly created context is the active context at the time of code loading. The global variable `latestContext` is updated to refer to the newly created instance of the *Updated* context.

5. *Update activation.* When a new incoming request is accepted, the application spawns a new thread to serve the request in the `latestContext` (which is now the *Updated* context) while existing threads terminate in the *Root* context.

6. *Incremental update.* When the web server is accessed in the *Updated* context for the first time, the new version of the class is dynamically loaded, and the instance is *migrated*. Migration is triggered when the object is accessed from a different context for the first time. In our case, this results in the fields `port` and `services` being copied, and the field `siteUrl` being initialized with a default value. Fields can be accessed safely from either the *Root* or *Updated* context, as each context has its own representation of the object. To ensure that the count of requests processed so far, `numRequests`, remains consistent in both contexts, bidirectional transformations between the representations are used. They are executed *lazily*: writing a new value in a field in one context only invalidates the representation of the object in the other context. The representation in the other context will be *synchronized* only when it is accessed again. Synchronization is performed lazily when changes happen to objects that have already been migrated.

7. *Garbage collection.* Eventually the listener thread is restarted, and all requests in the old context terminate. A context only holds weakly onto its ancestor so when no code runs in the old context any longer, the context is finalized. The finalization forces the migration of all

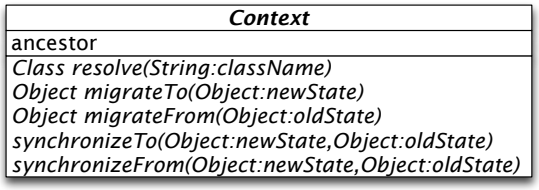


Figure 3.2: The `Context` class.

objects in the old context that have not been migrated yet. The old context and its object representations can then be garbage-collected. It must be noted that at the conceptual level, all objects in memory are migrated. In practice, only objects that are shared between contexts need to be migrated.

3.3 First-class Context

In our approach, a context is an object that represents a specific software version. The approach relies on a simple, yet fundamental, language change: the state of an object is contextual. We now describe the approach in more details. For the sake a clarity, we assume throughout the rest of this chapter that at most two contexts exist at a time, which we refer to as the "old" and "new" contexts. The model can be easily generalized to support any number of co-existing contexts, but the implementation would need to be revised since it takes advantage of this assumption.

3.3.1 User-defined Update Strategy

Contexts are first-class entities in our system. Programmers have complete control over the dynamic update of objects and classes. Contexts are ordinary instances of the class `Context`, shown in Figure 3.2. A context is responsible for maintaining the consistency of the representations of the objects belonging to it. Methods `Context.migrate{To|From}` and `Context.synchronize{To|From}` define the update strategy.

An object is initially *local* to the context it was created. When the object is first accessed in the "other" context (the context that isn't the one it was created in), the system triggers the *migration* of the object, after which the object is *shared* (see Figure 3.3). The migration creates the initial representation of the object in the "other" context. It is implemented in methods `Context.migrate{To|From}`. In our case, the migration of the web server from the old context to the new context would copy the existing fields *as is* and initialize the new field `siteUrl` with a predefined value (see Figure 3.4). By construction, either `migrateTo` or `migrateFrom` will be fired for

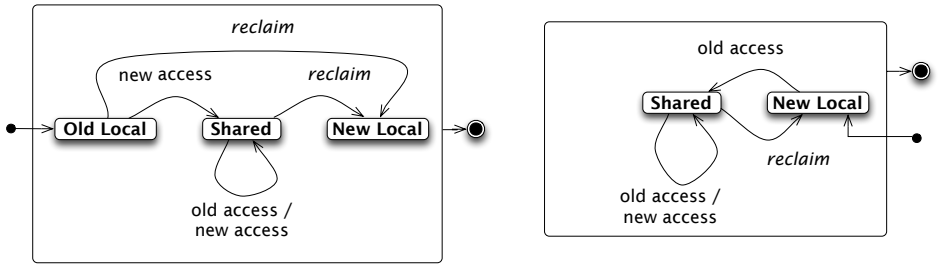


Figure 3.3: Objects are originally local to a context. Depending on its access patterns, the object might become shared between contexts. Eventually, the object is reclaimed when the update completes and the object is local again.

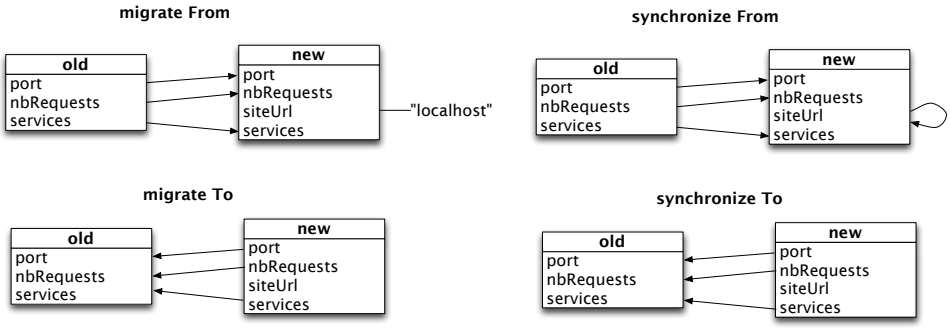


Figure 3.4: The effects of the various methods that class `Context` mandates. Note that the arrow means a field copy operation and the method always applies to the new context.

an object, depending on whether it was originally created in the old or new context.

Methods `Context.synchronize{To|From}` are responsible for subsequent updates of the state when the object is shared. In the case of our example, the field `siteUrl` must not be initialized again. Both methods `synchronize{To|From}` can be fired multiple times for a given object depending on its access patterns from the old and new contexts. In the rest of the chapter, we use the term "transformations" for both migrations and synchronizations.

Each context has an ancestor. Since the contexts are loaded dynamically in an unanticipated fashion the update strategy is encoded in the newest context and expressed in terms of its ancestor, never in terms of its successor: methods `Context.*From` use the old representation to update or create the new representation (new *from* old), and methods `Context.*To` use the new representation to create or update the old representation (old *to* new). The *Root* context is the only context that does not encode any transformation and has no ancestor.

3.3.2 Object Representations

Let us explain the semantics of our approach by considering how it could be implemented with a meta-circular interpreter. At the application level, an object has a unique identity and its state differs depending on the active context. An application thread will implicitly impact either the old or the new representation of an object, but cannot explicitly refer to either one. At the interpreter level, an application object is implemented by several representations. Transformations are reflective hooks that run at the interpreter level outside of any context. They can access simultaneously the old and new representations of an application object. From within a transformation, arbitrary messages cannot be sent to representations since the absence of context makes their behavior ill-defined. Representations must be manipulated with reflective state accesses that operate correctly. Listing 3.1 shows for instance the identity transformation.

```
Context>>synchronize: newState from: oldState
  newState instVarsDo: [ :idx |
    newState instVarAt: idx put:
      (oldState instVarAt: idx ) other
  ]
```

Listing 3.1: Identity Transformation

The message `other` returns the representation in the "other" context. Certain objects in the system are primitive. They have a unique state in the system. This is notably the case for contexts, scalar values (string, numbers, *etc.*), and semaphores. Primitive objects can be used from the application and from the interpreter (*i.e.*, within transformations). If two strings must be concatenated during a transformation, the following code works since strings are primitive:

```
| address domain |
address ← newState instVar: #address.
domain ← newState instVar: #domain.
oldState instVar: #email put: ( address , '@' , domain )
```

Listing 3.2: Concatenation of two fields

The inverse transformation would need to split the string into two parts. The method `split` returns an array, which isn't a primitive object. The manipulation of an array from within a transformation is possible with an explicit context switch which restores the existence of a context. Within a transformation, `self` represents the new context, and `self ancestor` the old context.

```
| address domain |
self ancestor do: [
  | email |
  email ← oldState instVar: #email.
  address ←  ( email split: '@'  ) first.
  domain ←  ( email split: '@'  ) second.
].
newState instVar: #address put: address.
newState instVar: #domain put: domain.
```

Listing 3.3: Splitting a field into two

3.3.3 First-class Classes

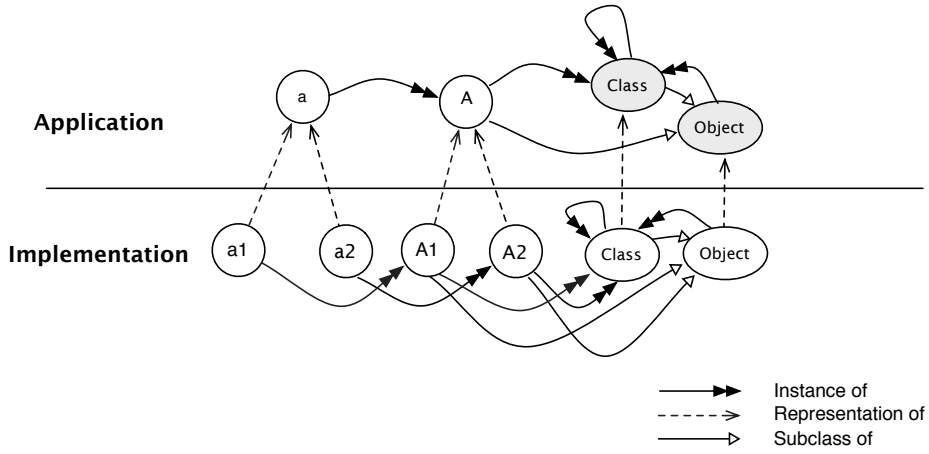


Figure 3.5: Object `a` is an instance of class `A`. Both `a` and `A` are contextual objects since classes are first-class. At the interpreter level, contextual objects have multiple representations. Classes `Object` and `Class` are primitive and have a unique representation.

Classes are also objects in our approach. At the application level, a contextual object `a` is an instance of the contextual class `A`. At the interpreter level, contextual objects and classes have multiple representations, as depicted in Figure 3.5. A representation of a contextual object is an instance of the representation of the corresponding contextual class. For example, the object representation `a1` is an instance of the class representation `A1`.

When an object is migrated, its original representation is passed as a parameter to `migrate{To|From}`. The method creates and returns a second representation for the given context. For instance, the migration of `a1`, which is an instance of `A1`, results in `a2`, which is an instance of `A2`. The class

can change only during migration. Indeed, methods `synchronize{From|to}` take as arguments two representations, but are not able to change the class they correspond to.

Classes are migrated similarly to regular objects. A specific representation of a class is passed as parameter to `migrate{To|From}`, which must return a second representation. For instance, the migration of `A1`, which is an instance of `Class`, returns `A2`, which is also an instance of `Class`. Note that `Class` is a primitive in the system. Names of classes are literals in the source that resolve at run-time to first-class classes. This binding is also contextual to support class renaming. Contexts are responsible for class name resolution and must implement the method `Context.resolve(String)`.

Similarly to reflective state accesses with `instVar:` and `instVar:put`, the method `class` can be used from within transformations to obtain the class of a representation. Methods `Context.migrate{To|From}` and `Context.synchronize{To|From}` can thus apply custom transformations for specific objects and default to the identity transformation for objects without structural changes. The code below applies a custom transformation to instances of the `Contact` class:

```
UpdatedContext>>synchronize: newState to: oldState
  ( newState class instVar: #name ) = #Contact ifTrue: [
    | address domain |
    address ← newState instVar: #address.
    domain ← newState instVar: #domain.
    oldState instVar: #email put: ( address, '@' ,domain ).
    ↑ self.
  ]
  ↑ super synchronize: newState to: oldState.
```

Listing 3.4: Custom transformation

3.3.4 Threads and Contexts

A thread can have one *active* context at a time. A predefined context exists, called the *Root*, which is the default context after startup. The runtime must be extended with a mechanism to query the active context, and to switch the current context. When a thread is forked, it will inherit the context of its parent thread. For convenience, a thread can be forked and change its context right after. This way, code executed by the thread runs entirely in the given target context.

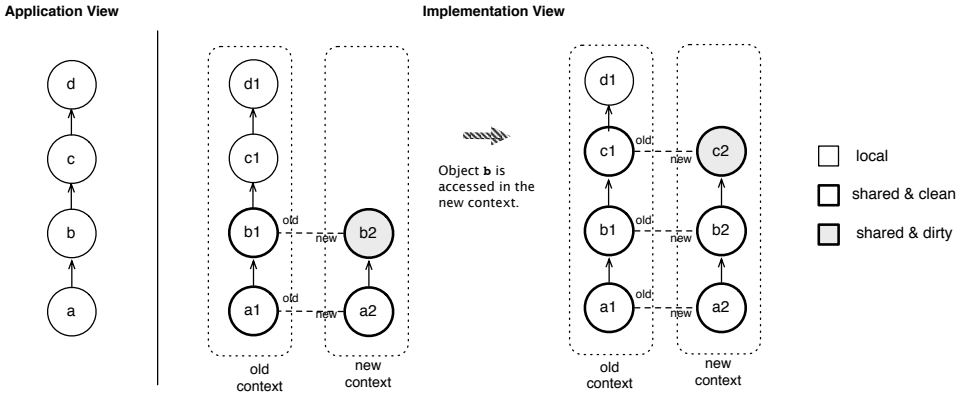


Figure 3.6: From the application view, four contextual objects `a,b,c,d` and form a list. From the implementation view, shared objects have one representation per context, which can be either "clean" or "dirty". Objects are migrated lazily. When object `b` is accessed in the new context for the first time, the representation `b2` is synchronized. Since `b` refers to `c`, this triggers the migration of `c` and the representation `c2` is created, originally considered "dirty". An access to `c` in the new context would create the representation `d2`, *etc.*, Dashed lines represent relationships visible only to the implementation, not the application.

3.4 Implementation

We have implemented our approach in Pharo Smalltalk[5] using bytecode transformation to avoid changes to the virtual machine. A unique aspect of our implementation is that it does not rely on proxies or wrappers, which do not properly support self-reference, do not support adding or changing public method signature, and break reflection [137, 127].

During an incremental update, a contextual object corresponds concretely to two objects in memory, one per context. To maintain the illusion that the old and new representations of an object have the same identity, we adapt the references when necessary: for instance, if `b1` is assigned to a field of `a1` in the old context, this results in `b2` being assigned to the corresponding field of `a2` in the newest context. Figure 5.12 depicts such a setting.

Objects are migrated lazily, and can be either flagged as "clean" or "dirty". Dirty objects are out-of-date, and need to be synchronized upon the next access. Figure 5.12 shows the effect of an access to the dirty representation `b2`, which triggers the migration of the representation `c2` it references directly. After the synchronization, the two representations `b1` and `b2` of object `b` are clean. Subsequent writes to either representation would

[5] `http://www.pharo-project.org`

```
if self is dirty and global then
    // synchronization
    for field ← fields of self do
        // migration
        if self.other.field is local then
            migrate( self.other.field) ;
        end
        // synchronization
        self.field = self.other.field.other;
    end
    mark self as clean;
end
read field self.name;
```

Algorithm 1: Pseudo-code for state reads in the case of the identity transformation

however result in the other one to be flagged as dirty. In the case of Figure 5.12, if `b2` is modified, `b1` would be marked as dirty.

We use bytecode rewriting to alter accesses to state and the way classes are resolved. Concretely, an extra check is added before each state read and state write to determine whether the object is shared between contexts. For state reads, if the object is shared and "dirty" it is first synchronized and then marked as "clean". Algorithm 1 shows in pseudo code what happens upon state reads in case of the identity transformation. For state writes, the new value is written and the other representation is invalidated and flagged as "dirty". We maintain the correspondence between representations using synthetic fields added during the program transformation.

We must ensure that all objects reachable from the old context have been migrated and are up-to-date before old representations are garbage-collected. Therefore, when the old context becomes eligible for garbage collection, the system traverses the object graph and forces the migration or synchronization of objects if necessary. This corresponds to the transitions labelled "reclaim" in Figure 3.3. In the case of Figure 5.12, the system would force the migration of `d1` before garbage collection. If the graph of reachable objects is big, the traversal can take long but can be conducted in the background with low priority.

3.4.1 Concurrency and Garbage Collection

We assume that the system has a coherent memory, as is the case for the Cog VM for Pharo. With a coherent memory, state reads and writes are atomic and side-effects are immediately visibile to all threads.

Since our implementation instruments state accesses with additional

logic, it does not automatically preserve thread safety. A trivial way to preserve it would be to acquire a per-object lock for each state access. Migrations and synchronizations of an object would therefore never conflict. This would however lead to an unacceptable performance penalty.

```
if self is local then
  read field self.name;
else
  acquire self.lock do
    if self is not local then
      if self.name is dirty then
        // migration
        if self.other.name is local then
          acquire self.other.name.lock do
            if self.other.name is local then
              migrate( self.other.name) ;
            end
          release
        end
        // synchronization
        self.name = self.other.name.other;
        mark self.name as clean;
      end
    end
    read field self.name;
  release
end
```

Algorithm 2: Pseudo-code for state reads using a per-field dirty flag and locks for mutual exclusion.

Instead, we use a relaxed locking scheme where state accesses to objects that are local to a context do not require the acquisition of a lock. This scheme relies on the use of per-field dirty flags instead of per-object dirty flags. It also assumes that the original program is correctly synchronized and that reads and writes to a given field never happen concurrently. This should be the case for any program, since developers should never rely on the atomicity of state reads and writes even if memory is coherent.

Algorithm 2 adapts the pseudo code of the previous section to reflect this strategy. It assumes that two representations of a contextual object have the same lock, *i.e.*, self.lock = self.other.lock. An object is originally local to the context that created it. The object might later become shared between contexts. The object does not become shared at the moment it is accessed from another context, but when a reference to it is obtained (lines 7-13 in 2). For instance, in Figure 5.12, the old state `b1` is migrated when

a new thread accesses `a2.f` and obtains a reference to `b2`. After migration, the migrated state is considered dirty. This corresponds to the mechanism of lazy propagation explained in the previous section.

Before we discuss the validity of our strategy, let us introduce some terminology: we use the term *old threads* for threads running in the old context, and *new threads* for threads running in the new context. Similarly, we use the terminology *old local object* and *new local object* for local objects created originally in the old or new context. The synthetic thread that forces the update of the reachable objects before garbage collection is referred to as the *background thread*. We assume that before it forces the update of an object, it acquires first its lock. As stated previously, concurrent reads and writes to the same field are excluded, since we assume the program is correctly synchronized. Let us use the variables f and g to refer to distinct fields of an object.

We can informally list all possible cases and show that the strategy effectively prevents *lost updates*:

Let us consider first the case where an old thread reads field f of an old local object. We consider three sub-cases: 1) Concurrent reads and writes to g by old threads. This doesn't lead to conflicts since fields are independent. 2) Concurrent reads and writes to g by new threads. This will trigger the migration of the object. After the migration, all fields of the new representation are considered dirty, except g. If it was a read, g holds the value of the old context; if it was a write it holds the updated value and the field in the old context is marked as dirty. In both cases, there is no conflict. 3) Forced updates by the background thread. By definition, this thread runs when there are no old threads any longer, so this case is not possible.

Let us consider now the case where an old thread writes into field f of an old local object. We consider three sub-cases: 1) Concurrent reads and writes to g by old threads. This doesn't lead to conflicts since fields are independent. 2) Concurrent reads and writes to g by new threads. This will trigger the migration of the object. If it is a read, all fields of the new representation expect g will be dirty. The update to f is not lost and will be reflected if it is later accessed in the new context. The situation is identical for a write. 3) Forced updates by the background thread. By definition, this thread runs when there are no old threads any longer, so this case is not possible.

Let us now consider the case where old and new threads access an object shared between contexts. We consider two sub-cases: 1) The background thread isn't running. If the background thread is not running, there is no way for a shared object to become local again. All accesses will be mutually exclusive. 2) The background thread is running. If only new threads exist, the background thread can force the update of an object after which it will be local again. Lock acquisitions follow an idiom similar to double-checked locking [13] where the condition *is local*/*is not local* is tested twice.

If an object transitions from context-shared to context-local when a thread awaits for a lock, this change will be detected when the lock is acquired and won't lead to conflicts.

Let us now consider the case where a new thread reads field f of a new local object. We consider three sub-cases: 1) Concurrent reads and writes to g by new threads. This doesn't lead to conflicts since fields are independent. 2) Concurrent reads and writes to g by old threads. This will trigger the migration of the object. After the migration, all fields of the old representation are considered dirty, except g. If it was a read, g holds the value of the new context; if it was a write it holds the updated value and the field in the new context is marked as dirty. In both cases, there is no conflict. 3) Forced updates by the background thread. By definition, this thread does not mutate any data so it cannot lead to lost updates.

Let us now consider the case where a new thread writes into field f of a new local object. We consider three sub-cases: 1) Concurrent reads and writes to g by new threads. This doesn't lead to conflicts since fields are independent. 2) Concurrent reads and writes to g by old threads. This will trigger the migration of the object. If it is a read, all fields of the old representation expect g will be dirty. The update to f is not lost and will be reflected if it is later accessed in the old context. The situation is identical for a write. 3) Forced updates by the background thread. By definition, the background thread skips new local objects since they are by definition up-to-date. No update can be lost.

3.4.2 State Relocation

Transformations can be more complex than one-to-one mappings. For instance, instead of keeping track of the number of requests in `numRequests` using a primitive numeric type, the developer might introduce and use a class `Counter` for better encapsulation[6]. During the transformation, the actual count would be "relocated" from the web server object to the counter object that is now used. However, in this case, when the counter is incremented, the old representation of the web server with field `numRequests` needs to be invalidated. So far we have assumed that a write would invalidate only the representation of the object written to, which is not the case any longer. To support such transformations, the full interface enables custom invalidation on a per-field basis with `Context.invalidate{To|From}-(Object oldState,Object newState,String field)`.

[6]This would be the refactoring "Replace Data Value with Object". See `http://www.refactoring.com`

3.4.3 Special Language Constructs

The implementation described so far assumes a uniform language where state is accessible solely via instance variables, and object instantiation is realized with message sends to classes. Pharo Smalltalk is very close to this ideal language, with a few peculiarities nevertheless.

Closures Closures are first-class in Smalltak. Closures are instances of `BlockClosure` and encode offsets of bytecode in the `CompiledMethod` they belong to. They are treated analogously to other objects. After migration, they reference the newest version of the corresponding `CompiledMethod`. Offsets are copied as is, assuming the same syntactic position of the closure in both versions of the sources. If this assumptions turns out wrong, it would be possible to write a custom transformation that corrects offsets during transformation.

Primitive Methods Our approach cannot intercept state changes from primitive methods. Primitive methods that operate on contextual objects must be adapted to work according to our model. Since primitive methods are specified with the `primitive` pragma, they can be renamed, and a wrapper method working correctly can be provided with the original name. The majority of primitive methods operate on primitive objects though, and do not need to be modified.

Cloning One special primitive method used pervasively is `copy` (and its variant `copyFrom:`). The correctly working wrapper must first ensure that if the object is shared, it is fully up-to-date. It can then be copied to produce a local clone.

Syntactic Sugar Pharo Smalltak has syntactic sugar for arrays `{...}`, literal arrays `#(...)`, and class variables that are visible to instances of a class and the class itself. These constructs are first desugared, then processed through our regular transformation.

Hashcode Comparing objects based on their identity works correctly with our approach: in Figure 5.12, code comparing `a == b` would consistently compare either `a1` with `b1`, or `a2` with `b2` depending on the context. The hash code of `a` would however be different depending on the context, breaking notably the collection classes that use hash codes to position elements in internal data structures. To ensure that different versions of an object have the same hash code, we keep a unique identity in an additional synthetic field.

Continuations The stack is a sequence of activation frames. Continuations can capture context switches or ignore them. In the first case, the continuation would be a primitive object that would not be modified when exchanged across contexts; calling a continuation would restore the corresponding context switches along the way. In the second case, the continuation would be contextual and its corresponding activation frames would be adjusted when exchanged across contexts. Activation frames could be migrated similarly to closures, assuming that methods on the stack have not been modified between versions. If activated methods would have been modified, the adjustment of the continuation would require a mapping that might be difficult to achieve, as shown by Makris and Bazzi [104]. We have not implemented support for continuations.

Exceptions In our approach, threads run consistently in one context (see subsection 3.3.4). An exception is an object that is thrown and caught within the same context. If a thread can switch temporarily its context, objects flowing in and out of the boundary of the temporary context must be migrated accordingly, which includes exceptions. The migration of the stack frames the exception refers to would lead in this case to issues similar to those with continuations.

3.4.4 Further Details

We used a custom compiler to rewrite the bytecode of contextual classes. We instrumented copies of kernel classes (array, dictionary, etc.) to avoid metacircularity issues during development. Also, the `Object` class cannot be extended with the necessary information for our model (namely the synthetic fields to related versions with each other) and we instead extended the subclasses of `Object`. Primitive classes (see subsection 3.3.2) do not require any bytecode rewriting. This design requires methods of primitive classes like `String>>split` to be trapped: invoked from the environment, an instance of the uninstrumented collection class is returned; invoked from our application, an instance of the instrumented collection class is returned. The complete set of such methods has not been identified and trapped, which can lead to minor bugs.

The active context is stored in a thread-local variable and we add a new method to fork a closure in a specific context, *e.g.*, `[ ... ] forkInContext: aContext`. When a closure is forked, it becomes a shared contextual object and is migrated. As the program proceeds, objects referenced by the closure are migrated lazily when accessed. Contexts hold only weak references to their ancestor and implement the method `Object>>#finalize`, which forces the migration of all reachable objects before the context becomes eligible for garbage collection.

3.5 Validation

3.5.1 Evolution

We conducted a first experiment whose goal was to assess whether our model could support long-term evolution, that is, whether it could sustain successive updates. We considered the small web server of Section 5.2, which despite its simplicity cannot be updated easily with global updates. The sever has a simple architecture and is comprised of 7 classes: `WebServer`, `WebRequest`, `WebResponse`, `WebMessage`, `WebUtils`, `WebCookie`, `WebSocket`. We selected the 4 last versions with effective changes: version 75 introduced chunked data transfer, version 78 fixed a bug in the encoding of URL, version 82 introduced `siteUrl`, and version 84 fixed a bug in MIME multipart support.

To restart periodically, the listener thread executing `WebServer>>runListener` was adapted to accept incoming connections only during 1 second, after which a new listener thread is restarted (see Listing 3.5). It required only a couple of lines to be changed. The method `WebServer>>asyncHandleConnectionFrom:` spawns a new thread per connection. It was simply modified to spawn the thread in the latest context.

```
WebServer>>runListener
  | connectionSocket startTime |
  startTime ← Time now.
  [ (Time millisecondsSince: startTime)<1000 ] whileTrue: [
    connectionSocket ← listenerSocket waitForAcceptFor: 5.
    self asyncHandleConnectionFrom: connectionSocket.
  ].
  [ self runListener ] forkInContext: CurrentContext lastest
```

Listing 3.5: Modified listener thread. It omits error handling code for readability.

We implemented the hello world service of Section 5.2. Only one update required us to write a custom transformation: the one that introduced the `siteUrl` field, which we initialized to a default value.

```
UpdatedContext>>migrateFrom: oldState
  ( oldState class instVar: #name ) = #WebServer ifTrue: [
    | newState |
    newState ← oldState class other new.
    newState instVar: #port put: (oldState instVar: #port).
    newState instVar: #nbReq
             put: (oldState instVar: #nbReq).
    newState instVar: #services
             put: (oldState instVar: #services).
```

```
    newState instVar: #siteUrl put: 'http//localhost'.
    ↑ newState.
  ]
  ↑ super migrateFrom: oldState.
```

Listing 3.6: Migration that initializes field `siteUrl`

We ran the 4 successive dynamic updates, and verified that once it was no longer used, the old context would be garbage-collected. Since web browsers keep one connection alive for multiple requests, we could observe different versions of the services in two browsers.

3.5.2 Run-time Characteristics

For the second experiment, we picked a typical technology stack with two well-known production projects: the Swazoo[7] web server and the Seaside[8] web framework. This corresponds to several hundred classes. Similarly to the previous experiment, the web server was adapted to periodically restart its listener thread and process requests in the latest context. We were interested in the run-time characteristics and in assessing (1) whether our assumptions about object sharing hold, (2) what is the memory overhead, and (3) what is the time overhead.

As a case study, we considered the counter component example that comes with the Seaside distribution. During maintenance, only few classes change. Most objects are migrated with the identity transformation, and only certain objects require custom transformations. The exact nature of the transformation is not significant. Therefore, for the sake of simplicity, we artificially updated the system and used the identity transformation for all objects.

We were interested to assess the overhead of our implementation during the five following phases:

(i) with only the old context when no object is shared,

(ii) during the incremental update when objects are shared and migrated lazily,

(iii) after objects have been migrated but are still considered shared,

(iv) when the old context is finalized and the system forces the migration/update of all objects reachable in the old context, and

(v) after the old context has been garbage-collected and the system runs as in (i).

[7] `http://www.swazoo.org`
[8] `http://www.seaside.st`

Table 3.1: Read/write ratios and heap size per phase. The star (*) indicates phases with increasing memory consumption, for which we considered the peak.

	i	ii* / iii	iv*	v
# reads	59'908	60'952	62'021	61'568
# shared reads	0	29'278	29'666	0
% shared	0.00	46.80	46.63	0.00
# writes	3'773	3'749	3'777	3'745
# shared writes	0	174	174	0
% shared	0.00	4.64	4.61	0.00
# objects	48'905	49'390	49'829	49'961
# shared objects	0	1'118	36'591	0
% shared	0.00	2.26	73.43	0.00
Heap size	1'063'492	1'078'494	1'087'065	1'085'704
Shared heap size	0	30'160	858'695	0
% shared	0.00	2.80	78.99	0.00

Object Sharing and Memory Overhead

We studied object sharing and memory overhead by manually incrementing the counter from one browser session. We tracked the number of reads and writes to objects shared between contexts, and to objects local to a context. To account for "sharable" objects in the heap, we tracked all objects reachable from classes and all objects reachable from variables captured by forked closures. To account for the "local" objects in the heap, we tracked all objects that were receivers or return values of message sends.

We measured first the memory after 5 increments of the counter, before any update. This corresponds to phase (i). We reset the tracking, installed the update, incremented the counter 5 more times and then measured memory again. This corresponds to phase (iii) which itself corresponds to the peak of memory of phase (ii). We then measured the memory at the peak of consumption for phase (iv), when the complete graph of objects has been traversed but no object has been reclaimed yet. Finally, we reset the tracking, incremented the counter 5 times, and measured again the memory. This corresponds to phase (v), after memory has been reclaimed.

Since our implementation uses a copy of the kernel classes, we observe only the effects of our application in isolation from the Smalltalk environment. We track receivers and return values at call sites, which correctly

considers primitive objects whose classes haven't been instrumented (see Section 5.7). Since transient objects are tracked, they will not be garbage-collected. Our approach does not measure the effective size of the heap, but estimates the upper bound. This upper bound is thus the sum of the sizes of the individual objects. The size of an object was computed with a known algorithm that considers its structure[9], then it was doubled in the case the object is shared. When an object is shared between contexts, we need to keep for each object representation the following information: a semaphore (see subsection 3.4.1), a unique identifier (see subsection 3.4.3), dirty flags (see subsection 3.4.1), and one reference to the "other" representation (see subsection 3.4.1). Table 3.1 does not account for this overhead that is very implementation-specific.

We make the following observations from the results presented in Table 3.1:

- There are an order of magnitude more reads than writes.
- The percentage of objects effectively shared (2.26%) during phase (iii) is smaller than the percentage of objects that are shareable (73.43%) and thus migrated in phase (iv).
- Shared reads represent 46% of all reads. Shared writes are mostly neglectable (4.6% in phases (iii) and (iv)). This supports the idea that an overhead for accesses to shared objects is tolerable.
- The heap is composed mostly of shareable objects (73.43%, phase (iv)) and only few transient, local objects. This might be related to the stateful nature of the Seaside framework. This means however that our approach might entail an important memory overhead in the case study.

Time Overhead

We studied the time overhead in a multi-threaded scenario. We conducted load tests[10] with the following setting: 10 concurrent visitors connect to the website and start a web session each. They go to the counter page. From there, they keep incrementing the counter as quickly as possible. The system had been modified to install updates per request. The server and the load testing tool ran both on the same machine (a 2.3 GHz Intel Core i7 laptop), to minimize the effect of the network. We ran the server on the CogVM 6.0.

The results for the time overhead are presented in Figure 3.7, which shows the quantiles in response time for the various phases. The dashed

[9] See Seaside's internal memory profiler, and the method `WAMemoryItem>>sizeOfObject`

[10] `http://jmeter.apache.org`

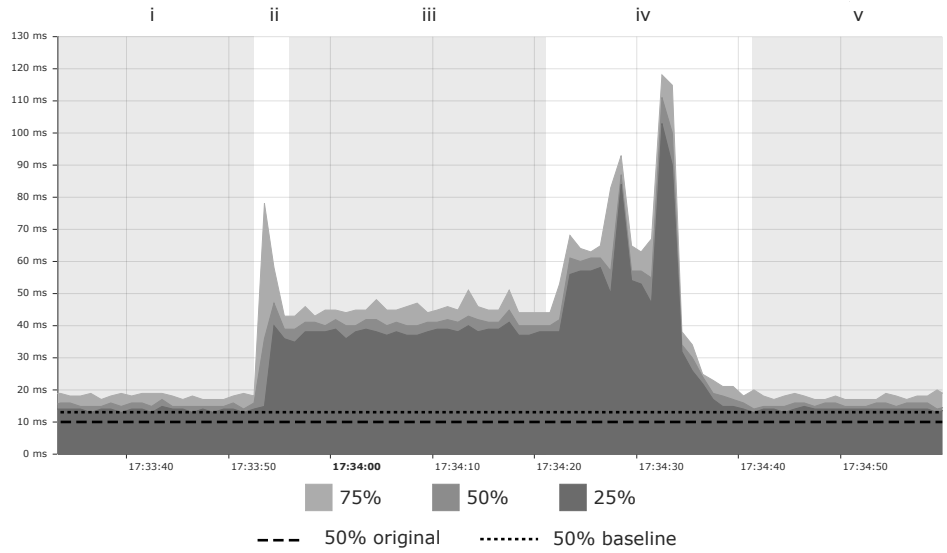


Figure 3.7: Response time quantiles for a typical run of the load test. Quantiles are computed using measures within a window of 1 second. The various phases (i) to (v) of the dynamic update are highlighted with different background colors.

and dotted lines represent the median of the response time for the original system, and the baseline system. The baseline system is the original system with one level of indirection for state accesses that go through synthetic accessors. This helps one to compare the overhead of the approach instead of the cost of the indirection mechanism, which could be aggressively inlined if necessary. We make the following observations from the results of Figure 3.7:

- The sole use of synthetic getters and setters (without additional logic) entails a 30% overhead, as can be seen in the difference between the original system and the baseline (10ms vs 13ms). Other virtual machines might be able to aggressively optimize this case. The overhead between the baseline and phases (i) and (iv) is 15% (13ms vs. 15ms), supporting the idea that our approach is attractive at "steady state".
- The five phases are clearly visible. Phase (ii) corresponds to a peak when the majority of objects are first migrated. Phase (iii) corresponds to a plateau where performance is degraded due to lock acquisition, checks for "dirtiness" of object representations, and resulting synchronizations. Phase (iv) shows further degradation when the system operates in background the synchronization of all objects reachable in the new context. Eventually, the system reverts back to its original performance (iv).

- The performance degradation between phases (i) and (iii) is of factor 2.6, which is tolerable for a short period of time.
- Giving a lower priority to the background task during phase (iv) would make the update take longer to complete, but lower the performance degradation.

Overall, the benchmark shows the expected profile, and suggest our approach can be made practical for realistic production systems.

3.6 Discussion

We discuss in this section the applicability of our approach and its implementation from three different perspectives:

3.6.1 Portability

Our approach and implementation technique are portable to other object-oriented languages. The approach can notably be ported to a statically typed language. Particular language constructs of the target language might however pose obstacles, specifically those we listed in subsection 3.4.3. Experiments porting the approach to Java showed it is feasible [170] with the following language constructs as obstacles: constructors, nested classes, arrays, concurrency control in the language semantics. Our locking scheme resembles the double-checked locking [13] idiom which is correct only for systems with a coherent memory. Java has for instance a relaxed memory model (JSR-133) where the double-checked locking is correct only if the field that is tested is declared `volatile` [13]. Our locking scheme would need to be revised accordingly for a full port to Java.

3.6.2 Performance

A drawback of our implementation is that shared objects need two representations, even if they are structurally identical and will use the identity transformation. Wrappers would make it possible to keep only one representation in such cases, but pose problems of self-reference, do not support adding or changing method signatures, and break reflection [137, 127, 133]. The benefit of our implementation is that it avoids such problems. Our implementation entails a performance degradation due to internal locking when shared objects are accessed. These locks are however typically uncontended. Their impact on performance depends on how well uncontended locking is optimized by the virtual machine. Two directions could be explored to reduce locking: 1) synchronize groups of fields at precise locations, instead of each individual field (*e.g.*, synchronize all fields

a method uses at once at the beginning and end of the method), and 2) emulate safepoints and run the pre-garbage collection thread exclusively from application threads, making migration the only operation that requires locking.

3.6.3 Development Effort

The impact on development is small. Developers must modify the application's thread management to make the application updatable. Essentially, long-running loops must be modified to restart periodically and requests must be processed in the desired context. Threads executing long-running loops can usually be aborted and restarted without problem. Adapting request dispatching is also usually easy and entails only few local changes. In our first experiment, these changes represent about 10 lines of code. The use of thread pools might complicate the adaptation of the application in which case the pool must be adapted to renew its threads periodically [170]. Reflective code doesn't need to be adapted since object representations are really instances of their respective classes and reflective code works correctly.

Writing transformations to transfer the application state requires additional effort. Compared to other dynamic update mechanisms, there must exist a state mapping only for shared entities (not all entities), but the mapping must be bidirectional (not unidirectional). Transformations are usually simple (field addition, renaming, suppression, type conversion) and complex transformations are very occasional [153, 105, 24]. In our first experiment, only one update required a transformation, which was simple. Recent works showed that static [132] and dynamic analysis [103] can generate most of the needed transformations automatically. It would be interesting to assess whether we can extend these techniques for bidirectional transformations as well.

3.6.4 Type Safety

Our approach guarantees type safety assuming that the state transformation is correct. State transformations are usually simple. Typically, a state transformation converts the representation of objects whose classes have changed. However, the object graph expected by the old and new versions might be different, even without class changes. Let us imagine a program that manipulates a list. The list in the old version might be composed of integers, and of strings in the new version. No classes have changed. Yet, the state transformer must transform the list of integers to a list of strings for the update to succeed. Moreover, it must not convert all lists of integers, but only specific lists of integers in the heap. Verifying the correctness of the state transformation in such cases is an open challenge.

3.7 Related Work

Our approach is closely related to two main bodies of work presented in Chapter 2: techniques that enable class redefinitions and techniques that support first-class layers of behaviors. We now compare our approach in more detail with respect to these two bodies of work.

3.7.1 Class Redefinition

The main challenge of dynamic updates to reconcile safety and practicality. Systems that support immediate code changes are very practical but subject to limitations or safety issues. Dynamic languages, including Smalltalk, belong to this category. If an object attempts to invoke a method that was suppressed, an error is raised. Several approaches of this kind have also been devised for Java [52, 127, 86, 72, 33, 174, 138], with various levels of flexibility (a good comparison can be found in [72]). To be type-safe, HotSwap [52] imposes restrictions and only method bodies can be updated. The most recent approaches (JavAdaptor [138], DCEVM [174], Javaleon [72]) overcame most of these restrictions, and provide a similar flexibility as dynamic languages. Some approaches use bytecode transformation (JavAdaptor [138], Javaleon [72]) or custom virtual machines (DCEVM [174]).

Systems that impose constraints on the timing of updates are safe, but less practical since temporal *update points* must first be identified. Such systems have been devised for C (Ginseng [79, 118], UpStare [104], Kitsune [76]), and Java (JVolve [153], DVM [105]). Update points might be hard to reach, especially in multi-threaded applications [116, 153], and this compromises the timely installation of updates. Our approach entails the identification of *context switch* points, but relaxes the need for threads to reach the points simultaneously.

Some mechanisms diverge from a global update and enable different versions of the code or entities to coexist. In the most simple scheme, old entities are simply not migrated at all and only new entities use the updated type definition [81], or this burden might be left to the developer who must request the migration explicitly [66]. The granularity of the update for such approaches is the object; it is hard to guarantee *version consistency* and to ensure that mutually compatible versions of objects will always be used. When leveraged, transactions [24, 133] provide version consistency but impede mutations of shared entities. Contexts enable mutations of shared entities and can be long-lived, thanks to the use of bidirectional transformations. With asynchronous communication between objects, the update of an object can wait until dependent objects have been upgraded in order to remain type-safe [85].

To the best of our knowledge, only three approaches rely on bidirectional transformations to ease dynamic updates. POLUS is a dynamic up-

dating system for C [34] which maintains coherence between versions by running synchronizations on writes. We synchronize lazily on read, operate at the level of objects, and take garbage collection into account. Duggan [55] formalized a type system that adapts objects back and forth: when the run-time version tag of an object doesn't match the version expected statically, the system converts the object with an adapter. We do not rely on static typing but on dynamic scoping with first-class contexts, we address garbage collection, concurrency, and provide a working implementation. Oracle enables a table to have two versions that are kept consistent thanks to bidirectional "cross-edition triggers" [37].

A common technique to achieve hot updates is to use redundant hardware [79], possibly using "session affinity" to ensure that the traffic of a given client is always routed to the same server. Our approach is more lightweight and enables the migration of the state shared across contexts, notably persistent objects. Also, an advantage of being reflective is that the software can "patch itself" as soon as patches become available.

3.7.2 Layers

Context-oriented programming [80] enables fine-grained variations based on dynamic attributes, *e.g.*, dynamically activated "layers". The existing approaches [42, 128, 7, 143, 165] focus mainly on behavioral changes. Certain approaches support contextual state in the form of dynamic variables [165]. Tanter explored possibilities to contextualize the application state with contextual values [155], which generalize dynamic variables. However, none of these approaches address changing the structure and state of objects as is necessary for dynamic updates.

3.7.3 Additional Related Work

Schema evolution addresses the update of persistent object stores, which closely relates to dynamic updates. To cope with the volume of data, migrations should happen lazily. To be type-safe, objects should be migrated in a valid order (*e.g.*, points of a rectangle must be migrated before the rectangle itself) [24, 133]. Our approach migrates objects lazily, and avoids the problem of ordering by keeping both versions as long as necessary.

How to keep two corresponding data structures synchronized is related both to the view-update problem [88] and lenses [22, 62, 82, 166]. We need in our approach to define a pair of transformations to map the source to the view, and the view to the source. Lenses are bidirectional programs that specify both a view definition and update policy. Lenses can be state-based or operation-based. State-based lenses synchronize structures as a whole, without knowing where the changes occurred, whereas operation-based lenses propagate local changes (or edits). Using lenses to express

transformations would make their expression more compact and less error-prone. Since we synchronize lazily, edits are lost and we would need to use state-based lenses.

3.8 Conclusions

Existing approaches to dynamically update software systems entail trade-offs in terms of safety, practicality, and timeliness. We have demonstrated that using first-class contexts that implement active variability can overcome this challenge and enable run-time evolution.

During an incremental update, clients might see different versions of the system, which avoids the need for the system to reach a quiescent, global update point. We have implemented our approach in a dynamic language with a strong memory model and conducted experiments on two existing web servers.

The approach improves the support of two modularity principles presented in Chapter 1:

- *Encapsulation.* A context represents a whole software version. It encapsulates all the changes from the previous version, including the mapping of the state.
- *Late Binding.* The behavior is dispatched dynamically based on the class of the object and the current version of the software that is active. Accesses to state might trigger state transformations.

We can draw the following conclusions about the practicality of the approach:

- *Flexibility.* The explicit context enables the update scheme to be customized to the nature of the application and provides a convenient way to trigger the garbage collection of old data, since the context is reclaimed like any other object once it isn't used any longer.
- *Effort.* Only few modifications to the original web servers were required to make them updatable, and simple transformations can be easily expressed in our approach.
- *Performance.* Results indicate that only a fraction of accesses concern objects shared between contexts which makes the cost of bidirectional transformations tolerable.

The main drawback of the approach is that is it relatively heavyweight to introduce minor changes. If the changes are known to be type-preserving and state-preserving, the variations could be scoped to individual threads

without needing the update to synchronize any state nor eventually stabilize. In the next chapter, we present building blocks to implement such kind of run-time variability in a lightweight and flexible manner.

4

Contextualizing Behavior with Delegation Proxies

4.1 Introduction

When an application feature is encapsulated into an object or an aggregate, it can be changed at run time by replacing the object or aggregate. It is however very common for application features to crosscut multiple classes and objects. This is typically true for features implementing non-functional concerns, *e.g.*, tracing [91] , but also for features implementing pervasive business rules, *e.g.*, game logic [9]. Modifying such behaviors at run time is thus challenging, even if the change preserves type signatures and does not alter object layouts.

The use of aspects is the de-facto solution to encapsulate crosscutting code. Only few aspect frameworks support dynamic aspects, though. With dynamic aspects, the activation of an aspect is global. In multithreaded systems, it has the consequence that the logic of a thread could change at arbitrary point in time. This could compromise the consistency of the application. As was illustrated in the previous chapter (Chapter 3), implementing active variability is a natural solution to this problem.

We show in this chapter how minor changes to the way dynamic proxies operate make them suitable building blocks to implement active variability. By default, dynamic proxies enable *homogenous* variations [6]. In that case, the same variation is applied in multiple methods. If necessary, *heterogenous* variations can be implemented on top of homogenous variations. In that case, different methods have different variations. Unlike contextual layers, which enable anticipated heterogenous variations, delega-

tion proxies enable unanticipated variations since they rely only on objects and classes that can be loaded dynamically.

The key ideas behind delegation proxies are the following: 1) rebind self-references by using delegation [99] instead of forwarding, 2) intercept object instantiations, and 3) intercept state accesses, including accesses to captured variables in closures. With these changes, it becomes possible to implement proxies that *propagate* to all objects accessed during the execution. All objects accessed during the evaluation of an expression, *i.e.*, the dynamic extent, are thus consistently represented with proxies.

In contrast to active contexts (Chapter 3), delegation proxies do not operate at the level of classes, but at the level of methods. Also, they do not synchronize state and so do not need that the update eventually stabilizes in one unique context; multiple contexts can live forever. They are useful complements, not competitors, to active contexts.

Delegation proxies have several positive properties. First, delegation proxies do not lead to *meta-regressions*, as is the case with traditional reflective architectures [36, 50] or aspects [157]. The behavior of a delegation proxy is defined by a separate *handler* object. This ensures that the base and meta-levels do not conflict. When an operation is applied to a proxy, the proxy reifies the operation and invokes instead a method of the handler. Methods in handlers are referred as *traps* [162]. Second, variations expressed with delegation proxies *compose*, similarly to aspects. For instance, profiling and tracing can be implemented with aspects and delegation proxies without interferences between the two concerns. Third, delegation proxies naturally support *partial reflection* [158]. Only the objects effectively accessed in the dynamic extent of an execution involving a proxy pay a performance overhead; all other objects in the system remain unaffected, including the target.

This chapter is organized as follows. In Section 4.2 we present delegation proxies and in Section 4.3 a catalog of examples; in Section 5.6 we formalize their semantics and in Section 5.7 we report on our implementation in Smalltalk.

4.2 Delegation Proxies

We now describe how delegation proxies work and exemplify them with a consistent implementation of tracing. Let us consider the extension method `Integer>>fib` which computes the Fibonacci value of a number using recursion:

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

Listing 4.1: Fibonacci computation

The computation of the Fibonacci value of 2 should produce the following trace: `2 fib, 2 <, false ifTrue:, 2 -, 1 fib, 1 <, true ifTrue:, 2 -, 0 fib, 0 <, true ifTrue:, 1 +` (Booleans are objects in Smalltalk).

Before we can trace the evaluation, we must obtain a tracing proxy of an object. We add method `Object>>tracing` after which we can trace the computation with `2 tracing fib`.

```
Object>> tracing
  | handler |
  handler ← TracingHandler target: self.
  ↑ Proxy handler: handler.
```

Listing 4.2: Creation of a tracing proxy

A delegation proxy is a special object whose behavior is defined by a separate *handler* object. Methods in the handlers are referred as *traps*. When an operation (message send, state accesses, etc.) is applied to a proxy, the proxy reifies the operation and instead invokes the corresponding trap in the handler. The handler has a *target*. The handler takes some action and then reflectively performs the original operation on the target.

Proxies and handlers are distinct to avoid name conflicts between application methods and traps, *i.e.*, between the base-level and the meta-level. The target of a handler can be a regular object or another proxy. In the later case, proxies form chains of delegation. Handlers are regular objects that can be proxied as well. Figure 4.1 shows the relationships between proxy, handler and target.

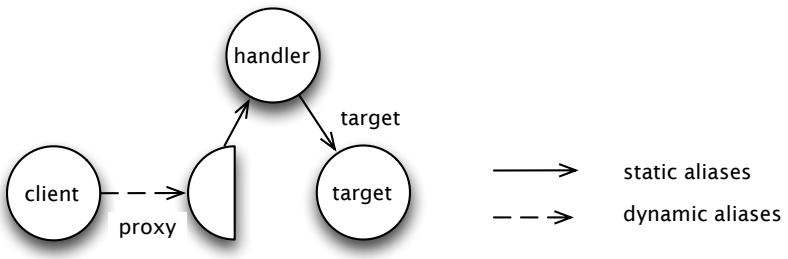


Figure 4.1: A proxy with a handler and its target.

In our Smalltalk implementation, handlers define the following traps:

- *Message Sends* The trap for message sends takes as parameters the original sender, the selector, the arguments.
- *Field Reads* The trap for field reads takes as parameters the field name.
- *Field Writes* The trap for field writes takes as parameters the field name and the value to write.
- *Literals* The trap for the resolution of literals (symbols, string, numbers, class names, and closures) takes as parameters the resolved literal.

All traps take an additional parameter `myself`. When a trap is invoked, the parameter `myself` refers to the proxy on which the operation was originally applied. The message trap of the tracing handler looks as follows:

```
TracingHandler>>message: aMessage
          sender: theSender
          myself: myself

     Transcript show: target asString;
            show: ' ';
            show: aMessage selector;
            cr.

     ↑ target perform: aMessage
             sender: theSender
             myself: myself.
```

Listing 4.3: A simple tracing handler

The reflective invocation with `perform` takes two additional parameters, `sender` and `myself`. The parameter `myself` specifies how `self` must be rebound. The sender can be used to distinguish between self-sends and regular messages. We support a true delegation model where `self` is rebound to the original receiver of the message [99]. This contrasts to the mere forwarding of operations that is usual with proxies.

Similarly, reflective methods to read and and write fields also take an additional parameter `myself`. They become `instVarNamed:myself:` and `instVarNamed:put:myself`. The possible stratification of those reflective capabilities with mirrors [28] is orthogonal to our approach. For simplicity, we assume they are not stratified.

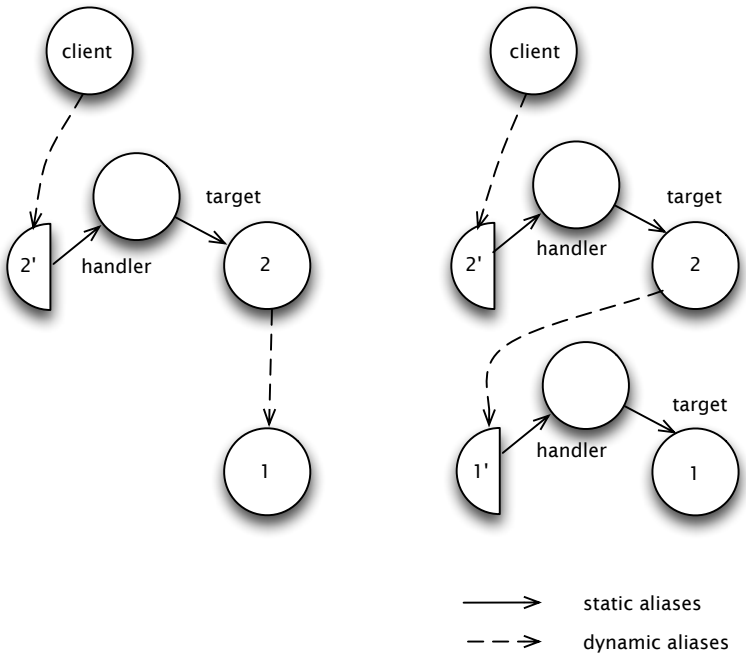


Figure 4.2: Illustration of transitive propagation. On the right, the proxy `2'` transitively proxies the results of operation on its base object, the number `1`.

4.2.1 Propagation

The tracing handler in the previous section defines a *message* trap. Doing so ensures that messages received by the proxy are traced, including self-sends. However, it would fail to trace messages sent to other objects. The evaluation of `2 tracing fib` would print `2 fib, 2 <, 2 -, 2 -`, but message sends to numbers `1, 0, true, false` would not be traced.

Within the evaluation of the base method, the provenance of another object can be i) an argument, ii) the result of a state read, iii) the return value of a message send, iv) the resolution of a literal. Tracing can be propagated by transitively "proxying" the results of state reads, and the resolutions of literals. State writes must unproxy the base object before the actual state write. Arguments do not need to be proxied, except for the initial message send. Here the initial message is `2 fib`, which has no arguments. We need to proxy the results of primitive message sends that return references to objects *ex nihilo*, but do not otherwise need to proxy the return value of message sends. Such primitive messages include explicit instantiations with `new` and arithmetic computations with `+,-,/`. If the receiver of the message, the arguments of the message, the results of state reads, and

the results of primitive messages are all proxies already, there is no way to obtain a reference to a base object.

```
PropHandler>>literal: literal myself: myself
  ↑ self wrap: literal.

PropHandler>>message: msg sender: sender myself: myself
  ... code to trace ...
  msg selector isPrimitive ifTrue: [
    ↑ self wrap: ( target perform... ).
  ].
  ↑ target perform...

PropHandler>>readField: field myself: myself
 ↑ self wrap: ( target instVarNamed ... ).

PropHandler>>writeField: field value: val myself: myself
  target instVarNamed: field
    put: (self unwrap: val )
    myself:  myself .
  ↑ val.

PropHandler>>wrap: anObject
  | handler |
  handler ← self class target: anObject.
  ↑ Proxy handler: handler.

PropHandler>>unwrap: aProxy
  ↑ ( Reflect handlerOf: aProxy ) target.
```

Listing 4.4: Tracing handler implementing the propagation technique. For readability, we use ... when the arguments are passed as-is to the reflective operation, and omit the code that prints the tracing messages.

Figure 4.2 and Listing 4.4 illustrate this technique. We assume the existence of a class `Reflect` to unproxy objects. When the number is subtracted (with message `-`) the resulting object is proxied as well since it is a primitive operation. This way, tracing is consistently applied during the computation of Fibonacci numbers. Delegation proxies can be leveraged to adapt the evaluation of an expression and its sub-expressions.

4.2.2 Closures

The evaluation of a closure created when tracing was enabled should *not* activate tracing again. For instance, if the closure `[ self printString ]` is created when tracing is enabled, its evaluation during an execution without

any adaption should not trace the message `printString`. Inversely, if the closure `[ self printString ]` is created during an execution without any adaption, its evaluation when tracing is enabled should trace the message `printString`. For this to work correctly, closures are always created in an *unproxied* form, and proxying is only applied on demand.

Let us describe first how creation works and illustrate it with the closure `[ self printString ]` and tracing:

1. The closure is created by the runtime and captures variables as-is. *Tracing example:* the closure captures `self`, which refers to a proxy.

2. The closure creation is intercepted by the *literal* trap of the creator. *Tracing example:* the closure is treated like other literals and thus proxied.

3. If the closure was proxied, the runtime invokes the *write* trap of the closure's proxy for all captured variables. *Tracing example:* the runtime invokes the *write* trap of the closure's proxy passing 0 as field name and the `self` proxy as value. The trap unproxies the value and reflectively invokes `instVarNamed:put:myself:` for field 0. This overwrites the previous value in the closure with a reference to the base object.

Unlike regular fields, captured variables in the closure are stored in anonymous fields identified by index. Evaluation of closures follows the inverse scheme:

1. If the closure is evaluate via a proxy, the runtime invokes the *read* trap each time a captured variable is accessed. *Tracing example:* the runtime invokes the *read* trap of the closure's proxy passing 0 as field name. The trap reflectively invokes `instVarNamed:myself:` for field 0 and wraps the result with a proxy. The message `printString` is sent to the proxy.

Note that this scheme is quite natural if we consider that closures could be encoded with regular objects, similarly to anonymous classes in Java. In that case, captured variables are effectively stored in synthetic fields initialized in the constructor. The instantiation of the anonymous class would trigger *write* traps, and evaluation would trigger *read* traps.

Closures provide a convenient way to activate a behavioral variation. Adding method `valueWithHandler` in `BlockClosure`, tracing `2 fib` can also be achieved with `[2 fib] valueWithHandler: TracingHandler` instead of `2 tracing fib`. When we evaluate the closure via the proxy, the *literal* trap will proxy `2` before it is used. Closures provide a convenient way to activate a behavioral variation in the dynamic extent of expression.

```
BlockClosure>> valueWithHandler: aHandlerClass
    | handler |
    handler ← aHandlerClass target: self.
    ↑ ( Proxy handler: handler ) value.
```

Listing 4.5: Convenience method to change semantics

4.2.3 Forwarding

Since both the sender and the proxy to which the operation was originally applied (see `myself` Listing 4.3) are passed as parameters when the trap is invoked, the proxy can bypass self-sends to fall back to forwarding. The tracing handler below would trace only messages sent from other objects:

```
TracingHandler>>message: aMessage
          sender: theSender
          myself: myself

     (myself == theSender) ifFalse: [
         Transcript show: target asString;
            show: ' ';
            show: aMessage selector;
            cr.
     ].

     ↑ target perform: aMessage
             sender: theSender
             myself: myself.
```

Listing 4.6: Tracing without self-sends

Alternatively, the handler can also fall back to forwarding by *not* rebinding self to the proxy when invoking the method reflectively.

4.2.4 Transparency

Traps are implemented in a separate handler and not in the proxy itself. If an application defines an application-level method that collides with the name of a trap, the explicit invocation of this method will be trapped correctly by the handler. It avoids conflicts between the base- and meta-levels. The proxy exposes the exact same interface as its target.

Identity comparison is realized via message sends that are reified as usual. The handler can either delegate the comparison to the target, or use its own identity for comparison (handlers are regular objects). In the first case, a proxy and its target are indistinguishable without using reflective

facilities. In the second case, two proxies with the same handler are still indistinguishable. In both cases, several proxies of a given target can be created without confusion of identity.

It is impossible to deconstruct a proxy to obtain its handler without using reflective capabilities. How reflective capabilities are provided exactly is orthogonal to our approach. There are however essentially two ways: additional language syntax [12] and mirrors [28]. For simplicity throughout this chapter, we assume the existence of a class `Reflect` that exposes the following methods globally:

- `Reflect class>>isProxy: aProxy` Returns whether the argument is a proxy or not.
- `Reflect class>>handlerOf: aProxy` If the argument is a proxy, returns its handler. Otherwise, it fails.

If these methods were to be stratified with mirrors, handlers would need to be parametrized with a mirror when instantiated. Mirrors could also be used to stratify the reflective capabilities to invoke methods, read fields, and write fields as well (see Section 4.2).

4.3 Examples

We exemplify now the use of delegation proxies. First we show traditional examples of proxies (the examples would be supported by forwarding proxies as well) in order to illustrate the rationale for various design decisions of delegation proxies.

Second, we show how the active variability enabled by delegation proxies and our propagation technique enables new language extensions. These examples are marked with a star ($\star$). They all rely on the propagation technique presented earlier. For readability, we assume that handlers marked with a star ($\star$) inherit from a base class that implements the propagation technique (see Listing 4.4) for reuse.

4.3.1 Lazy Values

The actual creation of an object can be delayed until it is really needed with a lazy value. Since the base object doesn't initially exist, the proxy is created initially with `nil` as a target. The proxy's handler takes a block (*i.e.*, a function) that creates the object upon the first access. When the base object has been instantiated the target is replaced. This example illustrates the motivation to have the handler refer to the target, instead of the proxy. It can be implemented with forwarding proxies as well.

Below is the code to create a lazy proxy of the value `42 fib`. Since the target is initially `nil`, the factory method `Proxy>>for:` cannot infer the type of the target. We implemented delegation proxies with code generation and the type of the target is required and must be passed as a parameter in this case. The creation block must succeed and return an instance of the expected type.

```
| handler proxy |
handler ← LazyHandler block: [ 42 fib ].
proxy ← Proxy for: handler class: Number.
```

Listing 4.7: Creation of a lazy proxy

The actual handler looks as follow:

```
LazyHandler>>message: aMessage
              sender: theSender
              myself: myself

  target isNil ifTrue: [
    target ← block value.
  ].
  ↑ target perform: aMessage
         sender: theSender
         myself: myself.
```

Listing 4.8: Lazy handler

4.3.2 Membranes

A membrane is a security construct for access control that transitively imposes revocability on all references exchanged via the membrane [110, 162, 44]. Objects that are exchanged inward or outward are wrapped in a proxy.

A membrane is a first-class object that keeps track of the proxies it manages. When the membrane is revoked, it resets the target of all managed proxies to `nil`, which guarantees garbage collection of objects within the membranes. Subsequent invocations via a revoked proxy fail:

```
| membrane proxyArray val |
membrane ← Membrane new.
proxyArray ← membrane proxyFor: { 42 }.
val ← proxyArray at: 1. "Proxy of 42"
val printString. "42"
membrane revoke.
val printString. "fail"
```

Listing 4.9: Revoking a membrane

If proxies of a base object all have the same identity, the membrane can create multiple proxies for objects exchanged multiple times via the membrane. In the code snippet below, `c1` and `c2` are two distinct objects that share the same identity. This illustrates the motivation for transparent proxies discussed in subsection 4.2.4:

```
| membrane baseArray c1 c2 |
membrane ← Membrane new.
proxyArray ← membrane proxyFor: { 42 }.
c1 ← proxyArray at: 1. "Proxy of 42"
c2 ← proxyArray at: 1. "Proxy of 42"
c1 == c2. "true"
c1 handler == c2 handler. "false"
```

Listing 4.10: Transparency of revokable references

Relevant code of the membrane and membrane handler is shown below. The handler wraps the arguments of the message send as well as its return value. The handler forwards the invocation: it does not rebind `self` to the proxy.

```
MembraneHandler>>message: aMessage
             sender: theSender
             myself: myself
   | newArgs  retVal newMsg |
   self membrane failIfRevoked. "Fail if revoked"
   newArgs ← aMessage arguments collect:
                [ :arg |  membrane proxyFor: arg  ].
   newMsg ← Message selector: aMessage selector
                 arguments: newArgs.
   retVal ←  target perform: newMsg
                 sender: theSender
                 myself: target.
   ↑ membrane proxyFor: retVal

Membrane>>proxyFor: aRef
    | proxy handler |
    handler ← MembraneHandler target: aRef membrane: self.
    ↑ self add: (Proxy handler: handler ).
```

Listing 4.11: Membrane and membrane handler

4.3.3 Layers ⋆

Delegation proxies enable the contextualization of crosscutting behavior like tracing. So far, we have shown how the evaluation of a closure can be

"parametrized" with a handler, *e.g.*, `[2 fib] valueWithHandler: TracingHandler`. In this case, the handler is the entity that encapsulates the behavioral variation.

This principle could be extended to implement first-class layers. Instead of passing a handler to `valueWithHandler:`, the developer passes a layer to a method `valueWithLayer:`. The method `valueWithLayer:` creates a proxy with a handler of type `LayerHandler` and configures the handler with the layer. The handler then uses the layer to decide how to intercept message sends.

For instance, a test layer that overrides the method `asString` of class `Object` could be defined with:

```
TestLayer>>asString
    <target: Object>
    ↑ 42.
```

Listing 4.12: A simple layer

The handler inspects the layer's pragmas (`<target:>`) to decide how to dispatch the message send. Evaluating a piece of code with the test layer activated is as simple as `[...] valueWithLayer: TestLayer new`.

One limitation of the approach is however that variations cannot be introduced for super sends: once the proxy reflectively performs an operation on its target, it loses control over the evaluation until the next trap is invoked. Also, there is no way to reflectively invoke a method and specify at which level of the class hierarchy the lookup should start; the lookup will always start with the class of the receiver. This is a general problem of the use of reflection to simulate custom message lookup rules.

4.3.4 Interceptors ⋆

Previous sections already illustrated delegation proxies using tracing. The exact same approach could be used to implement other interceptors like profiling or code contracts. Below is the code of a profiling handler.

```
ProfilingHandler>>message: aMessage
        sender: theSender
        myself: myself
  | start |
  start ← Time now.
  [   ↑ super message: aMessage
        sender: theSender
        myself: myself
  ]  ensure: [ Transcript show: (Time now - start) ; cr.  ]
```

Listing 4.13: A simple profiling handler

Since the propagation is implemented reflectively, it can be customized. This flexibility enables interceptors to be limited to application objects by simply stopping the propagation for kernel objects (dictionaries, arrays, etc.). The application and the kernel are two layers. Any application object referenced by a kernel object must have been provided by the application. If application objects are proxied, this guarantees that kernel objects hold only references to proxies of application objects. Therefore, if a kernel object sends a message to an application object, the propagation will start again. Omitting kernel objects from the propagation can improve performance.

4.3.5 Object Versioning ⋆

To tolerate errors, developers implement recovery blocks that undo mutations and leave the objects in a consistent state [134]. Typically, this requires cloning objects to obtain snapshots. Delegation proxies enable the implementation of object versioning elegantly. Before any field is mutated, the handler 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. Similarly to the other examples that follow, we assume that the handler inherits from a base handler that implements the propagation technique.

```
RecordingHandler>>writeField: field
                   value: newVal
                   myself: myself
  | oldValue |
  oldValue ← target instVarNamed: field myself: myself.
  log add: { target . field . oldValue }.
  ↑ super writeField: field value: newVal myself:  myself
```

Listing 4.14: Recording handler

A convenience method can be added to enable recording with `[...] recordInLog: aLog`. Recording can be thought of as a behavioral layer that is activated during the evaluation of the block.

```
BlockClosure>>recordInLog: aLog
  | res handler |
  handler ← RecordingHandler target: self log: aLog.
 ↑ ( Proxy handler: handler ) value
```

Listing 4.15: Enabling recording

4.3.6 Read-only Execution ⋆

Read-only execution [11] prevents mutation of state during evaluation. Read-only execution can dynamically guarantee that a given piece of code is side-effect free, or that clients of an object do not mutate it by mistake.

Classical proxies could restrict the interface of a given object to the subset of read-only methods. They would fail to enable read-only execution of arbitrary functions, or to guarantee that read-only methods do not mutate other objects. Read-only execution can be implemented trivially using the propagation technique and a handler that fails upon state writes.

```
ReadOnlyHandler>>writeField: aField
                  value: aValue
                  myself: myself
   ReadOnlyError signal:  'Illegal write'.
```

Listing 4.16: Read-only handler

Similarly to the previous example, a convenience method can be added to turn on read only execution `[...] evaluateReadOnly`.

```
BlockClosure>> evaluateReadOnly
  | res handler |
  handler ← ReadOnlyHandler target: self.
 ↑ ( Proxy handler: handler ) value
```

Listing 4.17: Enabling read-only execution

4.3.7 Dynamic Scoping ⋆

In most modern programming languages, variables are lexically scoped and can't be dynamically scoped. Dynamic scoping is sometimes desirable, for instance in web frameworks to access easily the ongoing request. Developers must use in this case alternatives like thread locals. It is for instance the strategy taken by Java Server Faces in the static method `getCurrentInstance()` of class `FacesContext`[1]).

Dynamic scoping can be realized in Smalltalk using stack manipulation [51] or by accessing the active process. Delegation proxies offer an additional approach to implement dynamic bindings by simply sharing a common (key,value) pair between handlers. If multiple dynamic bindings are defined, objects will be proxied multiple times, once per binding. When a binding value must be retrieved, a utility method locates the handler corresponding to the request key, and returns the corresponding value:

[1] `http://www.webcitation.org/6FOF4DFab`

```
ScopeUtils>>valueOf: aKey for: aProxy
   | h p |
   p ← aProxy.
   [ Reflect isProxy: p ] whileTrue: [
      h ← Reflect handlerOf: p.
      ( h bindingKey == aKey ) ifTrue: [
         ↑ h bindingValue.
      ].
      p ← h target.
   ].
   ↑nil. "Not found"
```

Listing 4.18: Inspection of a chain of proxies

During the evaluation of a block, a dynamic variable can be bound with `[...] valueWith: #currentRequest value: aRequest` and accessed pervasively with `ScopeUtils valueOf: #currentRequest for: self`.

4.4 Semantics

We formalize delegation proxies by extending SMALLTALKLITE [16], a lightweight calculus in the spirit of Featherweight Java that omits static types. This chapter does not assume any prior knowledge of it. Our formalization simplifies two aspects of the semantics presented in the previous sections:

1. It models neither first-class classes nor literals. Consequently, a *literal* trap does not make sense. Instead, we introduce a *new* trap that intercepts object instantiations.

2. It models a proxy as a pair (handler, target). When a trap is invoked on the handler, the target is passed as parameter. In the previous examples, the handler holds its target. We make this simplification to ease the presentation of identity proxy.

The syntax of our extended calculus, SMALLTALKPROXY, is shown in Figure 4.3. The only addition to the original syntax is the new expression **proxy** *e e*.

During evaluation, the abstract syntax tree of the program is annotated with the object and class context of the ongoing evaluation, since this information is missing from the static syntax. For instance, the super call $\mathbf{super}.m(v^*)$ is decorated with its object and class into $\mathbf{super}\langle c\rangle.m\langle o\rangle(v^*)$ before being interpreted; **self** is translated into the value of the corresponding object; message sends $o.m(v^*)$ are decorated with the current object context to keep track of the sender of the message. The rules for the translation of expressions into redexes are shown below.

$$
\begin{array}{rcl}
P & = & \mathit{defn}^* e \\
\mathit{defn} & = & \textbf{class}\ c\ \textbf{extends}\ c\ \{\ f^* \mathit{meth}^*\ \} \\
\mathit{meth} & = & m(x^*)\ \{\ e\ \} \\
e & = & \textbf{new}\ c \ \mid\ x \ \mid\ \textbf{self} \ \mid\ \mathsf{nil} \ \mid\ f \ \mid\ f = e \\
 & \mid & e.m(e^*) \ \mid\ \textbf{super}.m(e^*) \ \mid\ \textbf{let}\ x = e\ \textbf{in}\ e \\
 & \mid & \textbf{proxy}\ \ e\ e
\end{array}
$$

Figure 4.3: Syntax of SMALLTALKPROXY

$$
\begin{array}{rcl}
o[\![\textbf{new}\ c]\!]_c & = & \textbf{new}\langle o\rangle\ c \\
o[\![x]\!]_c & = & x \\
o[\![\textbf{self}]\!]_c & = & o \\
o[\![\mathsf{nil}]\!]_c & = & \mathsf{nil} \\
o[\![f]\!]_c & = & f\langle o\rangle \\
o[\![f = e]\!]_c & = & f\langle o\rangle = o[\![e]\!]_c \\
o[\![e.m(e_i^*)]\!]_c & = & o[\![e]\!]_c.m\langle o\rangle(o[\![e_i]\!]_c^*) \\
o[\![\textbf{super}.m(e_i^*)]\!]_c & = & \textbf{super}\langle c\rangle.m\langle o\rangle(o[\![e_i]\!]_c^*) \\
o[\![\textbf{let}\ x = e\ \textbf{in}\ e']\!]_c & = & \textbf{let}\ x = o[\![e]\!]_c\ \textbf{in}\ o[\![e']\!]_c \\
o[\![\textbf{proxy}\ \ e\ e']\!]_c & = & \textbf{proxy}\ \ o[\![e]\!]_c\ o[\![e']\!]_c\ o[\![e]\!]_c\ o[\![e']\!]_c
\end{array}
$$

Figure 4.4: Translating expressions to redexes

Redexes and their subexpressions reduce to a value, which is either an address a, nil, or a proxy. A proxy has a handler h and a target t. A proxy is itself a value. Both h and t can be proxies as well. Subexpressions may be evaluated within an expression context E.

Translation from the main expression to an initial redex is carried out by the $o[\![e]\!]_c$ function (see Figure 4.4). This binds fields to their enclosing object context and binds **self** to the value o of the receiver. The initial object context for a program is nil. (*i.e.*, there are no global fields accessible to the main expression). So if e is the main expression associated to a program P, then $\mathsf{nil}[\![e]\!]_{\mathsf{Object}}$ is the initial redex.

$P \vdash \langle \epsilon, \mathcal{S}\rangle \hookrightarrow \langle \epsilon', \mathcal{S}'\rangle$ means that we reduce an expression (redex) ϵ in the context of a (static) program P and a (dynamic) store of objects $\mathcal{S}$ to a new expression ϵ' and (possibly) updated store $\mathcal{S}'$. The store consists of a set of mappings from addresses $a \in \mathrm{dom}(\mathcal{S})$ to tuples $\langle c, \{f \mapsto v\}\rangle$ representing the class c of an object and the set of its field values. The initial value of the store is $\mathcal{S} = \{\}$.

$$
\begin{array}{rcl}
\epsilon & = & o \mid \mathbf{new}\langle o\rangle\ c \mid x \mid \mathbf{self} \mid \mathsf{nil} \\
& \mid & f\langle o\rangle \mid f\langle o\rangle = \epsilon \mid \epsilon.m\langle o\rangle(\epsilon^*) \\
& \mid & \mathbf{super}\langle c\rangle.m\langle o\rangle(\epsilon^*) \mid \mathbf{let}\ x = \epsilon\ \mathbf{in}\ \epsilon \\
E & = & [\,] \mid f\langle o\rangle = E \mid E.m\langle o\rangle(\epsilon^*) \\
& \mid & o.m\langle o\rangle(o^*\ E\ \epsilon^*) \mid \mathbf{super}\langle c\rangle.m\langle o\rangle(o^*\ E\ \epsilon^*) \\
& \mid & \mathbf{let}\ x = E\ \mathbf{in}\ \epsilon \mid \mathbf{proxy}\ E\ \epsilon \\
r & = & \mathsf{nil} \mid a \\
o, v, t, h & = & r \mid \mathbf{proxy}\ h\ t
\end{array}
$$

Figure 4.5: Runtime redex syntax

The reductions are summarized in Figure 4.6. Predicate $\in_P^*$ is used for field lookup in a class, $f \in_P^* c$, and method lookup, $\langle c, m, x^*, e\rangle \in_P^* c'$, where c' is the class where the method was found in the hierarchy. Predicates $\leq_P$ and $\prec_P$ are used respectively for subclass and direct subclass.

If the object context $\langle o\rangle$ of an instantiation with $\mathbf{new}\langle o\rangle\ c$ is a reference (*i.e.*, not a proxy), the expression reduces to a fresh address a, bound in the store to an object whose class is c and whose fields are all nil [new]. If the object context of the instantiation is a **proxy** $h\ t$, the **newTrap** is invoked on the handler instead [new-proxy]. The trap takes the result of the instantiation $\mathbf{new}\langle t\rangle\ c$ as parameter; it can take further action or return it as-is.

The object context $\langle o\rangle$ of field accesses and field writes can be an object address a or a **proxy** $h\ t$. A local field access in the context of an object address [get] reduces to the value of the field. A local field access in the context of a **proxy** $h\ t$ [get-proxy] invokes the trap **readTrap** on the handler h. A field update in the context of an object address [set] simply updates the corresponding binding of the field in the store. A local field update in the context of a **proxy** $h\ t$ [set-proxy] invokes the trap **writeTrap** on the handler h. The sender, **proxy** $h\ t$, is passed as parameter of the trap invocation.

Messages can be sent to an object address a or to a **proxy** $h\ t$. When we send a message to an object address [send], we must look up the corresponding method body e, starting from the class c of the receiver a. The method body is then evaluated in the context of the receiver, binding **self** to the address a. Formal parameters to the method are substituted by the actual arguments. We also pass in the actual class in which the method is found, so that **super** sends have the right context to start their method lookup. When a message is sent to a **proxy** $h\ t$, the trap **callTrap** is invoked on the handler. The object context $\langle s\rangle$ that decorates the message corresponds to the *sender* of the message. The trap takes as parameters the message and its arguments, the sender s, and the initial receiver of the message **proxy** $h\ t$.

$$P \vdash \langle E[\mathbf{new}\langle r\rangle\ c], \mathcal{S}\rangle \hookrightarrow \langle E[a], \mathcal{S}[a \mapsto \langle c, \{f \mapsto \mathsf{nil} \mid \forall f, f \in^*_P c\}\rangle]\rangle \quad [new]$$
where $a \notin \mathrm{dom}(\mathcal{S})$

$$P \vdash \langle E[\mathbf{new}\langle \mathbf{proxy}\ h\ t\rangle\ c], \mathcal{S}\rangle \hookrightarrow \langle E[o], \mathcal{S}'\rangle \quad [new\text{-}proxy]$$
where $\langle E[h.\mathbf{newTrap}(\mathbf{new}\langle t\rangle c, \mathbf{proxy}\ h\ t)], \mathcal{S}\rangle \hookrightarrow^*\ \langle E[o], \mathcal{S}'\rangle$

$$P \vdash \langle E[f\langle a\rangle], \mathcal{S}\rangle \hookrightarrow \langle E[o], \mathcal{S}\rangle \quad [get]$$
where $\mathcal{S}(a) = \langle c, \mathcal{F}\rangle$ and $\mathcal{F}(f) = o$

$$P \vdash \langle E[f\langle \mathbf{proxy}\ h\ t\rangle], \mathcal{S}\rangle \hookrightarrow \langle E[o], \mathcal{S}'\rangle \quad [get\text{-}proxy]$$
where $\langle E[h.\mathbf{readTrap}(t, f, \mathbf{proxy}\ h\ t)], \mathcal{S}\rangle \hookrightarrow^*\ \langle E[o], \mathcal{S}'\rangle$

$$P \vdash \langle E[f\langle a\rangle = o], \mathcal{S}\rangle \hookrightarrow \langle E[o], \mathcal{S}[a \mapsto \langle c, \mathcal{F}[f \mapsto o]\rangle]\rangle \quad [set]$$
where $\mathcal{S}(a) = \langle c, \mathcal{F}\rangle$

$$P \vdash \langle E[f\langle \mathbf{proxy}\ h\ t\rangle = o], \mathcal{S}\rangle \hookrightarrow \langle E[o'], \mathcal{S}'\rangle \quad [set\text{-}proxy]$$
where $\langle E[h.\mathbf{writeTrap}(t, f, o, \mathbf{proxy}\ h\ t)], \mathcal{S}\rangle \hookrightarrow^*\ \langle E[o'], \mathcal{S}'\rangle$

$$P \vdash \langle E[a.m\langle s\rangle(o^*)], \mathcal{S}\rangle \hookrightarrow \langle E[a[\![e[o^*/x^*]]\!]_{c'}], \mathcal{S}\rangle \quad [call]$$
where $\mathcal{S}[a] = \langle c, \mathcal{F}\rangle$ and $\langle c, m, x^*, e\rangle \in^*_P c'$

$$P \vdash \langle E[(\mathbf{proxy}\ h\ t).m\langle s\rangle(o^*)], \mathcal{S}\rangle \hookrightarrow \langle E[o'], \mathcal{S}'\rangle \quad [call\text{-}proxy]$$
where $\langle E[h.\mathbf{callTrap}(t, m, o^*, s, \mathbf{proxy}\ h\ t)], \mathcal{S}\rangle \hookrightarrow^*\ \langle E[o'], \mathcal{S}'\rangle$

$$P \vdash \langle E[\mathbf{super}\langle c\rangle.m\langle s\rangle(o^*)], \mathcal{S}\rangle \hookrightarrow \langle E[s[\![e[o^*/x^*]]\!]_{c''}], \mathcal{S}\rangle \quad [super]$$
where $c \prec_P c'$ and $\langle c', m, x^*, e\rangle \in^*_P c''$ and $c' \leq_P c''$

$$P \vdash \langle E[\mathbf{let}\ x = o\ \mathbf{in}\ \epsilon], \mathcal{S}\rangle \hookrightarrow \langle E[\epsilon[o/x]], \mathcal{S}\rangle \quad [let]$$

Figure 4.6: Reductions for SMALLTALKPROXY

super sends [super] are similar to regular message sends, except that the method lookup must start in the superclass of the class of the method in which the **super** send was declared. In the case of super send, the object context $\langle s\rangle$ corresponds to the sender of the message as well as the receiver. The object context is used to rebind **self**. When we reduce the **super** send, we must take care to pass on the class c'' of the method in which the **super** method was found, since that method may make further **super** sends.

Finally, **let** **in** expressions [let] simply represent local variable bindings. Errors occur if an expression gets stuck and does not reduce to an a or to nil. This may occur if a non-existent variable, field or method is referenced (for example, when sending any message to nil, or applying traps on a handler h that isn't suitable). We are not concerned with errors, so we do not introduce any special rules to generate an error value in these cases.

$$P \vdash \langle E[a.\mathbf{call}(m, o^*, s, my)], \mathcal{S}\rangle \hookrightarrow \langle E[my[\![e[o^*/x^*]]\!]_{c'}], \mathcal{S}\rangle \quad [\textit{reflect-call}]$$
where $\mathcal{S}[a] = \langle c, \mathcal{F}\rangle$ and $\langle c, m, x^*, e\rangle \in^*_P c'$

$$P \vdash \langle E[(\mathbf{proxy}\ h\ t).\mathbf{call}(m, o^*, s, my)], \mathcal{S}\rangle \hookrightarrow \langle E[o'], \mathcal{S}'\rangle \quad [\textit{reflect-call-proxy}]$$
where $\langle E[h.\mathbf{callTrap}(t, m, o^*, s, my)], \mathcal{S}\rangle \hookrightarrow^* \langle E[o'], \mathcal{S}'\rangle$

$$P \vdash \langle E[a.\mathbf{read}(f, my)], \mathcal{S}\rangle \hookrightarrow \langle E[o], \mathcal{S}\rangle \quad [\textit{reflect-get}]$$
where $\mathcal{S}(a) = \langle c, \mathcal{F}\rangle$ and $\mathcal{F}(f) = o$

$$P \vdash \langle E[(\mathbf{proxy}\ h\ t).\mathbf{read}(f, my)], \mathcal{S}\rangle \hookrightarrow \langle E[o'], \mathcal{S}'\rangle \quad [\textit{reflect-get-proxy}]$$
where $\langle E[h.\mathbf{readTrap}(t, f, my)], \mathcal{S}\rangle \hookrightarrow^* \langle E[o'], \mathcal{S}'\rangle$

$$P \vdash \langle E[a.\mathbf{write}(f, o, my)], \mathcal{S}\rangle \hookrightarrow \langle E[o], \mathcal{S}[a \mapsto \langle c, \mathcal{F}[f \mapsto o]\rangle]\rangle \quad [\textit{reflect-set}]$$
where $\mathcal{S}(a) = \langle c, \mathcal{F}\rangle$

$$P \vdash \langle E[(\mathbf{proxy}\ h\ t).\mathbf{write}(f, o, my)], \mathcal{S}\rangle \hookrightarrow \langle E[o'], \mathcal{S}'\rangle \quad [\textit{reflect-set-proxy}]$$
where $\langle E[h.\mathbf{writeTrap}(t, f, o, my)], \mathcal{S}\rangle \hookrightarrow^* \langle E[o'], \mathcal{S}'\rangle$

$$P \vdash \langle E[\mathbf{unproxy}(\mathbf{proxy}\ h\ t)], \mathcal{S}\rangle \hookrightarrow \langle E[t], \mathcal{S}\rangle \quad [\textit{unproxy}]$$

Figure 4.7: Reflective facilities added to SMALLTALKPROXY

4.4.1 Identity Proxy

As was discussed in subsection 4.2.4, the system requires the ability to reflectively apply operations on base objects and proxies to be useful. How these facilities are provided is orthogonal to our approach. For simplicity, we extend the language with three additional non-stratified reflective primitives: **call**, **read**, and **write**. The semantics of these primitives is given in Figure 4.7.

All three primitives take a last argument *my* (shortcut for "myself") representing the object context that will be rebound. Additionally, the identity of the sender *s* can be specified for reflective message sends. When applied to a proxy, the operations invoke the corresponding trap in a straightforward manner, passing *s* and *my* as-is. When **read** or **write** is applied to an object address, the arguments *s* and *my* are ignored. When **call** is applied to an object address, *my* defines how **self** will be rebound during the reflective invocation.

With these primitives, we can trivially define the identity handler, idHandler. idHandler is an instance of a handler class that defines the following methods:

$$
\begin{array}{lcl}
\textbf{newTrap}(t, my) & = & t \\
\textbf{readTrap}(t, f, my) & = & t.\textbf{read}(f, my) \\
\textbf{writeTrap}(t, f, o, my) & = & t.\textbf{write}(f, o, my) \\
\textbf{callTrap}(t, m, o^*, s, my) & = & t.\textbf{call}(m, o^*, s, my)
\end{array}
$$

We can show that sending a message to an identity proxy will delegate the message to the target, and rebind **self** to the proxy.

Let us consider an object s that sends the message $m(o)$ to a proxy $p =$ **proxy** idHandler t. Object t is an instance of class c which defines method $m(x)$ with body $e =$ **self** $n(x)$.

$$
\begin{array}{ll}
p.m\langle s \rangle(o) & \\
\mathsf{idHandler}.\textbf{callTrap}(t, m, o, s, p) & [\textit{call-proxy}] \\
t.\textbf{call}(m, o, s, p) & [\textit{call}] \\
p[\![e[o/x]]\!]_c & [\textit{reflect-call}] \\
p[\![\textbf{self}]\!]_c.n\langle p \rangle(o) & [\textit{translation}] \\
p.n\langle p \rangle(o) & [\textit{translation}]
\end{array}
$$

4.4.2 Propagating Identity Proxy

The identity handler can be turned into a propagating identity handler, idHandler*, following the technique of propagation presented in subsection 4.2.1. This technique requires the ability to unproxy a proxy. The expression **unproxy** is added to the language as defined in Figure 5.8. We also assume the existence of the traditional sequencing (;) operation.

The handler idHandler* is defined as follows:

$$
\begin{array}{lcl}
\textbf{newTrap}(t, my) & = & \textbf{proxy}\ \mathsf{idHandler}^*\ t \\
\textbf{readTrap}(t, f, my) & = & \textbf{proxy}\ \mathsf{idHandler}^*\ (t.\textbf{read}(f, my)) \\
\textbf{writeTrap}(t, f, o, my) & = & t.\textbf{write}(f, \textbf{unproxy}\ o, my); o \\
\textbf{callTrap}(t, m, o^*, s, my) & = & t.\textbf{call}(m, o^*, s, my)
\end{array}
$$

We can formally express the intuitive explanation of subsection 4.2.1 about soundness of the propagation.

Let us assume that all values in the expression $E[e]$ are proxies (using the idHandler*). The reduction rules that can match are [new-proxy], [get-proxy], [set-proxy], [super], [let], and [call-proxy]. According to the definition of the idHandler* traps, rule [new-proxy] will preserve the invariant that all values are proxies. Rule [get-proxy] does so as well. Rule [set-proxy] preserves the assumption since it returns the value written, which we know is a proxy. Rules [super] and [let] do as well since they only bind variables with existing values, which we know are proxies. Similarly, rule [call-proxy] will bind **self** with the expected proxy (see previous section). It also binds the variables with the passed arguments, which are known to

be proxies. Since all values remain proxies, the evaluation is consistent.

The initial redex is evaluated with nil as object context: $\mathsf{nil}[\![e]\!]_{\mathsf{Object}}$. If the proxy $p = \textbf{proxy}\ \mathsf{idHandler}^*\mathsf{nil}$ is used instead of nil, the assumption is initially true, and will not be broken during evaluation.

4.5 Implementation

We have implemented a prototype of delegation proxies in Smalltalk as a program transformation. The transformation adds two parameters, `myself` and `sender` to rewritten methods. Instead of `self`, `myself` is used in the transformed method body[2]. Following the same approach as Uniform Proxies for Java [59], proxy classes are auto-generated. Let us consider the class `Suitcase`:

```
Object>>subclass: #Suitcase
   instanceVariableNames: 'content'

Suitcase>>printString
  ↑ 'Content: ' concat: content.
```

Listing 4.19: Original code of class `Suitcase`

Applying our transformation, the class `Suitcase` is augmented with synthetic methods to read and write the field `content`, as well as a literal trap. The definition of the class remains unchanged.

```
Suitcase>>literal: aLiteral myself: slf
  ↑ aLiteral.

Suitcase>> readContentMyself: slf
  ↑ content.

 Suitcase>> writeContent: value myself: slf
  ↑ content ← value.
```

Listing 4.20: Synthetic methods to read and write instance variable `content`

In Smalltalk, fields are encapsulated and can be accessed only by their respective object. The sender of a state access is always `myself`, and can thus be omitted from the traps. The existing method in class `Suitcase` is then rewritten in two versions, one with additional parameters `myself` and `sender`, and one with parameter `sender` only. The second one uses the receiver of the message as `myself`.

[2]This is similar to Python's explicit `self` argument

```
Suitcase>>printStringMyself: slf sender: s
  ↑ ( slf literal: 'Content: ' myself: slf )
     concat: (self readContentMyself: slf) sender: slf.

Suitcase>>printStringSender: s
  ↑ self printStringMyself: self sender: s
```

Listing 4.21: The method `printString` is rewritten to two versions

A proxy class for `Suitcase` is then generated. It inherits from a class `Proxy`, which defines the `handler` field common to all proxies. The generated class implements the same methods as the `Suitcase` class, *i.e.*, `printStringSender:`, `printStringMyself:sender:`, `readContentMyself:`, and `writeContent:myself:`. The methods invoke respectively *message*, *read* and *write* traps on the handler.

```
SuitcaseProxy>> printStringMyself: slf sender: s
  | msg |
  msg ← Message selector: #printString arguments: {} .
  ↑ handler message: msg myself: slf sender: s.
```

Listing 4.22: Sample generated method in proxy class of `Suitcase`

Smalltalk has first-class classes whose behaviors are defined in meta-classes. The class and meta-class hierarchies are parallel. Classes can be proxied like any object. Consequently, meta-classes are rewritten and extended with synthetic methods similarly to classes. However, the generated proxy classes do not inherit from `Class`, but `Proxy`, as is shown in Figure 4.8.

Closures are regular objects that are adapted upon creation and evaluation according to subsection 4.2.2.

Exceptions and exception classes are regular objects that are proxied as well by our propagation technique. Since exception handling is implemented within Smalltalk using first-class activation frames, it would be possible to make it compatible with proxies.

4.5.1 Performance

We need to distinguish between the performance of delegation proxies themselves and the technique that uses them to implement adaptations. Delegation proxies have in themselves little impact on performance. The replacement of the implicit `self` with an explicit `myself` is mostly negligible. The introduction of traps for literal does however degrades performance.

Sending a message to a proxy entails reification of the message, invocation of the handler's trap, and then reflective invocation of the message on

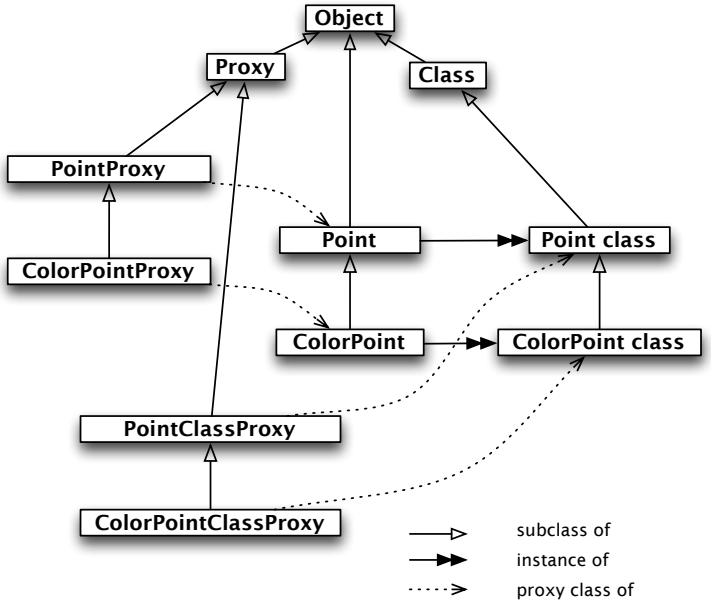


Figure 4.8: Inheritance of classes, meta-classes, and auto generated proxy classes.

the target. This has a cost when used extensively. In addition, the handler might take additional actions that entail costs. This is the case with our propagation technique.

Benchmarks of Fibonacci[3] reveal a performance degradation of two orders of magnitude (44x slower) between executions with our technique (using propagation of the identity handler) and without it. Manually weaving the handlers and proxies into specialized classes (*e.g.*, `NumberPropIdProxy`) reduces the overhead to below one order of magnitude (8x slower). We believe it is a good result when we consider that delegation proxies enable unanticipated behavioral reflection, which is known to be costly.

To further reduce the overhead, our propagation technique can be adapted to proxy only instances of application classes and skip system classes. Depending on the nature of the adaptation, this choice might be viable or even desirable.

The automatic weaving of handlers and proxies can be addressed in future work using techniques for partial evaluation [63] developed for aspect compilers [107]. Future work could also address the caching of literal interceptions. Indeed, in the Fibonacci examples, `1`, `2` and `[^ self]` are literals that are intercepted and unnecessarily proxied thousands of times.

[3]CogVM 6.0, Mac OS X, 2.3 GHz Intel Core

4.5.2 Static Typing

There is no major obstacle to port our implementation to a statically-typed language. Delegation proxies preserve the interface of their target, like traditional forwarding proxies. For type compatibility, the generated proxy must inherit from the original class. Reflective operations can fail with runtime type errors. Forwarding and delegation proxies suffer the same lack of type safety from this perspective.

Delegation proxies require however that reflective operations have an additional parameter that specifies how to rebind `self`. Naturally, this parameter must be of a valid type: in practice it will be either the target of the invocation or a proxy of the target. Both implement the same interface. Rebinding `self` with an "arbitrary" object that is of a valid type but is not related to the target would be technically sound, but is undesirable in practice. Reflective operations should reject this case to favor good design.

4.6 Related Work

We now compare in more details delegation proxies with selected related work presented in Chapter 2.

Proxies and Reflection MOPs, AOP and proxies are various approaches that enable the interception and customization of method dispatch. MOPs reify the execution into meta-objects that can be customized [90]. AOP adopts another perspective on the problem and enables the definition of join points where additional logic is woven [91]. MOP and AOP share similarities with method combination of CLOS [48]. Proxies enable the reification of method dispatch on a per-object basis.

Proxies have found many usefully applications that can be categorized as "interceptors" or "virtual objects" [162]. An important question for proxies is whether to support them natively at the language level or via lower-level abstractions. Most dynamic languages support proxies via traps that are invoked when a message cannot be delivered [106]. Modern proxy mechanisms stratify the base and meta levels with a handler [106, 59, 162], including Java that uses code generation to enable proxies for interfaces. The mechanism was extended to enable proxies of classes as well [59].

Delegation proxies do not suffer from meta-regression issues of AOP and MOP [157] since the adapted object and the base object are distinct. For instance, the tracing handler in Listing 4.3 does not lead to a meta-regression since it sends the message `asString` to the target, which is distinct from the proxy (in parameter `myself`). System code can in this way be adapted. Also, delegation proxies naturally enable partial reflection [158] since objects are selectively proxied.

Recent works on proxies in dynamic languages have studied orthogonal issues related to stratification [162, 44], preservation of abstractions and invariants [152, 45], and traps for values [12]. Only Javascript direct proxies support delegation [45]. However, Javascript proxies do not enable the interception of object instantiations; the variables captured in a closure will not be unproxied upon capture and proxied upon evaluation.

In addition to full-fledged MOPs and AOP, reflective language like Smalltalk provide various ways to intercept message sends [54]. Java and .NET support custom method dispatch via JSR 292 [125] and the Dynamic Language Runtime [109].

Composing Behavior Inheritance leads to an explosion in the number of classes when multiple variations (decorations) of a given set of classes must be designed. Static traits [144] 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.

Decorators refine a specific set of known methods, *e.g.*, the method `paint` of a window. Static and dynamic approaches have been proposed to decoration. Unlike decorators, proxies find their use when the refinement applies to unknown methods, *e.g.*, to trace all invocations. Büchi and Weck proposed a mechanism [31] to statically parameterize classes with a decorator (called wrapper in their terminology). Bettini *et al.* [17] proposed a similar construct but composition happens at creation time. Ressia *et al.* proposed *talents* [141] which enable adaptations of the behavior of individual objects by composing trait-like units of behavior dynamically. Other works enable dynamic replacement of behavior in a trait-like fashion [19].

The code snippet below illustrates how to achieve the decoration of a `Window` with a `Border` and shows the conceptual differences between these approaches. The two first approaches can work with forwarding or delegation (but no implementations with delegation are available). The third approach replaces the behavior or the object so the distinction does not apply.

```
// Buchi and Weck
Window w = new Window<Border>();
// Bettini
Window w = new BorderWrap( new Window() );
// Ressia
Window w = new WindowEmptyPaint();
w.acquire( new BorderedPaint() );
```

Listing 4.23: Differences between approaches to decoration

Several languages that combine class-based inheritance and object inheritance (*i.e.*, delegation) have been proposed [92, 163]. Delegation enables

the behavior of an object to be composed dynamically from other objects with partial behaviors. Essentially, delegation achieves trait-like dynamic composition of behavior.

Ostermann proposed delegation layers [128], which extend the notion of delegation from objects to collaborations of nested objects, *e.g.*, a graph with edges and nodes. An outer object wrapped with a delegation layer will affects its nested objects as well. Similary to decorators, the mechanism refines specific sets of methods of the objects in the collaboration.

Dynamic Scoping Techniques for method dispatch customization make it hard to customize the dispatch based on the control flow. AOP supports it for instance with `cflow`, but in a limited way.

In context-oriented programming (COP) [80, 165], variations can be encapsulated into layers that are dynamically activated in the dynamic extent of an expression. Unlike delegation proxies that support *homogenous* variations, COP supports best *heterogenous* variations [6]. COP can be seen as a form of multi-dimensional dispatch, where the context is an additional dimension.

Other mechanisms to vary the behavior of objects in a contextual manner are roles [94], perspectives [150], and subjects [75]. Delegation proxies can realize dynamic scoping via reference flow, by proxying and unproxying objects accesses during the execution. Delegation proxies can provide a foundation to design contextual variations.

Similarly to our approach, the handle model proposed by Arnaud *et al.* [11, 10] enables the adaptation of references with behavioral variations that propagate. The propagation belongs to the semantics of the handles, whereas in our approach, the propagation is encoded reflectively. Propagation unfolds from a principled use of delegation. Our approach is more flexible since it decouples the notion of propagation from the notion of proxy.

4.7 Conclusions

Using active variability is a convenient approach to introduce changes at run time while avoiding consistency issues. We have presented proxies that work by delegation instead of forwarding, and demonstrated how they can be used to implement such kind of variability.

In our approach, object instantiations are intercepted via the interception of class name resolutions (*i.e.*, literals), and a variations can propagate to all objects accessed in a dynamic extent. Contextual mechanisms can be built *on top* of delegation proxies. We have sketched for instance how first-class contextual layers could be implemented on top of delegation proxies. In future work, it could evolve into a mature implementation.

Delegation proxies improve the support of two modularity principles presented in Chapter 1:

- *Encapsulation.* Delegation proxies are flexible building blocks to implement contextual mechanisms that encapsulate behavioral variations.
- *Late Binding.* Delegation proxies enable `self` to be rebound to the proxy.

We can draw the following conclusions about the practicality of delegation proxies:

- *Metaness.* Delegation proxies naturally compose, support partial behavioral reflection, and avoid meta-regressions. We can for instance trace and profile an execution without interference between the two (composition). Objects are proxied selectively. Adapting objects during an execution will not affect other objects in the system (partial reflection). Proxies and targets represent the same object at two meta-levels but have distinct identities (no meta-regression).
- *Performance.* Delegation proxies do not entail performance issues when used sporadically (same situation as with forwarding proxies). To implement active variability, the behavioral variation must be propagated, which entails an overhead close to one order of magnitude in our implementation. The overhead could be reduced with a more mature implementation or with VM support.

With delegation proxies and active contexts, developers have powerful language features to enable run-time changes. Active contexts enable arbitrary changes including layout modifications, but require a careful design to ensure that the update stabilizes. Delegation proxies are easier to leverage but the range of changes they support is more restricted. Unlike active contexts, delegation proxies can encapsulate crosscutting behavior.

It is important to keep in mind that the ability to adapt software at run time is not only driven by the technicalities of features like active contexts and delegation proxies, but by the very organization of the code base in the first place. In the next chapter, we present an approach to address this point and promote a good modularization of code.

5 Scaling Information Hiding with Dynamic Ownership

5.1 Introduction

In a modular software system, application features should be encapsulated into objects that communicate via stable interfaces. Since the implementation details of an object are hidden to its clients, objects with similar interfaces but different implementations remain interoperable [2]. When a change must be introduced, the existing object can be replaced with a new one easily. Application servers rely for instance on this principle to enable the redeployment of web applications without restart. The application server communicates with the web application only via stable interfaces that have been standardized (*e.g.*, Java servlets).

Objects are routinely composed of other objects, though. For instance, a linked list is composed of nodes that must be organized in a certain way. Such a composite object is a system of objects [83]. The list must maintain invariants about multiple objects to properly work, and clients of the list should not depend on its internal nodes for the list to be easily replaceable. Unfortunately, in object-oriented languages the heap is an unstructured graph of objects, and there is hence no way to enforce information hiding beyond individual objects.

We propose in this chapter to improve information hiding by structuring objects in the heap in an ownership hierarchy (similarly to ownership types [39, 121, 23]) and to vary the behavior of objects depending on the relative positions of the caller and callee in the hierarchy. Each object in the tree is a *context* that alters the behavior of its children. In analogy to the

notion of *dynamic extent*, the children of an object in the hierarchy form its *structural extent*.

The behavioral variation works as follows. Methods are assigned one or more tags, which we call *topics* to avoid ambiguities. Each object can define *in* and *out* filters which are sets of topics. When a caller object sends a message to a receiver, the system checks whether the topic of the method is visible to the caller. If it isn't, the method fails. If it is, the method is executed as usual. In addition, the system checks that parameters and return values exchanged between objects do not result in references that go "up" in the ownership hierarchy. If this is the case, the system triggers a *crossing handler* before the faulty reference is established. By default, the crossing handler raises an exception and prevents such reference transfer. Crossing handlers are reflective hooks that can be modified by developers.

According to our definitions of structural and active variability, this mechanism is a mix of both: it belongs to structural variability since the ownership relationship defines the visibility of methods, and at the same time it belongs to active variability since the caller (*i.e.*, information in the dynamic extent) of the message send is used to define the behavioral variation.

Filters and crossing handlers default to a policy that confines objects to their owner. This policy is referred as owner-as-dominator. Filters and crossing handlers can however be configure to relax this policy. Filters in particular can be used to expose parts of an aggregate, or expose a limited view of the objects within an owner. The latter case allows read-only objects to be exposed. Crossing handlers facilitate the systematic implementation of defensive copying, which complements well the owner-as-dominator policy.

The chapter is organized as follows: Section 5.2 presents filters and crossing handlers and their default behavior; Section 5.3 and Section 5.4 show examples of filters and crossing handlers; Section 5.5 discusses the relationship between secure programming and ownership; Section 5.6 and Section 5.7 define the semantics and implementation of our variant of dynamic ownership; Section 5.8 describes the adaptation of the web server and Section 5.9 opens further perspectives. We discuss related work in Section 5.10 before we conclude in Section 5.11.

5.2 Filters and Crossing Handlers

Let us consider the web server in Figure 5.1. The web server contains web sites that are composed of web pages. The web server has a `hostname` and `port`. The web server uses a list to maintain references to its two web sites, called "Intranet" and "Extranet". Each web site has a name and a home web page. Web pages are instances of `Page`. Web pages can have subpages;

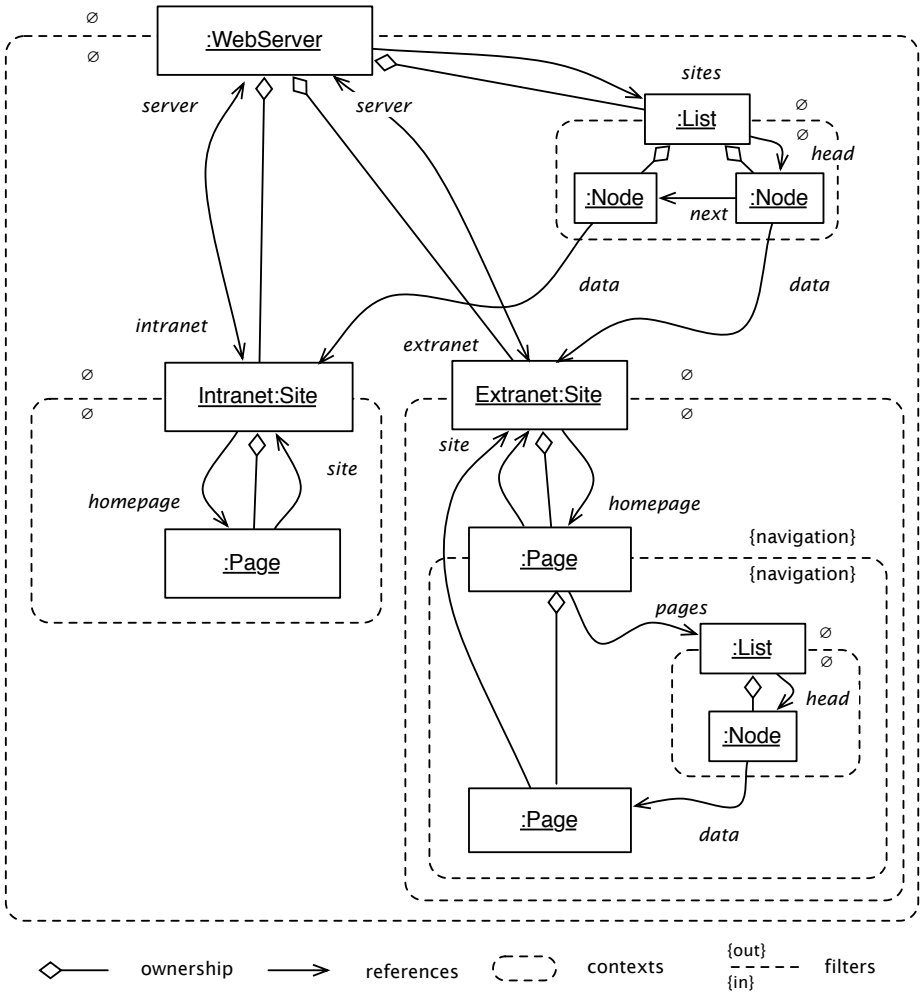


Figure 5.1: Example of an ownership topology.

a web page keeps references to its subpages in a list. Web pages have URLs that are of the form `http://host:port/site/relative/path`.

Information hiding is a well established principle and it is important to control the accessibility of object members to not break system invariants. For instance, to move a page between web sites, it is not enough to remove it from one list and add it to another list; in addition, the page and subpages must be updated to reference the right web site. The list of subpages should be manipulated only via methods exposed by the corresponding page.

Ownership Objects are organized at run-time in an ownership tree. In Figure 5.1, the web server owns the web sites; the web sites own the web pages; lists own their corresponding nodes. The nodes do *not* own the ob-

```
Page>>inFilters ↑ #(navigation)
Page>>outFilters ↑ #(navigation)
```

Listing 5.1: Definition of the in and out filters for the Page class

```
Page>>fullUrl
  <topic:navigation>
   ↑ site baseUrl , '/' , self relativeUrl.

Site>>baseUrl
  <topic:navigation>
  ↑ server hostUrl , '/' , self name.

Site>>siteMap
  <topic:navigation>
  ↑ homepage flattenChildren
             collect: [ :page | page fullUrl ].

Server>>hostUrl
   <topic:navigation>
   ↑ hostname , ':' , port.
```

Listing 5.2: Methods belonging to the navigation topic

jects they reference, though. Each object acts as a context object that affects the behavior of all the objects it owns directly and indirectly. The ownership tree is established at run-time: when new objects are instantiated, the owner of a new object is by default the sender of the new message.

Filters Methods belong to zero or more *topics*, and filters select sets of topics. From a given reference, a method is accessible only if its topics match the *in* and *out* filters of the contexts crossed by the reference. The effect of filters is cumulative. An object can access all methods of its parent, children, and siblings. No filter is applied in the case of such message sends.

In Figure 5.1, the class Page uses the topic navigation as in and out filters. Other classes have empty filters. Listing 5.1 shows how filters are technically defined. Let us consider the methods in Listing 5.2 that deal with URL and site map generation. The methods belong to the navigation topic:

- Pages cannot access method WebServer>>hostUrl since web sites have empty out filters. To render the URL, pages must use Site>>baseUrl.

- Nested pages can access method `Site>>baseUrl` since the out filter of pages is `navigation`.
- In `Site>>siteMap`, the site flattens the tree of pages into a list. It can access method `Page>>fullUrl` on all pages in the list, since the in filter of pages is `navigation`.

Crossing Handlers Filters can lead to references that expose no methods. Outgoing references (references going "up" in the hierarchy) of this kind are valid, while incoming references (references going "down" in the hierarchy) are not. All incoming references in the system must expose at least one method. With this restriction, hiding all methods of an object expresses confinement.

This corresponds to the traditional principles of alias protection [121]: nodes of a list can reference the data they hold (outgoing references), but external (incoming) references to the nodes violate information hiding and must be forbidden. Only the list can reference its nodes.

The event of an invalid reference transfer is reified and the corresponding *crossing handler* is triggered. By default, the crossing handler performs no action and raises an exception. It could however be adapted to perform some actions after which the reference transfer should be valid. Let us consider Figure 5.1, which uses the default crossing handler:

- When instances of `List`, `WebServer` and `Site` return a reference to objects they own, a crossing handler is fired since their in filter is empty. The default crossing handler will raise an exception and prevent the reference transfer.
- An instance of `Page` can return a reference to a subpages, since `Page`'s in filter contains the `navigation` topic.

Ownership Transfer When an object creates another object, it is assigned by default to be the owner of the new object. Since the default owner is not always appropriate, ownership can be transferred if necessary. Methods `Object>>owner` and `Object>>owner:` respectively query the current owner of an object, or modify it. Ownership transfer must preserve the tree structure of the ownership graph, and must not result in invalid incoming references.

5.2.1 Default Policy

By default, the set of *in filters* is empty, as well as the set of *out filters*. With an empty set of in filters, the topics of the methods are irrelevant and references to internal objects cannot be passed to the outside. The default

crossing handler raises an exception when a reference to an internal object is passed to the outside, either as a return value or as a parameter. This corresponds to the classic owner-as-dominator policy, which enforces confinement [39].

Let us consider the web server in Figure 5.1. The list of sites is implemented as a list composed of nodes. Since the set of in filters is empty, an attempt to return a reference to a node will trigger the crossing handler which will raise an exception: the list is an aggregate and the nodes are effectively inaccessible outside the aggregate. With an empty set of out filters, objects within a context can only depend on the identity of objects outside the context.

5.3 Using Filters

In our approach, each object acts as a *context object* that affects the behavior of its children. An object can have a behavior of its own or not. If an object has no behavior and has no other purpose than controlling the behavior of its children, we refer to it as *first-class context*. We first show how regular objects can be configured with filters to relax the owner-as-dominator policy. We then show how first-class contexts can be leveraged as well.

5.3.1 Iterators

Owner-as-dominator is too restrictive to implement common idioms like iterators: for efficiency the iterator must be owned by the aggregate to have access to internal data, but cannot then be returned to the outside [120].

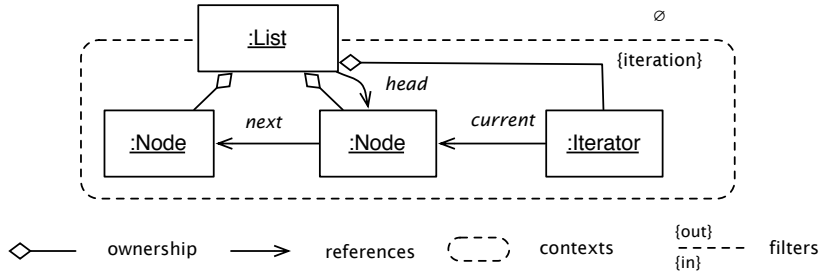


Figure 5.2: Iterators can be returned to the outside since they match the `iteration` topic.

In our approach, filters can easily be used to solve this situation. The list owns the iterator, which is then a sibling of the nodes and has full access to them. The in filters of the list contain the `iteration` topic, which match methods `next` and `current` of the iterator, shown in Listing 5.3. The iterator can be by consequence returned outside the list, while nodes cannot.

Several variants of ownership types using class nesting [23], ownership domains [3], relaxed constraints for dynamic aliases[1] [39] or additional access modifiers [101], have been devised to solve this problem. The implementation of dynamic ownership by Gordon and Noble [121, 70] relies on a special language feature to "export" objects to solve this issue. Filters and crossing handlers support this situation, while being general mechanisms.

```
Iterator>>next
  <topic:iteration>
   current ← current next.

Iterator>>current
  <topic:iteration>
  ↑ current data.
```

Listing 5.3: Methods belonging to the `iteration` topic

5.3.2 Read-only References

With owner-as-dominator, encapsulated objects cannot be returned to the outside, which effectively prevents unwanted modification to the internal representation from ever happening. When internal state must be exposed, a safe alternative is to expose only a limited read-only view. This is known as representation observation [25].

Let us consider that each web site has several administrators that are stored in an array. Administrators can be changed only via a special administration page. The ability to obtain an unrestricted reference to the internal array from outside the web server would imply that the list can be freely changed. To prevent mutations, the method `Array>>at:` is assigned the topic `read-only` and the in filters of the web site matches the `read-only` topic. The situation is shown in Figure 5.3. This way, objects outside the web site only have a limited access to the array.

Since the effect of filters is cumulative, read-only access will be applied transitively to all objects within the context. This works well for nested and recursive structures, as was shown previously when limiting access to the `navigation` topic for web pages.

There have been several proposals for read-only references [25]. For dynamic languages, only few approaches have been proposed. Schaerli *et al.* proposed encapsulation policies [145], which enable fine-grained control of the interface objects expose. It however fell short in dealing with recursive structures. Arnaud *et al.* proposed a specific solution to this problem with

[1]Static aliases correspond to references from instance variables. Dynamic aliases correspond to references from temporary variables, parameters, and return values. Static aliases are allocated in the heap. Dynamic aliases are allocated in the stack.

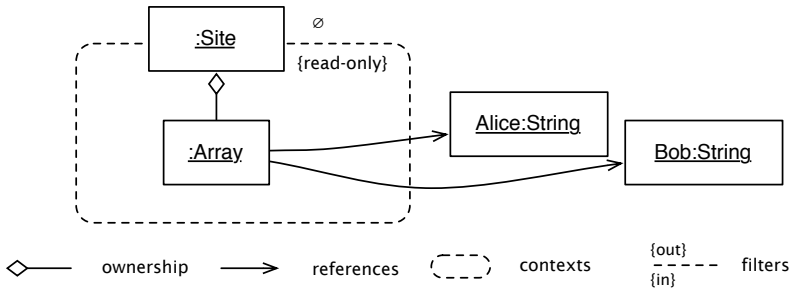


Figure 5.3: The site enables a read-only view of the internal array

read-only references [11]. In both cases, the policy is attached to a reference, not the object itself, and references with limited capabilities must be created explicitly. We believe these approaches are counter-intuitive since they define behavior based on the history of the reference instead of the dynamic context. Using the ownership topology appears to be much more natural.

5.3.3 Access Modifiers

The `public` and `protected` access modifiers can be simulated with first-class context objects. Such objects have no behavior of their own.

Each class categorizes its methods into one of two topics `public` and `protected`, in addition to other existing topics. Each object is then owned by a "wrapper" which exposes only public methods. Figure 5.4 shows this situation. Instead of owning the array directly, the web site owns now the wrapper of the array.

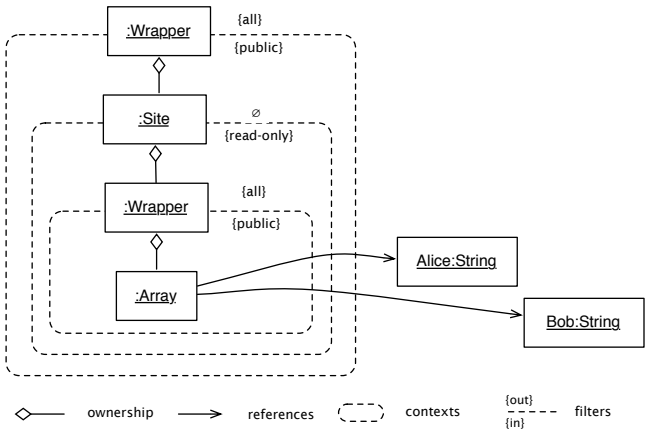


Figure 5.4: A synthetic wrapper exposes only `public` methods

This strategy implements instance protected methods: an object cannot access the protected methods of another object. Whether the receiver of a message is `self` or an alias of `self` has no impact. The strategy is similar to accessibility of instance variables in Smalltalk.

This contrasts with Ruby and Newspeak. Ruby implements class protected methods. Newspeak implements instance protected methods, but the lack of consideration of the sender in the method lookup algorithm results in different semantics for self sends with `self` or an alias of `self` [2]. Since our approach works with objects and not classes, we cannot simulate `private`.

5.3.4 Sandboxing

An object has full access to its siblings. Therefore, for the sake of security, one might want to protect objects within an additional first-class context.

We call this *sandboxing*. Figure 5.6 shows the design of the web server with the web handler, and illustrates two forms of sandboxing. The server is a generic infrastructure that abstracts the HTTP protocol. It manages instances of `Connection`, `Request` and `Response` classes. HTTP request data can be read with `Request>>fields` and the response is produced with `Request>>sendResponse:stream:` that takes a stream and an HTTP response code. The actual treatment of the request is delegated to a `WebHandler`. The handler is an extension of the web server that implements the desired functionality, *e.g.*, list directories, evaluate templates, *etc.*, The handler is an untrusted component from the point of view of the web server.

Two forms of sandboxing are possible, as shown in Figure 5.5 and Figure 5.6. In one case, the sandbox is around the web server, and in the other case it is inside the web server. In both cases, the web handler has only access to the `request` topic, which suffices to treat the request and produce the response.

5.3.5 First-class State

In our model, the interface an object exposes depends on its direct and indirect owners. While an object cannot "on its own" change the interface it exposes, we can very easily do so using first-class state [154]. Depending on which state it is owned, the object exposes a different set of operations. Ownership transfer is used to perform actual state changes.

Let us imagine that connections in Figure 5.6 have two states: open and closed. We model these settings with three objects, one being the connection itself, the two others the first-class states. The connection class defines two topics, `open` and `closed`. Each first-class state exposes one topic. When

[2] §5.7 of the specification

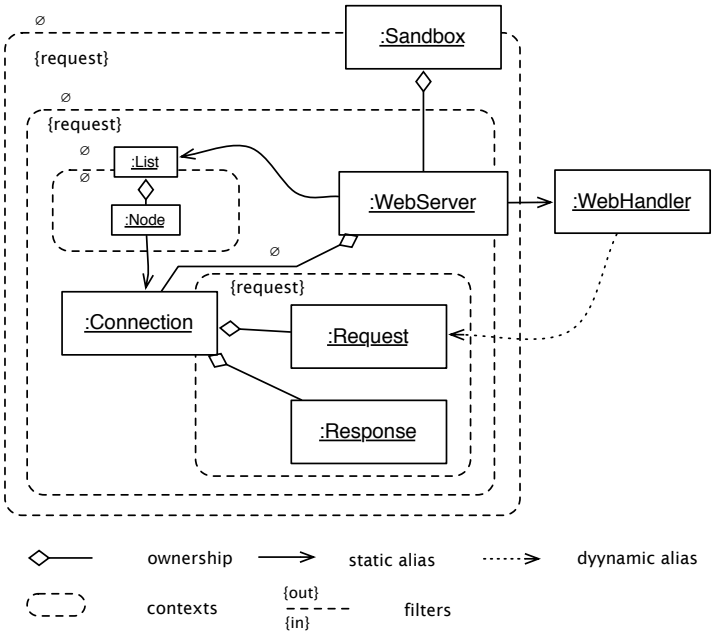


Figure 5.5: The web server is sandboxed

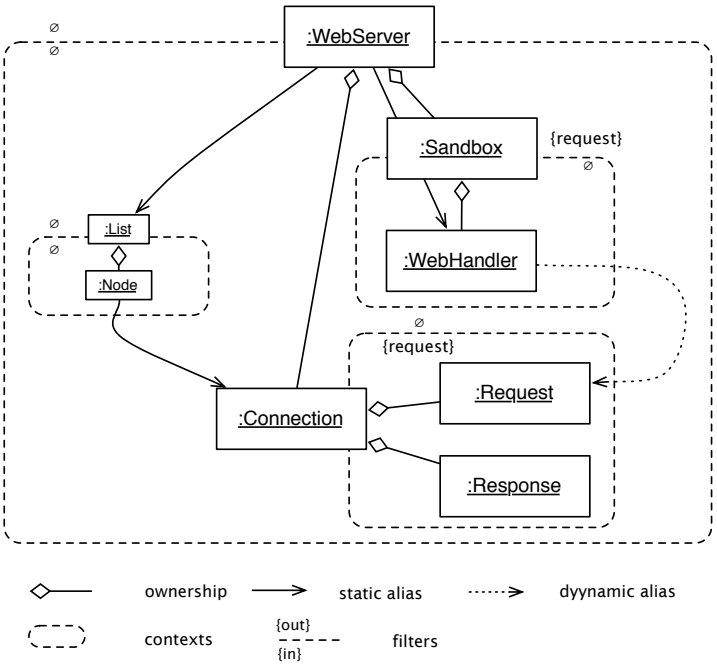


Figure 5.6: The handler is sandboxed

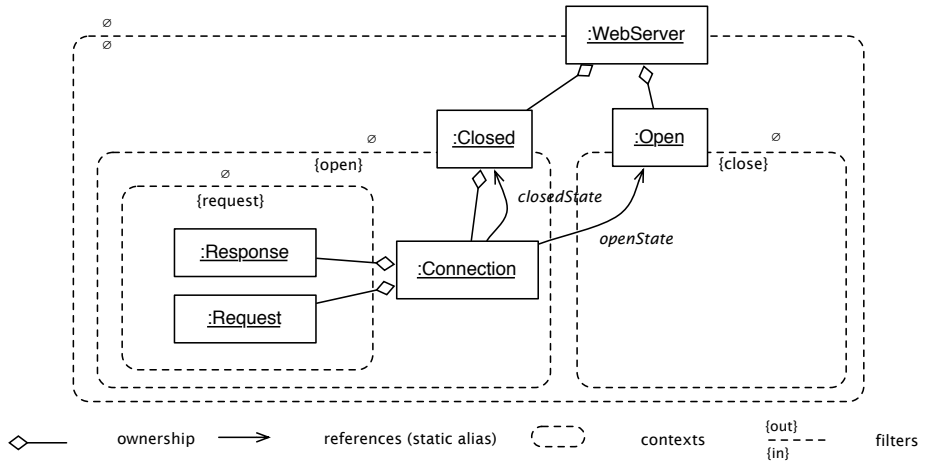


Figure 5.7: A connection can be owned either by the `closed` or `open` state.

the connection is owned by the open state, it exposes only methods of the open topic. Inversely, when it is owned by the closed state, it exposes the closed topic. The connection must be owned by either state. Figure 5.7 shows such a situation. Similarly to sandboxing, the web server owns now the states instead of the connection. For objects that reference the connection, the pattern is transparent.

Note that a useful variant of this technique can be used to "freeze" objects [95], after which they are immutable. All that is required is to implement a first-class "frozen" state, which filters out all mutating methods making the owned object effectively immutable.

5.4 Using Crossing Handlers

Examples in the previous chapter relied on the default crossing handler which raises an exception when an invalid reference transfer occurs. We now show how crossing handlers complement filters in useful ways.

5.4.1 Defensive Copying

Let us consider again the problem of the previous chapter with the administrators. With owner-as-dominator, encapsulated objects cannot be returned to the outside. When internal state must be exposed, it is a common practice to return a copy of the object. This technique is known as *defensive copying* [21].

Let us consider that each web site has several administrators whose names are stored in an array. Administrators can be changed only via a

special administration page. The ability to obtain a reference to the internal array from outside the web server would imply that the list can be freely changed. A typical implementation of the `administrator` accessor would copy the array before returning it[3]:

```
Site>>administrator
    ↑ administrator copy.
```

Languages do not have mechanisms to express such policies cleanly: developers must manually add code for copying objects whenever appropriate; copying and non-copying accessors might exist side by side and cause confusion; applying the technique systematically when modules grow is hard.

In our approach, the crossing handler can be overridden to implement the strategy. Whenever an encapsulated object is returned to the outside, the crossing handler is triggered. It copies the object being referenced and assigns it the sender as owner. The copy is then used for the reference transfer that is resumed.

```
Site>>handleCrossing: anObject sender: theSender
    ↑ anObject copy owner: theSender.
```

5.4.2 Remoting

With distributed objects, objects can be local and remote. Remote invocations have a pass-by-value semantics while local invocations have a pass-by-reference semantics. Parameters of remote invocations must be serializable. Since the caller does not know whether the receiver is local or remote, all parameters must be serializable. This prevents useful optimizations, such as using implicit futures as parameters.

Local and remote objects can be organized into distinct first-class contexts in the ownership hierarchy. The local context exposes only serializable objects. When a non-serializable object is passed as parameter, the crossing handler is fired and can attempt to resolve the conflict. For instance, if an implicit future is passed as parameter of a remote invocation, the handler can wait until its value is available and pass it instead.

```
Local>>handleCrossing: aFuture sender: theSender
    ↑ aFuture value.
```

[3]This example is intentionally close to Java's `Class.getSigners()` bug in early versions of the JDK. The method returned the internal array which could be tampered with by a malicious client to break security. This bug was motivational for ownership types [3].

5.4.3 Synchronization

Threads are objects. Objects that are owned by the threads are thread-local. Objects that are not owned by any thread are global. Rather than statically controlling thread locality [173], we control it dynamically. Threads expose only the `sharable` objects, *i.e.*, objects with at least one `sharable` method. Object that do not have `sharable` members cannot be passed to other threads or global object since the crossing handler triggers an error.

To pass thread local objects outside the boundary of the thread, they must be adapted first to become `sharable`. This adaptation can be done manually, or automatically in a crossing handler. For instance, a crossing handler can synchronize objects when they escape their threads by dynamically changing the class of the object (*e.g.*, Smalltalk's `become:` or `changeClassTo:`).

```
Thread>>handleCrossing: anObject sender: theSender
    ↑ anObject synchronize.
```

Note that the handler is triggered independently of whether the receiver would create a static alias of the thread local object or not. It is more conservative than tracking whether objects are reachable by multiple threads.

5.5 Security

Dynamic ownership can be used to increase the security of open systems. We consider in this section the impact of reflection and ownership transfer from the perspective of security.

5.5.1 Ownership Transfer

Ownership transfer could be used to bypass the constraints imposed by filters and crossing handlers, and thus break information hiding. In Figure 5.6, the handler is an untrusted component. Ownership is leveraged to constrain interactions between the handler and the web server to legal scenarios according to the principle of least privilege.

There are essentially two privilege escalation scenarios to consider: 1) a malicious object changes the owner of an object to obtain privileged access to it, and 2) a malicious object changes its owner to obtain privileged access to other objects.

Since ownership transfer is realized via regular message sends, it can be limited by using filters to mitigate the first threat: if ownership transfer is not exposed with in filters, external objects will not be able to transfer ownership of internal objects; if an object does not trust one of its internal

objects, it can sandbox it (see subsection 5.3.4) and use an out filter to prevent ownership transfers. It is preferable to own only objects one trusts. Container objects should usually not own their content, *e.g.*, a list does not own the data it holds.

Filters and crossing handlers are however not sufficient to address the second threat. The web handler could for instance make the web request its owner. This way, it would be a sibling of the web response and have full access to it. It could use this privilege to break encryption protocols.

To prevent such a case, the new candidate owner must accept the transfer first. The web request would for instance reject ownership of the web handler. In our approach, different objects can specify different ownership transfer policies by overriding the hook `Object>>acceptOwnershipOf: anObject`. Following the principle of deny by default, all transfers are by default rejected. For convenience, the acceptance check is however bypassed if the initiator of the transfer is the new owner itself. That is, `anObject owner: self` always succeeds.

5.5.2 Reflection

Unstratified reflective capabilities to query the class of an object (`class`), invoke methods (`perform`), and access the state of an object (`at` and `at:`) defeat information hiding. For instance, it is possible with reflection to inspect the value of a field for which there is no accessor.

Since reflection is also realized via regular message sends, it can be limited using filters. If the out filters hide reflective methods but the in filters expose them, objects can only reflect on objects they own, but not arbitrary objects. This way, a site can for instance reflect on all the pages it owns transitively, but not on the page of another site. If we want to prevent reflection between siblings, sandboxing can be used in addition (see subsection 5.3.4). This would prevent that a web site reflects on another web site.

The effect of reflective methods is not subject to constraints imposed by filters and crossing handlers: it is possible to reflectively invoke a method that would otherwise be inaccessible, and reflective accesses to the state of an object bypass crossing handlers. This way, features to save and restore a website (including its pages) could for instance be implemented using a generic reflection-based serializer. If the effects of reflective methods were also constrained, many useful patterns that use reflection would not be applicable any longer.

Reflection can be used to reveal information of an object at most one level down. Let us consider Figure 5.8. The web server can reflect on the homepage since the web site enables reflection. Despite the empty filters of the homepage, the web site can then reflectively obtain a reference to the leaf page. It would however fail to reflectively invoke methods on the leaf page or inspect its state.

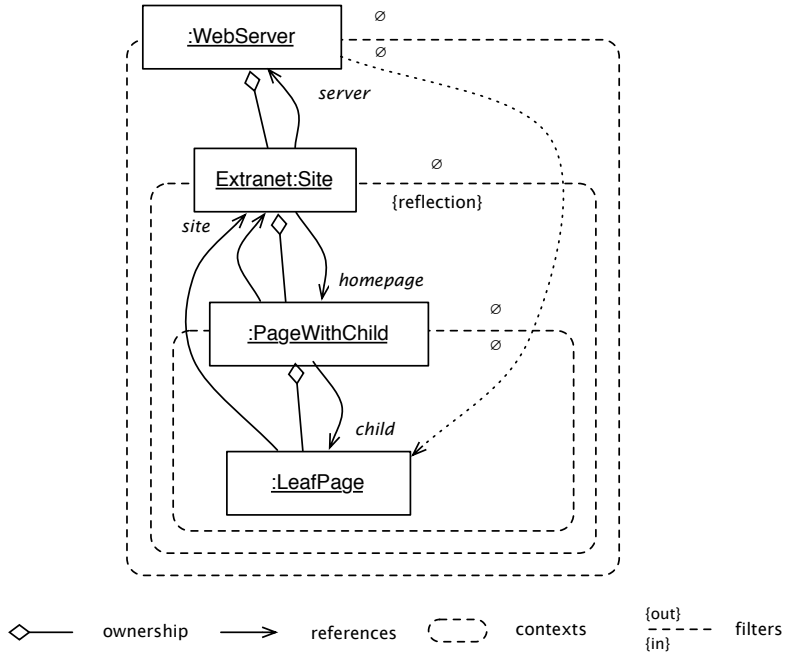


Figure 5.8: Reflection can be limited with filters. The dotted reference can be obtained only via reflection, since reflection bypasses accessibility and aliasing constraints.

5.6 Semantics

We have presented informally how our variant of dynamic ownership works with several examples. We describe in this section the semantics of the mechanisms more precisely, and how they can be integrated into a dynamic language. We omit custom crossing handler and use a set-theoretic approach to formalize the semantics of filters and the default crossing handler.

5.6.1 Ownership and References

The heap is a set of objects $\mathcal{O}$. The *world* is a special object in the heap, $world \in \mathcal{O}$. The partial function $owner : \mathcal{O} \rightarrow \mathcal{O}$ maps an object to its owner, possibly the *world*. *owner* is defined for all objects except *world*. The *owner* function defines a partial order $\prec$ over the set of objects:

$$o_1 \prec o_2 \iff \exists n > 0, owner^n(o_1) = o_2$$

The *world* is the indirect owner of all objects: $o \prec world, \forall o \in \mathcal{O}$. The function $references : \mathcal{O} \rightarrow 2^{\mathcal{O}}$ defines the existing references (static or dynamic).

5.6.2 Topics and Filters

For the purpose of explaining our model, we consider that methods are attached to objects, not classes. The details of method behavior is also irrelevant. We model only topics and filters as:

- *methods* : $\mathcal{O} \rightarrow 2^{\mathcal{M}}$ maps objects to methods names $m \in \mathcal{M}$;
- *topics* : $\mathcal{M} \rightarrow 2^{\mathcal{T}}$ maps method names to topic names $t \in \mathcal{T}$;
- *inFilters* : $\mathcal{O} \rightarrow 2^{\mathcal{T}}$ maps objects to set of topics that serve as in filters;
- *outFilters* : $\mathcal{O} \rightarrow 2^{\mathcal{T}}$ maps objects to set of topics that serve as out filters;

All methods belong to a special topic $all \in \mathcal{T}$, that can be used in filters.

If classes and inheritance were considered, rules for topic variance in overridden methods would need to be specified. Similarly to traditional access modifiers, subclasses can only make methods more visible, *i.e.*, they can only add topics to existing methods, not remove them.

5.6.3 Paths

The *ancestors* of an object o, $anc(o) = \{o' | o \prec o'\}$, is the set of all direct and indirect owners of that object, up to the $world$. Conversely, the *descendants* of an object, $desc(o) = \{o' | o' \prec o\}$ is the set of all objects that object owns directly or indirectly. The *depth* of an object, $d(o)$ is the cardinality of its set of ancestors, $d(o) = |anc(o)|$ The *first common ancestor* of two objects is:

$$com(o_1, o_2) = max(anc(o_1) \cap anc(o_2))$$

where max returns the object with the maximum depth. (Since the ownership relation forms a tree, there is a unique first common ancestor.)

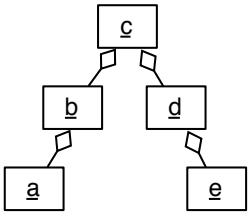
The *outPath* from an object a to an object b is the sequence of ancestors of a up to the first common ancestor of a and b. Respectively, the *inPath* from an object a to an object b is the sequence of descendants starting from the common ancestors of a and b down to b:

$$outPath(a, b) = \begin{cases} owner(a), outPath(owner(a), b) & \text{if } d(a) > D \\ () & \text{if } o/w \end{cases}$$

$$inPath(a, b) = \begin{cases} inPath(a, owner(b)), b & \text{if } d(b) > D \\ () & \text{if } o/w \end{cases}$$

where $D = 1 + depth(com(a, b))$.

The *path* from a to b is then the sequence $inPath(a,b), com(a,b), outPath(a,b)$. In the ownership tree below, $outPath(a,e) = (b)$ and $inPath(a,e) = (d)$. The *path* between a and e is (b,c,d).



5.6.4 Accessibility

An object $a \in \mathcal{O}$ can send message $m \in \mathcal{M}$ to object $b \in \mathcal{O}$ only if the topics of method m match the filters along the path from a to b. Let us define $exposesIn(o,m) \iff inFilters(o) \cap topics(m) \neq \emptyset$ and $exposesOut(o,m) \iff outFilters(o) \cap topics(m) \neq \emptyset$. Formally, the condition can be expressed as a predicate:

$$isAccessible(a,b,m) \iff \begin{cases} \forall o \in outPath(a,b), exposesOut(o,m) & \text{and} \\ \forall o \in inPath(a,b), exposesIn(o,m) \end{cases}$$

Let us consider Figure 5.4. If method `Array>>at` has topics `read-only` *and* `public`, it is visible outside of the web site, since each context exposes either the topic `read-only` *or* the topic `public`.

Note that $outPath(a,b) = \emptyset$ and $inPath(a,b) = \emptyset$ if a and b are parent and child, child and parent, or siblings. In these cases, the accessibility condition is trivially satisfied, and both objects can access all methods of the other.

5.6.5 Validity of References

A reference is valid if the contexts that it crosses inward expose at least one method; the contexts that it crosses outward could hide all methods, though. The validity of a reference between two objects can be defined as a variation of the accessibility for an individual method:

$$isValidReference(a,b) \iff \exists m \in \mathcal{M}, \forall o \in inPath(a,b), exposesIn(o,m).$$

Note that unlike *isAccessible*, *isValidReference* considers only the *inPath* between the two objects. The system must establish only valid references: $\forall a,b \in \mathcal{O}, b \in references(a) \Rightarrow isValidReference(a,b)$.

5.6.6 Instantiation

When an object o requests the creation of another object, a new object is allocated in the heap $\mathcal{O}$ and it is assigned o as its owner. A reference from o to the new object is established. Since o and the new object are parent and child, o can access all methods of the new object. This preserves the partial order $\prec$ and the consistency of the references at run-time.

5.6.7 Aliasing

Since references cover both static and dynamic aliases, passing references as parameters or return values of method invocations creates new references between objects. The system must establish only valid references and signal invalid reference transfer with an error.

Let us first consider return values. Let s, r, v be respectively the sender, receiver and return value of a message. The reference between r and v is valid, $v \in \mathit{references}(r)$ and $\mathit{isValidReference}(r, v)$, otherwise an error would have been raised previously. The system must ensure that the reference between s and v will be valid as well.

In practice, one does not need to check all contexts along the $\mathit{inPath}(s, v)$ to assess the validity of the reference. Let us define path truncation $\ominus$:

$$a, b \ominus c = \begin{cases} a & \text{if } b = c \\ a, b & \text{if } o/w \end{cases}$$

Since the reference from r to v is known to be valid, it suffices to assess whether the contexts $\mathit{inPath}(s, v) \ominus \mathit{inPath}(r, v)$ accept incoming references to v.

Figure 5.9 illustrates the references involved when `p1` sends a message to `s2` that returns a reference to `p2.1`. The *in* contexts crossed between `p1` and `p2.1` are {`s2`,`p2`}. The *in* contexts between `s2` and `p2.1` are {`p2`}. The list of contexts to check corresponds to $\mathit{inPath}(\texttt{p1},\texttt{p2.1}) \ominus \mathit{inPath}(\texttt{s2},\texttt{p2.1})$, which is {`s2`} in this case. Since `s2` has no in filters, the reference to `p2.1` exposes no method and `s2`'s crossing handler is fired. The handler raises an exception and prevents `p1` from obtaining a reference to `p2.1`.

The treatment of parameters is similar. Let s, r, p be respectively the sender, receiver and the parameter of interest of a message. The reference between s and p is known to be valid. The system must ensure that the reference between r and p will be valid as well. In practice, it suffices to assess whether the contexts $\mathit{inPath}(r, p) \ominus \mathit{inPath}(s, p)$ accept incoming references to p.

Custom crossing handler could be modeled by rewriting message sends $o.m(p^*, \ldots)$ as $\lfloor o.m(\lceil p^* \rceil_{\mathbf{self},o}, \ldots) \rfloor_{\mathbf{self},o}$. The operators $\lfloor \ldots \rfloor_{\mathbf{self},o}$ and $\lceil \ldots \rceil_{\mathbf{self},o}$ perform the checks for the parameters and return value as described above.

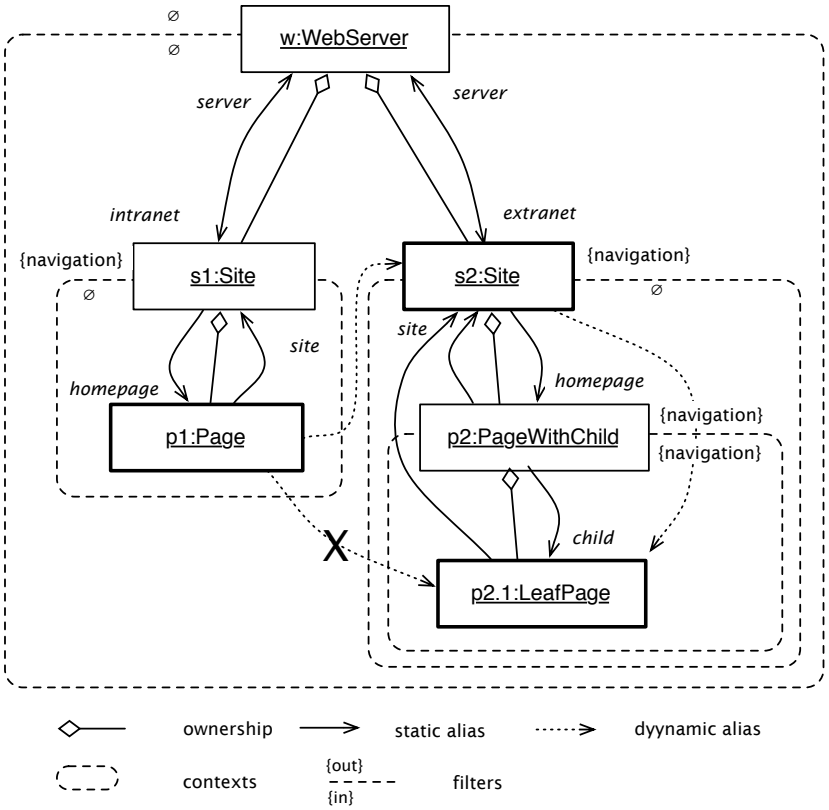


Figure 5.9: `p1` invokes a method on `s2` which returns a reference to `p2.1`. The *in* contexts crossed from `p1` to `p2.1` are {`s2`,`p2`}. The *in* contexts crossed from `s2` to `p2.1` are {`p2`}. The set of crossing handlers to check corresponds to {`s2`,`p2`} $\ominus$ {`p2`} = {`s2`}. `s2`'s crossing handler raises an exception and prevents that `p1` obtains a reference to `p2.1` (reference marked with a X).

They invoke the special method *handleCrossing* $\in \mathcal{M}$ for each context that is crossed inside.

5.6.8 Ownership Transfer

The owner of an object can be changed at run-time. The ownership transfer must preserve the partial order $\prec$ and the consistency of the references at run-time, though. Failure to do so results in a run-time error. In practice, this implies that the new owner must not be a descendant of the current owner, and that the system must verify the validity of all references to descendants of the impacted object.

Custom ownership transfer policies (see subsection 5.5.1) could be modeled by invoking a special method *acceptOwnershipOf* $\in \mathcal{M}$ on the new candidate owner. If the invocation returns false, the transfer is rejected.

5.7 Implementation

We have implemented a prototype of our variant of dynamic ownership in Pharo Smalltalk. Essentially, message sends must be intercepted to enforce the accessibility defined by filters, and execute crossing handlers if necessary. We implemented a compiler that transforms the original source and weaves it with additional logic, similarly to the technique used by Rivard to implement contracts [142]. Each call site is rewritten with one level of indirection that performs the additional logic, and then sends the original message.

5.7.1 Closures and `self`

The four statements in Listing 5.4 have the same intent: they send message `print:` to `self` with `anObject` as parameter. Self-sends are never filtered. In the first case, the self send is statically detected and is not rewritten with one level of indirection. The second case uses an alias `myself` of `self`. The self-send is not detected statically and the message send goes through one level of indirection. Cases 3 and 4 illustrate the peculiarities of closures. Within closures, `self` is not bound to the closure itself, but to the enclosing object. By consequence, the owner of a closure is the owner of the enclosing object. Therefore, despite the fact that closures are objects, `[...] value: anObject` in Listing 5.4 leads to a reference transfer that is within the same context. Also, since we rewrite the call sites, non-local returns within closures are correctly handled.

```
| myself |
myself ← self.
self print: anObject.                          "1"
myself print: anObject.                        "2"
[ :p | self print: p ] value: anObject.         "3"
[ :p | myself print: p ] value: anObject.       "4"
```

Listing 5.4: Message sends with similar intent

5.7.2 Primitive Types

Instances of `String`, `SmallInteger`, *etc.*, are immutable objects. Dynamic ownership raises the question whether such scalar values can be owned or not. We can argue that since they are immutable, so leaking such a value cannot compromise internal invariants of the object and it therefore makes no sense to own scalar values. On the other hand, such a value might still represent sensitive information that one does not want to expose. In this case, similar values need to be treated as distinct objects since their owner

can differ. For instance, two objects could each own an instance of a string with the same value. In our implementation, these objects are owned by the `world` and are not subject to dynamic accessibility and aliasing checks. This decision also applied to `nil`, the unique instance of `UndefinedObject`.

5.7.3 Control Flow

Control flow is realized in Smalltalk with message sends. The most common control flow messages (`ifTrue:ifFalse`, `whileTrue:`, *etc.*,) are not rewritten. The same treatment could be extended to boolean operators (`or:` and `and:`), assuming they are not redefined in other classes.

5.7.4 Ownership Transfer

After the owner of an object o has changed, the system must ensure that all references to objects within o's context are still valid (*i.e.*, expose at least one method). In Pharo, `Object>>pointersTo` performs a linear scan of all objects in memory and returns all objects pointing to a particular object.

5.7.5 First-class Classes

Objects are instantiated by sending the message `new` to the corresponding classes[4]. Classes are objects, and constructors are class-side methods that act as factories. In our implementation, classes are owned by the `world`. To assign a default owner to a newly instantiated object, our implementation intercepts the message `new`. The default owner is the sender of the `new` message. First-class classes pose two challenges to our model. First, constructor methods that use `self new` internally will produce objects owned by the class itself. Second, since classes are owned by the `world`, the constructor method might not be accessible if a context hides it. In our implementation, classes are treated in a special way to circumvent these problems.

5.8 Experiments

We used an existing Smalltalk web server[5] to experiment with filters and crossing handlers. We report here on our experience using these mechanisms.

[4] In Pharo, the primitive method is actually called `basicNew`

[5] Original sources: `http://www.squeaksource.com/WebClient.html`

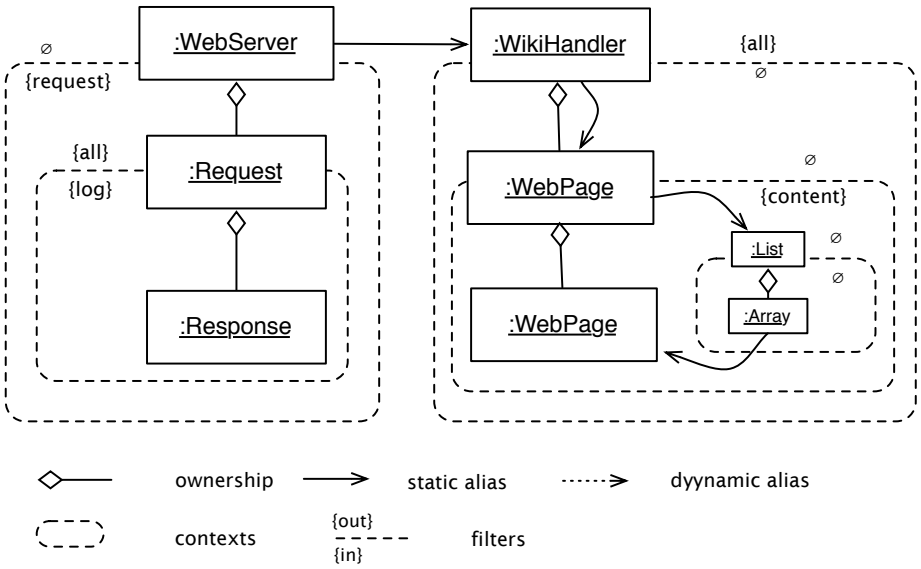


Figure 5.10: Key objects of the web server and their organization in an ownership structure.

5.8.1 Adapting the Web Server

Figure 5.10 shows the key objects of the web server and their organization in an ownership structure. The examples in the previous chapters were intentionally very similar to the design of this web server. The key classes are `WebServer`, `WebRequest` and `WebResponse`. Since the web server is only an infrastructure component to handle the HTTP protocol, we implemented in addition a minimal `WikiHandler` to view and edit `WebPage`. The web server uses several classes of the collection hierarchy that we adapted to our approach. The system consists of about 20 classes.

Ownership and Topics Two topics were mainly used to relax ownership: `request` and `content`. The first topic allows the web server to pass the request to the wiki handler. To process the request, the handler first locates the web page for the requested URL, and then renders the web page. The `request` topic enables the wiki to read (but not write) attributes of request such as the URL. The `content` topic enables the handle to access nested web pages and renders them. The response is owned by the request. It cannot be accessed by the web server. The web server logs entries with `WebServer>>logEntryFor: request response: response` so a third topic `log` was necessary. Only 14 methods needed to be annotated, and the effort was low.

Note that implementing `Request>>createLogEntry` would be more object-oriented and wouldn't need the `log` topic. In that sense, ownership favors object-orientation.

Factory Methods By default, the owner of an object is the one that invoked the corresponding factory method (see subsection 5.7.5). This is not always appropriate. For instance, `Collection>>select:` returns a collection that must have the same owner as the original collection. Such methods must be adapted to transfer the ownership after creation. The new owner accepts the transfer if it comes from an object it already owns.

Cloning Cloning should produce a new object that is indistinguishable from the original one. The implementation in `Object` was adapted to produce a copy of the object with the same owner as the original object. The implementation must be further adapted for specific classes. For instance, copying a dictionary copies the associations its owns as well. This kind of copying is called sheep cloning [96]. In this case, the owner of the copied associations must change to be the copied collection. Cloning can be considered to be a special factory method.

Utility Methods Class-side utility methods raise problems of ownership transfer similar to constructor methods (see subsection 5.7.5). We moved utility methods to a trait that could be reused whenever necessary. Code like `WebUtils decodeUrl: aString` is then rewritten as `self decodeUrl: aString`. This reformulation avoids problematic crossings, and accessibility restrictions. The trait contains 7 such methods.

Ownership Bugs The method `WebRequest>>fields` in Listing 5.5 violates information hiding and ownership: it adds the internal associations of a dictionary to another one without copying. Methods `associationsDo:` and `add:` are intended to be private to a dictionary. It is the goal of dynamic ownership to identity such violations.

```
fields
    | fields |
    fields ← Dictionary new.
    self getFields associations:[:a| fields add: a].
    self postFields associations:[:a| fields add: a].
    ↑fields
```

Listing 5.5: This code violates ownership and information hiding since it adds the internal associations of a dictionary to another one without copying.

A modified version of the code without the aliasing bug is shown below.

```
fields
    | fields |
    fields ← Dictionary new.
    self getFields keysAndValuesDo:
                [:key :val | fields at: key put: val ].
    self postFields keysAndValuesDo:
                [:key :val | fields at: key put: val].
    ↑fields
```

Listing 5.6: The modified code does not access the internal state of dictionaries.

This example shows that such code exists and that dynamic ownership can detect design anomalies.

5.8.2 Performance

Naturally, enforcing such behavioral variations at run time entails an overhead. The dynamic checks entail finding the common owner to two objects, and several manipulations of lists. Also, listing and comparing topics is expensive. Since our implementation was not optimized for performance, the overhead is significant.

The check for validity of references after an ownership transfer `transfer` was disabled during our evaluation of performance. Indeed, `pointersTo` is a very expensive operation that performs a linear scan of all objects in memory at the application level. Ownership transfers are occasional, so we can afford some overhead for them, but it would require support from the virtual machine in a production implementation.

Certain message sends can be optimized easily: messages sent to an alias of `self` can bypass all dynamic checks; messages sent to a child require only checking whether the return value crosses a context; messages sent to a parent require only checking whether the arguments cross a context; messages sent to a sibling require checking whether the arguments and the return value cross a context, but do not require checking for accessibility of methods.

Figure 5.11 shows micro benchmarks for message sends. Static self sends (*i.e.*, using `self`) are not rewritten and correspond to the performance of the original system. Dynamic self sends (*i.e.*, using an alias of self) go through one level of indirection, but bypass all dynamic checks. Dynamic self sends indicate that the level of indirection itself entails a degradation of factor 8 (25 vs. 3). Messages sent to a child, a parent, or a sibling can be partly optimized. Messages sent to "other" objects in the ownership

self (static)	self (dynamic)	parent to children	children to parent	sibling to sibling	other
3	25	28	41	47	390
3	25	28	41	48	87

Figure 5.11: Times (ms) for 10'000 executions of the method `returnParameter: 42` invoked on itself, a child, its parent, a siblings, or another object. The method takes a parameter as argument and returns it as-is. The first line shows measures when caching of accessibility checks is disabled, the second when it is enabled.

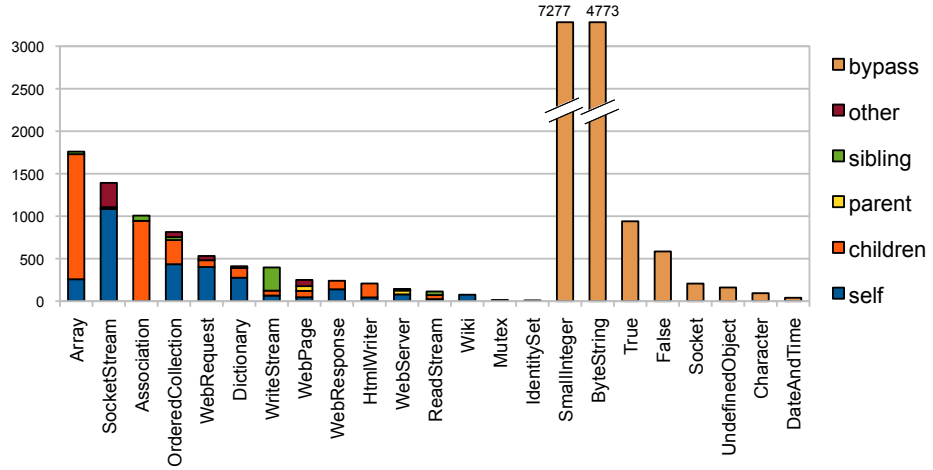


Figure 5.12: Distribution of message sends for the key classes (the graph is best viewed in color). The majority of interactions between objects are message sends that can be optimized.

structure require the execution of expensive logic to enforce the generic accessibility and aliasing constraints. The two lines show the times when the results of accessibility checks (see subsection 5.6.4) are cached or not. The overhead of the micro benchmarks range between factor 8 (25 vs. 3) and 29 (87 vs. 3) when caching is enabled. Note that passing a different parameter, or more than one parameter, might degrade the performance further.

Figure 5.12 shows the distribution of message sends across the five categories for key classes in the web server. On the left are measures for objects subject to dynamic checks. On the right are measures for primitive types (see subsection 5.7.2) not subject to dynamic checks. We can see that the majority of interactions between objects consists of message sends that can be optimized. Macro benchmarks using a mock socket independent of IO indicate a degradation of about factor 13 when caching of accessibility checks is enabled (204 vs. 2521). This overhead must be put in perspective with the overhead of the level of indirection (factor 8).

5.9 Discussion

Our experiments show that our approach can be put to work to better control information hiding. We found our approach flexible enough to accommodate an existing design, and at the same time constraining enough to highlight design anomalies. The system we studied was however small. We plan in future work to investigate how our approach scale for bigger systems. Here we sketch some improvements that our experiment suggests.

Flexibility with Impersonation Constructor methods and class-side utility methods are problematic since classes are owned by the `world`. For instance, invoking the copy constructor `OrderedCollection from: aCollection` implies that the reference `aCollection` is first transferred to the object `OrderedCollection`. The transfer might raise a crossing error.

A solution to this problem is to introduce impersonation of objects. When object *r* executes an impersonated method in response to a message from object *s*, the owner of *r* resolves temporary to the owner of *s* for the current execution. Following this strategy, classes can defined to always impersonate the sender of the messages they receive.

Improving Performance The dynamic checks follow the path between objects in the ownership tree. The identity of objects along the path is however not relevant, but only the classes of those objects matter to the dynamic checks. Computing information about the relative positioning of objects in the ownership tree is expensive. Our implementation caches the results of accessibility checks, but not the aliasing checks. We plan to investigate how this information could be effectively computed and cached in this case as well. The cache could be maintained per object, or per call site.

In tracing VM [64], traces are recorded and compiled to native code at run-time. Recorded traces are reused by speculating on branching and types. Guards in the traces validate this speculation, and if a guard fails, the trace exits. To support dynamic ownership, a tracing VM could speculate on the relative positions of objects in the ownership tree, in addition to branches and types. Checking the validity of references after ownership transfers would also benefit from VM support, since it is an expensive operation.

Note that dynamic ownership can be considered to be a special kind of contract. Filters could be turned off during execution of production systems for performance reasons. As long as the system does not use custom crossing handlers, they can be turned off as well (indeed, if the system uses custom crossing handlers, removing them might impact the application behavior).

Composition with Dynamic Topics It is easy to assign topics to a class as long as they are used in a specific context. If a class is reused in multiple contexts, it might be complicated to assign topics that satisfy all contexts. Let us consider Figure 5.3: the site enables a read-only view of the internal array. Let us imagine that the site uses another array internally to keep encryption keys. The second array will also be read-only since there is no way to distinguish from both contexts where arrays are used. The solution to this problem would be to have topics per objects: two instances of the same class could have different topics. We plan to achieve this with rewriting rules attached to individual objects: one array rewrites the topic `read-only` to `admin-read-only`, the other to `key-read-only`. The web site can decide to expose only `admin-read-only`. For composition, a rewriting rule attached to object o would impact o, but also objects owned by o.

5.10 Related work

We now compare more precisely our approach with other existing approaches briefly presented in Chapter 2.

The closest related work is Dynamic Ownership by Gordon and Noble [70], which itself built on previous concepts of Dynamic Alias Protection [122], Flexible Alias Protection [121] and Ownership Monitoring [89]. In contrast to Dynamic Ownership, filters enable objects to be accessed outside their owner's context via a possibly limited interface. Also, we not only check accessibility when messages are sent, but also aliasing of objects in return values and arguments. Flexible Alias Protection [121] enforces "external independence", a property which states that internal objects must not depend on mutable state of external objects: in Dynamic Ownership, invocations to external objects raise an exception if state is mutated or if a value is returned. In our approach, a topic can be used to categorize such legal methods, but it cannot raise an exception if a method is wrongly categorized.

Ownership Types Since the work by Clarke *et al.* [40] that introduced the owners-as-dominators model, many variants of static ownership types have been proposed. Similarly to our work, these approaches aim at relaxing the owner-as-dominator model to regain flexibility. In an extension of their previous work [39], Clark *et al.* enable dynamic aliases to expose internal objects such as iterators. Boyapati *et al.* used inner classes instead [23]. With Ownership Domains [4, 3], objects can be organized into various domains with different access constraints. Universe Types [114] similarly partition objects into universes and control references between them. Variant Ownership Types [101] parameterized types with an accessibility context in addition to the ownership context, thus giving more fine-grained control

over aliasing. In these mechanisms, a member of a class can be accessed if its type can be named. Filters and crossing handlers are dynamic. Gradual Ownership[147] combines static and dynamic typing. Dynamic checks are introduced for code that has not been statically typed. Other variants of ownership types exist which address other aspects of aliasing, for instance uniqueness of references [26, 129], thread-locality [173] or data transfer between actors [38]. Our examples of crossing handlers drew inspiration from the two latter works.

Limited Interfaces Variant Ownership Types [101] can specify whether references are writable or read-only. Universe Types [114] enforce the owner-as-modifier discipline, where read-only references across universes are allowed, but only the owner of an object can modify it. Our approach can encode the owner-as-modifier discipline by exposing a special `read-only` topic. Several languages can define write-once variables, *e.g.*, C++'s `const` and Java's `final` keywords. Used with references, `const` does not provide transitive read-only access. Schaerli *et al.* proposed encapsulation policies [145], which enable policies to be bound to references, but does not consider transitivity. There have been several proposals of type systems that support transitive read-only references [25, 175] (independently of ownership). Arnaud *et al.* proposed a variant of transitive read-only references for dynamic languages [11]. Filters are flexible and able to expose limited interfaces; an interface with only read-only methods is just a special case. In contrast to deep read-only references, read-only access is only transitive to objects within the context. It can be considered as a benefit, or a limitation depending on the context. Our approach is syntactic and it assumes that methods have been correctly categorized in the `read-only` topic by developers. Traditional access modifiers can be used to limit interfaces. Modifiers can implement class privacy or object privacy. Our approach implements object privacy, which was shown to be more intuitive [164]. Arbitrary accessibility rules can be easily implemented with techniques that reify message sends, such as composition filters [1]. The Law of Demeter [98] is a design principle which dissuades invocations to objects returned by previous invocations. Organizing the design in layers, where objects in a given layer can only call objects in the layer below, is a way to enforce the law. The law of Demeter, layers, and confinement with ownership are design principles that prevent interaction between distant entities.

Security Information hiding controls accesses from external to internal objects; secure programming controls accesses from an object to its external environment. Global namespaces compromise security since accesses to global namespaces cannot be controlled. In Java, access to the class names-

pace can be controlled with class loaders and security managers, which are mechanisms outside of the base language. In the object-capability [111] model, objects can only send messages to objects that have been obtained previously with message sends. In this model, global namespaces and reflection are loopholes. Filters can be used to limit access to external resources, since filters work in both in and out directions. To control interactions between modules, objects can be wrapped into membranes [111, 162, 44], which transitively impose revocability on all references exchanged via the membrane, both inward and outward. When the membrane is revoked, the wrapped module is guaranteed to become eligible for garbage collection; revoked references raise exceptions when used. Contexts resemble membranes that intercept outward transfer of references. Newspeak is a language that follows the object-capability model. In Newspeak, external dependencies must be provided when an object is created [29]; there is no global namespace, only nested virtual classes. Also, Newspeak decouples reflection from classes via mirrors [27]. Tribal Ownership [32] exploits class nesting to define an implicit ownership structure for objects. From the perspective of security, ownership transfer must be limited. Ownership transfer is hard to support in static type systems [115], but very natural in a dynamic approach.

5.11 Conclusions

Information hiding is an important principle to enable evolution, both at development time and run time. We have proposed an approach to improve information hiding by contextualizing the behavior of objects using active and structural variability. Objects are structured in an ownership tree and the behavior of an object varies dynamically depending on the sender of a message. Each object acts as a context that alters the interface of its children.

When a reference between two objects is about to cross a context inward, a crossing handler is triggered. By default the handler fails, which enforces the strict confinement of objects within their owner. However, the crossing handler can be customized to implement an alternative policy. For instance, an object can defensively copy its return value when it is accessed by an untrusted object, and return an internal reference when it is accessed by a trusted object.

Our approach improves the support of two modularity principles presented in Chapter 1:

- *Information hiding.* It scales information hiding from objects to aggregates.

- *Encapsulation.* Aliasing policies like defensive copying can be encapsulated.

We can draw the following conclusions about the practicality of our approach:

- *Adoption.* We found our approach flexible enough to accommodate an existing design, and at the same time constraining enough to highlight design anomalies. Object interactions tend to be local, *i.e.*, interactions between distant objects in the tree are rare. Filters fit well with this locality. Objects can be easily confined, not only within their direct context, but also within indirect contexts.

- *Flexibility.* The strength of our approach lies in the cumulative effect of filters. Filters of various contexts compose naturally to define the specific interface that is exposed to another object. Contexts can filter independent concerns. The behavior of an object depends on the context it is in, which can change over time. We can for instance partially simulate first-class states this way. Reflection can be scoped and limited with filters.

Dynamic ownership, delegation proxies, and active contexts demonstrate the benefits of embracing dynamism and contextualizing behavior. In the next chapter we discuss how the three mechanisms complement each others, and discuss the overall strengths and weaknesses of our approach to enable run-time changes.

6 Conclusions

We have argued throughout this dissertation that contextualizing behavior with active and structural variability improves the support for encapsulation, information hiding and late binding, which enables run-time changes. We have identified gaps in the support for run-time changes and devised contextual programming language features to address them, showing for each feature its respective benefits.

In this last chapter we review the three contributions together to provide a big picture of the progress that we achieved, but also of the drawbacks and open points that remain.

6.1 An Extended Toolbox

The three language features that we described extend the developer's toolbox with new tools:

- With active contexts, developers can roll out arbitrary behavioral changes and migrate the application state;
- With delegation proxies, developers can craft their own contextualization mechanism to adapt or evolve software at run time;
- With dynamic ownership, developers can enforce information hiding at the level of rigor they desire.

The three language features give novel options to developers to accommodate run-time changes at various levels of complexity. The three features

can be used in isolation, or together. Let us do a thought experiment[1] to recap how the features address the gaps that we identified (see Section 2.5), and to see how they mutually benefit each others. Let us consider for this purpose the small web server of subsection 5.3.4.

The web server is essentially a small piece of infrastructure that hosts request handlers. This is similar to a Java web server that hosts Java servlets[2]. Project stakeholders might be interested to support three forms of run-time changes.

1. The need to replace request handlers hosted by the server;
2. The need to change configuration options of request handlers (*e.g.*, log levels);
3. The need to evolve the server infrastructure itself.

With the new options available, developers can decide to address these requirements in the following way:

1. To ensure that the request handler can be replaced, the web server must not establish any incoming reference to its internals. If the request handler must return an object to the web server (*e.g.*, the HTTP response), the object must not have any dependencies to the request handler's internal. The request handler is an opaque aggregate that can be swapped when the system is quiescent. To enforce this discipline, developers can leverage dynamic ownership. Only objects that are not owned by the request handler can be returned to the web server. To perform the actual switch from one old request handler to the new one, they can use a regular proxy (or even better, a delegation proxy) that does the routing. This example shows that dynamic ownership helps partially cover the gap concerning advanced changes when the system is quiescent by increasing the granularity of classic objects.
2. To quickly change the configuration options of the request handlers, developers can leverage delegation proxies. They can implement a first-class context that intercepts certain method invocations, *e.g.*, invocation of the `log` method. There exists one context per request handler that represents its configuration. Before a request is processed by the request handler, the corresponding context is activated. Since the variation is scoped to the HTTP request, the configuration can change anytime. This example shows that delegation proxies cover the gap concerning simple changes when the system is busy.

[1] Unfortunately, the three features haven't been integrated in a single Smalltalk image at the moment.

[2] `http://jcp.org/aboutJava/communityprocess/final/jsr315/`

3. To enable the evolution of the server infrastructure at run time, developers can leverage active contexts. The developers can decide to roll out updates per request or per session without requiring quiescence. Additionally, using dynamic ownership, they can organize objects in a hierarchy that is useful to reason about state transformations and invariants. This example shows that active contexts cover the gap concerning advanced changes when the system is busy.

As this thought experiment shows, the features nicely complement each others. They make new design strategies possible. The above strategy is only one of them. For instance, developers could decide to use active contexts to update the request handlers and the server infrastructure.

With respect to the analogy of an extended toolbox, it is worth noting that the language features are really extensions[3]. If developers do not want to leverage the new features, they do not need to adapt their code.

6.2 Strengths and Weaknesses

A useful programming language feature provides an attractive tradeoff in terms of versatility, complexity, and performance. In other words: it should help in many situations, be easy to use, and have a small performance overhead.

Based on the experience gathered with our features, we now discuss the overall strengths and weaknesses of an approach that relies on contextual behavior.

Versatility. Active and structural variability helps in many situations. The primary goal that it enables is run-time changes, but the previous chapters have also illustrated other unexpected advantages. For instance, active and structural variability can address security and reliability (read-only references, object versioning, first-class states). Scoping variations to dynamic extents and structural extents seems relevant to software design in general.

Complexity. Our features are relatively easy to understand. However, they require some effort to be leveraged. For instance, in the cases of active contexts and delegation proxies, developers must identify where to place "switches" that activate variations. For dynamic ownership, developers must identify protocols and annotate methods. They require "object thinking" rather than "class thinking", which could be a challenge to some developers.

[3]Dynamic ownership defaults to a restrictive aliasing policy, but it can also be changed to default to a permissive policy that is backward compatible.

Performance. Active and structural variability inevitable entails a performance degradation since it increases late binding. With our features, the degradation ranged from factor 2 (dynamic updates) to factor 8 (dynamic ownership). This is the most serious drawback of our approach.

At the moment, our approach is attractive if the performance loss is acceptable. It is common for features that raise the abstraction level to originally entail a significant performance penalty. For instance, garbage collection was originally costly, but with optimized implementations tightly integrated at the virtual machine level the overhead was dramatically reduced. We believe our features could be made fast in the future.

6.3 Open Questions

This dissertation leaves two main questions unanswered with respect to contextualizing behavior with active and structural variability:

What Other Features Exist? We argue that the best way to enable runtime change is to support active and structural variability with first-class contexts. The three language features that we presented are only three possibilities within this design space. Additional features could be designed.

A particular feature that we envision is a generalization of dynamic ownership. In our current implementation, the behavior of an object varies in the sense that if the behavior is not visible, an invocation raises an error. Filters and topics are used to define whether an invocation results in a failure or not. With a generalized mechanism, filters and topics could be used to select one behavior within a set of variants. This way, the actual behavior of an object (as well as its interface) could be change via ownership changes.

Can Features Be Unified? The three language features that we presented are distinct mechanisms with their own specificities; they are not three applications of a general mechanism. The unification of active and structural variability in a single language feature remains an open challenge.

With a unification, a behavioral variation would be represented as an object that can either be attached to another object (structural variability), or activated during the evaluation of an expression (active variability). For instance, an "immutability" variation could be applied in a structural way or in an active way. How the behavioral variation should be expressed to make sense for both forms of variability is an open question.

Going further, it would be interesting to explore whether both forms of variability could be unified in a single theoretical framework, similarly to the approach of Tanter that unifies dynamic and static scope [156]. The open implementation for context-oriented layer composition [100] by Lincke *et al.* is also a step in that direction.

These two questions are exciting research tracks for the future.

Bibliography

[1] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting inter-object communications using composition filters. University of Twente, 1993.

[2] J. Aldrich. The power of interoperability: Why objects are inevitable. In *Onward!*, 2013.

[3] J. Aldrich and C. Chambers. Ownership domains: Separating aliasing policy from mechanism. In *ECOOP*, pages 1–25, 2004.

[4] J. Aldrich, C. Chambers, and D. Notkin. ArchJava: Connecting software architecture to implementation. In *ICSE'02: Proceedings of the 24th International Conference on Software Engineering*, pages 187–197, Orlando, FL, USA, 2002. ACM. ISBN 1-58113-472-X. doi: 10.1145/581339.581365.

[5] J. Andersson and T. Ritzau. Dynamic code update in jdrums. In *In Proceedings of the ICSE'00 Workshop on Software Engineering for Wearable and Pervasive Computing*, 2000.

[6] S. Apel, T. Leich, and G. Saake. Aspectual feature modules. *IEEE Trans. Softw. Eng.*, 34(2):162–180, Mar. 2008. ISSN 0098-5589. doi: 10.1109/TSE.2007.70770. URL `http://dx.doi.org/10.1109/TSE.2007.70770`.

[7] M. Appeltauer, R. Hirschfeld, M. Haupt, J. Lincke, and M. Perscheid. A comparison of context-oriented programming languages. In *COP '09: International Workshop on Context-Oriented Programming*, pages 1–6, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-538-3. doi: doi.acm.org/10.1145/1562112.1562118.

[8] M. Appeltauer, R. Hirschfeld, and H. Masuhara. Improving the development of context-dependent java applications with contextj. In *International Workshop on Context-Oriented Programming*, COP '09, pages 5:1–5:5, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-538-3. doi: 10.1145/1562112.1562117. URL `http://doi.acm.org/10.1145/1562112.1562117`.

[9] M. Appeltauer, R. Hirschfeld, and J. Lincke. Declarative layer composition with the jcop programming language. *Journal of Object Technology*, 12(2):4:1–37, June 2013. ISSN 1660-1769. doi: 10.5381/jot.2013.12.2.a4. URL `http://www.jot.fm/contents/issue_2013_06/article4.html`.

[10] J.-B. Arnaud. *Towards First Class References as a Security Infrastructure in Dynamically-Typed Languages*. PhD thesis, Université des Sciences et Technologies de Lille, 2013.

[11] J.-B. Arnaud, M. Denker, S. Ducasse, D. Pollet, A. Bergel, and M. Suen. Read-only execution for dynamic languages. In *Proceedings of the 48th International Conference on Objects, Models, Components, Patterns (TOOLS EUROPE'10)*. LNCS Springer Verlag, July 2010. URL `http://www.bergel.eu/download/papers/Berg10eReadOnly.pdf`.

[12] T. H. Austin, T. Disney, and C. Flanagan. Virtual values for language extension. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, volume 46 of *OOPSLA '11*, pages 921–938, New York, NY, USA, Oct. 2011. ACM. ISBN 978-1-4503-0940-0. doi: 10.1145/2048066.2048136. URL `http://dx.doi.org/10.1145/2048066.2048136`.

[13] D. Bacon, J. Bloch, J. Bogda, C. Click, P. Haahr, D. Lea, T. May, J.-W. Maessen, J. Manson, J. D. Mitchell, K. Nilsen, B. Pugh, and E. G. Sirer. The "double-checked locking is broken" declaration.

[14] K. Beck. Instance specific behavior: How and Why. *Smalltalk Report*, 2(7), Mar. 1993.

[15] A. Bergel, S. Ducasse, O. Nierstrasz, and R. Wuyts. Classboxes: Controlling visibility of class extensions. *Journal of Computer Languages, Systems and Structures*, 31(3-4):107–126, Dec. 2005. doi: 10.1016/j.cl.2004.11.002. URL `http://scg.unibe.ch/archive/papers/Berg05aclassboxesJournal.pdf`.

[16] A. Bergel, S. Ducasse, O. Nierstrasz, and R. Wuyts. Stateful traits and their formalization. *Journal of Computer Languages, Systems and Structures*, 34(2-3):83–108, 2008. ISSN 1477-8424. doi: 10.1016/j.cl.2007.05.003. URL `http://scg.unibe.ch/archive/papers/Berg08eStatefulTraits.pdf`.

[17] L. Bettini, S. Capecchi, and E. Giachino. Featherweight wrap java. In *Proc. of SAC (The 22nd Annual ACM Symposium on Applied Computing), Special Track on Object-Oriented Programming Languages and Systems (OOPS)*, pages 1094–1100. ACM Press, 2007. URL `http://gdn.dsi.unifi.it/phpbibliography/files/wrapjava.pdf`.

[18] L. Bettini, S. Capecchi, and F. Damiani. A mechanism for flexible dynamic trait replacement. In *Proceedings of the 11th International Workshop on Formal Techniques for Java-like Programs*, FTfJP '09, pages 9:1–9:7, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-540-6. doi: 10.1145/1557898.1557907. URL `http://doi.acm.org/10.1145/1557898.1557907`.

[19] L. Bettini, S. Capecchi, and F. Damiani. On flexible dynamic trait replacement for java-like languages. *Science of Computer Programming*, 2011. doi: 10.1016/j.scico.2012.11.003. URL `http://www.di.unito.it/~damiani/papers/SUBscp1.pdf`.

[20] K. Bierhoff and J. Aldrich. Lightweight object specification with typestates. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 217–226, New York, NY, USA, 2005. ACM. ISBN 1-59593-014-0. doi: 10.1145/1081706.1081741. URL `http://doi.acm.org/10.1145/1081706.1081741`.

[21] J. Bloch. *Effective Java (2nd Edition) (The Java Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2 edition, 2008. ISBN 0321356683, 9780321356680.

[22] A. Bohannon, B. C. Pierce, and J. A. Vaughan. Relational lenses: a language for updatable views. In *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '06, pages 338–347, New York, NY, USA, 2006. ACM. ISBN 1-59593-318-2. doi: 10.1145/1142351.1142399. URL `10.1145/1142351.1142399`.

[23] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *Principles of Programming Languages (POPL'03)*, pages 213–223. ACM Press, 2003. ISBN 1-58113-628-5. doi: 10.1145/604131.604156.

[24] C. Boyapati, B. Liskov, L. Shrira, C.-H. Moh, and S. Richman. Lazy modular upgrades in persistent object stores. *SIGPLAN Not.*, 38(11): 403–417, 2003. ISSN 0362-1340. doi: 10.1145/949343.949341. URL `10.1145/949343.949341`.

[25] J. Boyland. Why we should not add readonly to Java (yet. In *In FTfJP*, pages 5–29, 2005.

[26] J. Boyland, J. Noble, and W. Retert. Capabilities for aliasing: A generalisation of uniqueness and read-only. In J. L. Knudsen, editor, *Pro-*

ceesings ECOOP 2001, number 2072 in Lecture Notes in Computer Science, pages 2–27. Springer, June 2001.

[27] G. Bracha. Pluggable type systems, Oct. 2004. URL `http://prog.vub.ac.be/~wdmeuter/RDL04/papers/Bracha.pdf`. OOPSLA Workshop on Revival of Dynamic Languages.

[28] G. Bracha and D. Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04), ACM SIGPLAN Notices*, pages 331–344, New York, NY, USA, 2004. ACM Press. doi: 10.1145/1028976.1029004. URL `http://bracha.org/mirrors.pdf`.

[29] G. Bracha, P. von der Ahé, V. Bykov, Y. Kashai, W. Maddox, and E. Miranda. Modules as objects in Newspeak. In *ECOOP'10: Proceedings of the 24th European Conference on Object-Oriented Programming*, ECOOP'10, pages 405–428, Berlin, Heidelberg, June 2010. Springer-Verlag. ISBN 3-642-14106-4, 978-3-642-14106-5. doi: 10.1007/978-3-642-14107-2_20. URL `http://bracha.org/newspeak-modules.pdf`.

[30] K. B. Bruce, M. Odersky, and P. Wadler. A statically safe alternative to virtual types. In *Proceedings ECOOP '98*, pages 523–549. Springer-Verlag, 1998. ISBN 3-540-64737-6.

[31] M. Büchi and W. Weck. Generic wrappers. In E. Bertino, editor, *ECOOP 2000 - Object-Oriented Programming, 14th European Conference, Sophia Antipolis and Cannes, France, June 12-16, 2000, Proceedings*, volume 1850 of *Lecture Notes in Computer Science*, pages 201–225. Springer, 2000. ISBN 3-540-67660-0.

[32] N. Cameron, J. Noble, and T. Wrigstad. Tribal ownership. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 618–633, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0203-6. doi: doi.acm.org/10.1145/1869459.1869510. URL `doi.acm.org/10.1145/1869459.1869510`.

[33] S. Cech Previtali and T. R. Gross. Aspect-based dynamic software updating: a model and its empirical evaluation. In *Proceedings of the tenth international conference on Aspect-oriented software development*, AOSD '11, pages 105–116, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0605-8. doi: 10.1145/1960275.1960289.

[34] H. Chen, J. Yu, C. Hang, B. Zang, and P.-C. Yew. Dynamic software updating using a relaxed consistency model. *IEEE Trans. Softw. Eng.*,

37(5):679–694, Sept. 2011. ISSN 0098-5589. doi: 10.1109/TSE.2010.79. URL http://dx.doi.org/10.1109/TSE.2010.79.

[35] S. Chiba. Load-time structural reflection in Java. In *Proceedings of ECOOP 2000*, volume 1850 of *LNCS*, pages 313–336, 2000.

[36] S. Chiba, G. Kiczales, and J. Lamping. Avoiding confusion in metacircularity: The meta-helix. In K. Futatsugi and S. Matsuoka, editors, *Proceedings of ISOTAS '96*, volume 1049 of *Lecture Notes in Computer Science*, pages 157–172. Springer, 1996. ISBN 3-540-60954-7. doi: 10.1007/3-540-60954-7_49. URL http://www2.parc.com/csl/groups/sda/publications/papers/Chiba-ISOTAS96/for-web.pdf.

[37] A. Choi. Online application upgrade using edition-based redefinition. In *Proceedings of the 2nd International Workshop on Hot Topics in Software Upgrades*, HotSWUp '09, pages 4:1–4:5, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-723-3. doi: 10.1145/1656437.1656443.

[38] T. Clark, P. Sammut, and J. Willans. *Superlanguages, Developing Languages and Applications with XMF*, volume First Edition. Ceteva, 2008. URL http://www.ceteva.com/docs/Superlanguages.pdf.

[39] D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. *SIGPLAN Not.*, 37(11):292–310, Nov. 2002. ISSN 0362-1340. doi: 10.1145/583854.582447. URL http://doi.acm.org/10.1145/583854.582447.

[40] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Proceedings OOPSLA '98*, pages 48–64. ACM Press, 1998. ISBN 1-58113-005-8. doi: 10.1145/286936.286947.

[41] M. Conrad, T. French, , and C. Maple1. Object shadowing - a key concept for a modern programming language. In *2nd Workshop on Object-Oriented Language Engineering for the Post-Java Era: Back to Dynamicity*, 2004.

[42] P. Costanza and R. Hirschfeld. Language constructs for context-oriented programming: An overview of ContextL. In *Proceedings of the Dynamic Languages Symposium (DLS) '05, co-organized with OOPSLA'05*, pages 1–10, New York, NY, USA, Oct. 2005. ACM. ISBN 1-59593-283-6. doi: 10.1145/1146841.1146842. URL http://p-cos.net/documents/contextl-overview.pdf.

[43] P. Costanza, O. Stiemerling, and A. Cremers. Object identity and dynamic recomposition of components. In *Technology of Object-Oriented Languages and Systems, 2001. TOOLS 38. Proceedings*, pages 51 –65, 2001. doi: 10.1109/TOOLS.2001.911755.

[44] T. V. Cutsem and M. S. Miller. On the design of the ECMAScript reflection api. Technical report, Vrije Universiteit Brussel, 2012.

[45] T. V. Cutsem and M. S. Miller. Trustworthy proxies: Virtualizing objects with invariants. In *ECOOP 2013*, 2013. URL `http://soft.vub.ac.be/Publications/2013/vub-soft-tr-13-03.pdf`.

[46] F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. Re-classification and multi-threading: Ficklemt. In *Proceedings of the 2004 ACM symposium on Applied computing*, SAC '04, pages 1297–1304, New York, NY, USA, 2004. ACM. ISBN 1-58113-812-1. doi: 10.1145/967900.968163.

[47] R. DeLine and M. Fähndrich. Typestates for objects. In *ECOOP'04*, pages 465–490, 2004.

[48] L. G. DeMichiel and R. P. Gabriel. The Common Lisp object system: An overview. In J. Bézivin, J.-M. Hullot, P. Cointe, and H. Lieberman, editors, *Proceedings ECOOP '87*, volume 276 of *LNCS*, pages 151–170, Paris, France, June 1987. Springer-Verlag.

[49] M. Denker, S. Ducasse, A. Lienhard, and P. Marschall. Sub-method reflection. In *Journal of Object Technology, Special Issue. Proceedings of TOOLS Europe 2007*, volume 6/9, pages 231–251. ETH, Oct. 2007. doi: 10.5381/jot.2007.6.9.a14. URL `http://www.jot.fm/contents/issue_2007_10/paper14.html`.

[50] M. Denker, M. Suen, and S. Ducasse. The meta in meta-object architectures. In *Proceedings of TOOLS EUROPE 2008*, volume 11 of *LNBIP*, pages 218–237. Springer-Verlag, 2008. doi: 10.1007/978-3-540-69824-1_13. URL `http://scg.unibe.ch/archive/papers/Denk08bMetaContextLNBIP.pdf`.

[51] P. Deutsch. Building control structures in smalltalk-80. *Byte*, 6(8): 322–346, aug 1981.

[52] M. Dmitriev. Towards flexible and safe technology for runtime evolution of Java language applications. In *Proceedings of the Workshop on Engineering Complex Object-Oriented Systems for Evolution, in association with OOPSLA 2001*, Oct. 2001.

[53] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. Fickle: Dynamic object re-classification. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 130–149, London, UK, UK, 2001. Springer-Verlag. ISBN 3-540-42206-4. URL `http://portal.acm.org/citation.cfm?id=646158.680007`.

[54] S. Ducasse. Evaluating message passing control techniques in Smalltalk. *Journal of Object-Oriented Programming (JOOP)*, 12(6): 39–44, June 1999. URL `http://scg.unibe.ch/archive/papers/Duca99aMsgPassingControl.pdf`.

[55] D. Duggan. Type-based hot swapping of running modules (extended abstract). In *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, ICFP '01, pages 62–73, New York, NY, USA, 2001. ACM. ISBN 1-58113-415-0. doi: 10.1145/507635.507645. URL `http://doi.acm.org/10.1145/507635.507645`.

[56] E. Ernst. Family polymorphism. In J. L. Knudsen, editor, *ECOOP 2001*, number 2072 in LNCS, pages 303–326. Springer Verlag, 2001.

[57] E. Ernst. Higher-order hierarchies. In *Proceedings European Conference on Object-Oriented Programming (ECOOP 2003)*, LNCS, pages 303–329, Heidelberg, July 2003. Springer Verlag.

[58] M. Ernst, C. Kaplan, and C. Chambers. Predicate dispatching: A unified theory of dispatch. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 186–211, London, UK, 1998. Springer-Verlag. ISBN 3-540-64737-6. URL `http://portal.acm.org/citation.cfm?id=646155.679688`.

[59] P. Eugster. Uniform proxies for java. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 139–152, New York, NY, USA, 2006. ACM. ISBN 1-59593-348-4. doi: 10.1145/1167473.1167485. URL `http://doi.acm.org/10.1145/1167473.1167485`.

[60] P. T. Eugster, P. A. Felber, R. Guerraoui, and A. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003. doi: 10.1145/857076.857078. URL `http://portal.acm.org/citation.cfm?id=857078`.

[61] E. D. Falkenberg. Concepts for modelling information. In *IFIP Working Conference on Modelling in Data Base Management Systems*, pages 95–109, 1976.

[62] J. N. Foster, A. Pilkiewicz, and B. C. Pierce. Quotient lenses. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 383–396, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-919-7. doi: 10.1145/1411204.1411257.

[63] Y. Futamura. Partial evaluation of computation process: An approach to a compiler-compiler. *Higher Order Symbol. Comput.*, 12(4): 381–391, 1999. ISSN 1388-3690. doi: 10.1023/A:1010095604496.

[64] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 465–478, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1. doi: 10.1145/1542476.1542528. URL `http://doi.acm.org/10.1145/1542476.1542528`.

[65] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional, Reading, Mass., 1995. ISBN 978-0201633610.

[66] Gemstone. Gemstone/s programming guide, 2007. URL `http://seaside.gemstone.com/docs/GS64-ProgGuide-2.2.pdf`.

[67] A. Goldberg and D. Robson. *Smalltalk 80: the Language and its Implementation*. Addison Wesley, Reading, Mass., May 1983. ISBN 0-201-13688-0. URL `http://stephane.ducasse.free.fr/FreeBooks/BlueBook/Bluebook.pdf`.

[68] S. González, K. Mens, and A. Cádiz. Context-Oriented Programming with the Ambient Object System. *Journal of Universal Computer Science*, 14(20):3307–3332, 2008. ISSN 0948-6968.

[69] S. González, K. Mens, M. Colacioiu, and W. Cazzola. Context traits: dynamic behaviour adaptation through run-time trait recomposition. In *Proceedings of the 12th annual international conference on Aspect-oriented software development*, AOSD '13, pages 209–220, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1766-5. doi: 10.1145/2451436.2451461. URL `http://doi.acm.org/10.1145/2451436.2451461`.

[70] D. Gordon and J. Noble. Dynamic ownership in a dynamic language. In *DLS '07: Proceedings of the 2007 symposium on Dynamic languages*, pages 41–52, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-868-8.

[71] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification (Third Edition)*. Addison Wesley, 2005. ISBN 0-321-24678-0.

[72] A. R. Gregersen and B. N. Jorgensen. Dynamic update of Java applications — balancing change flexibility vs programming transparency. *J. Softw. Maint. Evol.*, 21:81–112, mar 2009. ISSN 1532-060X. doi: 10.1002/smr.v21:2. URL `http://portal.acm.org/citation.cfm?id=1526497.1526501`.

[73] J. Gustavsson. A classification of unanticipated runtime software changes in java. In *Proceedings of the International Conference on Software Maintenance*, ICSM '03, pages 4–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1905-9. URL http://dl.acm.org/citation.cfm?id=942800.943566.

[74] N. B. Harrison. Design patterns in communications software. In L. Rising, editor, *Design patterns in communications software*, chapter Patterns for logging diagnostic messages, pages 173–185. Cambridge University Press, New York, NY, USA, 2001. ISBN 0-521-79040-9. URL http://dl.acm.org/citation.cfm?id=566110.566121.

[75] W. Harrison and H. Ossher. Subject-oriented programming (a critique of pure objects). In *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, volume 28, pages 411–428, Oct. 1993. doi: 10.1145/165854.165932.

[76] C. M. Hayden, E. K. Smith, M. Denchev, M. Hicks, and J. S. Foster. Kitsune: efficient, general-purpose dynamic software updating for c, 2012. URL http://doi.acm.org/10.1145/2384616.2384635.

[77] S. Herrmann. A precise model for contextual roles: The programming language objectteams/java. *Appl. Ontol.*, 2(2):181–207, Apr. 2007. ISSN 1570-5838. URL http://dl.acm.org/citation.cfm?id=1412401.1412405.

[78] S. Herrmann. Demystifying object schizophrenia. In *Proceedings of the 4th Workshop on MechAnisms for SPEcialization, Generalization and inHerItance*, MASPEGHI '10, pages 2:1–2:5, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0535-8. doi: 10.1145/1929999.1930001. URL http://doi.acm.org/10.1145/1929999.1930001.

[79] M. Hicks and S. Nettles. Dynamic software updating. *ACM Transactions on Programming Languages and Systems*, 27(6):1049–1096, nov 2005. doi: 10.1145/1108970.1108971.

[80] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7 (3), Mar. 2008. doi: 10.5381/jot.2008.7.3.a4. URL http://www.jot.fm/contents/issue_2008_03/article4.htmlhttp://www.jot.fm/issues/issue_2008_03/article4.pdf.

[81] G. Hjálmtýsson and R. Gray. Dynamic C++ classes: a lightweight mechanism to update code in a running program. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '98, pages 6–6, Berkeley, CA, USA, 1998. USENIX Association. URL http://portal.acm.org/citation.cfm?id=1268256.1268262.

[82] M. Hofmann, B. Pierce, and D. Wagner. Symmetric lenses. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 371–384, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0490-0. doi: 10.1145/1926385.1926428. URL 10.1145/1926385.1926428.

[83] J. Hogg, D. Lea, A. Wills, D. deChampeaux, and R. Holt. The Geneva convention on the treatment of object aliasing. *SIGPLAN OOPS Mess.*, 3(2):11–16, 1992. ISSN 1055-6400. doi: 10.1145/130943.130947.

[84] M. D. Ingesman and E. Ernst. Lifted java: A minimal calculus for translation polymorphism. In *TOOLS (49)*, pages 179–193, 2011.

[85] E. B. Johnsen, M. Kyas, and I. C. Yu. Dynamic classes: Modular asynchronous evolution of distributed concurrent objects. In *Proceedings of the 2nd World Congress on Formal Methods*, FM '09, pages 596–611, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-05088-6. doi: /10.1007/978-3-642-05089-3_38. URL http://dx.doi.org/10.1007/978-3-642-05089-3_38.

[86] J. Kabanov. Jrebel tool demo. *Electron. Notes Theor. Comput. Sci.*, 264: 51–57, feb 2011. ISSN 1571-0661. doi: 10.1016/j.entcs.2011.02.005.

[87] J. Kabanov and V. Vene. A thousand years of productivity: the jrebel story. *Software: Practice and Experience*, pages n/a–n/a, 2012. ISSN 1097-024X. doi: 10.1002/spe.2158. URL http://dx.doi.org/10.1002/spe.2158.

[88] A. M. Keller. Algorithms for translating view updates to database updates for views involving selections, projections, and joins. In *Proceedings of the fourth ACM SIGACT-SIGMOD symposium on Principles of database systems*, PODS '85, pages 154–163, New York, NY, USA, 1985. ACM. ISBN 0-89791-153-9. doi: 10.1145/325405.325423. URL http://doi.acm.org/10.1145/325405.325423.

[89] S. Kent and I. Maung. Encapsulation and aggregation. In *In TOOLS Pacific 18*. Prentice Hall, 1995.

[90] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991. ISBN 0-262-11158-6.

[91] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *ECOOP'97: Proceedings of the 11th European Conference on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242, Jyvaskyla, Finland, June 1997. Springer-Verlag. doi: 10.1007/BFb0053381.

[92] G. Kniesel. Type-safe delegation for run-time component adaptation. In R. Guerraoui, editor, *Proceedings ECOOP '99*, volume 1628 of *LNCS*, pages 351–366, Lisbon, Portugal, June 1999. Springer-Verlag.

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

[94] B. B. Kristensen. Object-oriented modeling with roles. In J. Murphy and B. Stone, editors, *Proceedings of the 2nd International Conference on Object-Oriented Information Systems*, pages 57–71, London , UK, 1995. Springer-Verlag.

[95] K. R. Leino, P. Müller, and A. Wallenburg. Flexible immutability with frozen objects. In *VSTTE '08: Proceedings of the 2nd international conference on Verified Software: Theories, Tools, Experiments*, pages 192–208, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-87872-8. doi: dx.doi.org/10.1007/978-3-540-87873-5_17.

[96] P. Li, N. Cameron, and J. Noble. Cloning in ownership. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH '11, pages 63–66, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0942-4. doi: 10.1145/2048147.2048175. URL `http://doi.acm.org/10.1145/2048147.2048175`.

[97] S. Liang and G. Bracha. Dynamic class loading in the Java virtual machine. In *Proceedings of OOPSLA '98, ACM SIGPLAN Notices*, pages 36–44, 1998. doi: 10.1145/286936.286945.

[98] K. J. Lieberherr and A. J. Riel. Contributions to teaching object oriented design and programming. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, volume 24, pages 11–22, Oct. 1989.

[99] H. Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, volume 21, pages 214–223, Nov. 1986. doi: 10.1145/960112.28718. URL `http://web.media.mit.edu/~lieber/Lieberary/OOP/Delegation/Delegation.htmlhttp://reference.kfupm.edu.sa/content/u/s/using_prototypical_objects_to_implement__76339.pdf`.

[100] J. Lincke, M. Appeltauer, B. Steinert, and R. Hirschfeld. An open implementation for context-oriented layer composition in contextjs. *Sci. Comput. Program.*, 76:1194–1209, Dec. 2011. ISSN 0167-6423. doi: 10.1016/j.scico.2010.11.013. URL `http://dx.doi.org/10.1016/j.scico.2010.11.013`.

[101] Y. Lu and J. Potter. On ownership and accessibility. In *In ECOOP'06, volume 4067 of LNCS*, pages 99–123. Springer-Verlag, 2006.

[102] O. L. Madsen and B. Moller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, volume 24, pages 397–406, Oct. 1989.

[103] S. Magill, M. Hicks, S. Subramanian, and K. S. McKinley. Automating object transformations for dynamic software updating. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12, pages 265–280, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1561-6. doi: 10.1145/2384616.2384636. URL `http://doi.acm.org/10.1145/2384616.2384636`.

[104] K. Makris and R. A. Bazzi. Immediate multi-threaded dynamic software updates using stack reconstruction. In *Proceedings of the 2009 conference on USENIX Annual technical conference*, USENIX'09, pages 31–31, Berkeley, CA, USA, 2009. USENIX Association. URL `http://portal.acm.org/citation.cfm?id=1855807.1855838`.

[105] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. Barnes. Runtime support for type-safe dynamic Java classes. In *Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 337–361. Springer-Verlag, 2000. ISBN 3-540-67660-0. doi: 10.1007/3-540-45102-1_17.

[106] M. Martinez Peck, N. Bouraqadi, M. Denker, S. Ducasse, and L. Fabresse. Efficient proxies in smalltalk. In *Proceedings of the International Workshop on Smalltalk Technologies*, IWST '11, pages 8:1–8:16, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1050-5. doi: 10.1145/2166929.2166937. URL `http://doi.acm.org/10.1145/2166929.2166937`.

[107] 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, pages 46–60, Berlin, Heidelberg, 2003. Springer-Verlag. ISBN 3-540-00904-3. URL `http://dl.acm.org/citation.cfm?id=1765931.1765937`.

[108] Y. Matsumoto. *Ruby in a Nutshell*. O'Reilly, 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2001. ISBN 0596002149.

[109] Microsoft. Microsoft .net dynamic language runtime. URL `http://dlr.codeplex.com/`.

[110] M. S. Miller and J. S. Shapiro. Paradigm regained: Abstraction mechanisms for access control. In *Proceedings of the Eigth Asian Computing Science Conference*, pages 224–242, 2003.

[111] T. Millstein, M. Reay, and C. Chambers. Relaxed multijava: balancing extensibility and modular typechecking. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 224–240. ACM Press, 2003. ISBN 1-58113-712-5. doi: 10.1145/949305.949325.

[112] L. Moreau. A syntactic theory of dynamic binding. *Higher Order Symbol. Comput.*, 11(3):233–279, Oct. 1998. ISSN 1388-3690. doi: 10.1023/A:1010087314987. URL `http://dx.doi.org/10.1023/A:1010087314987`.

[113] W. B. Mugridge, J. Hamer, and J. G. Hosking. Multi-methods in a statically-typed programming language. In P. America, editor, *Proceedings ECOOP '91*, volume 512 of *LNCS*, pages 307–324, Geneva, Switzerland, July 1991. Springer-Verlag.

[114] P. Müller and A. Poetzsch-Heffter. Universes: A type system for controlling representation exposure. In A. Poetzsch-Heffter and J. Meyer, editors, *Programming Languages and Fundamentals of Programming*. Fernuniversität Hagen, 1999.

[115] P. Müller and A. Rudich. Ownership transfer in universe types. *SIGPLAN Not.*, 42(10):461–478, Oct. 2007. ISSN 0362-1340. doi: 10.1145/1297105.1297061. URL `http://doi.acm.org/10.1145/1297105.1297061`.

[116] I. Neamtiu and M. Hicks. Safe and timely updates to multi-threaded programs. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 13–24, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1. doi: 10.1145/1543135.1542479.

[117] I. Neamtiu, J. S. Foster, and M. Hicks. Understanding source code evolution using abstract syntax tree matching. In *Proceedings of the 2005 international workshop on Mining software repositories*, volume 30, pages 1–5, New York, NY, USA, may 2005. ACM. doi: doi.acm.org/10.1145/1082983.1083143. URL `http://doi.acm.org/10.1145/1082983.1083143`.

[118] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for C. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI

'06, pages 72–83, New York, NY, USA, 2006. ACM. ISBN 1-59593-320-4. doi: 10.1145/1133981.1133991. URL http://doi.acm.org/10.1145/1133981.1133991.

[119] O. Nierstrasz and F. Achermann. A calculus for modeling software components. In S. G. F. S. De Boer, M. M. Bonsangue and W.-P. de Roever, editors, *FMCO 2002 Proceedings*, volume 2852 of *LNCS*, pages 339–360. Springer-Verlag, 2003. ISBN 978-3-540-20303-2. doi: 10.1007/b14033. URL http://scg.unibe.ch/archive/papers/Nier03cPiccolaCalculus.pdf.

[120] J. Noble. Iterators and encapsulation. In *Proceedings of TOOLS '00*, page 431ff, June 2000.

[121] J. Noble, J. Potter, and J. Vitek. Flexible alias protection. In E. Jul, editor, *Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP'98)*, volume 1445 of *LNCS*, pages 158–185, Brussels, Belgium, July 1998. Springer-Verlag. ISBN 3-540-64737-6.

[122] J. Noble, A. Taivalsaari, and I. Moore. *Prototype-Based Programming*. Springer, 1999.

[123] N. Nystrom, X. Qi, and A. C. Myers. J&: nested intersection for scalable software composition. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 21–36, New York, NY, USA, 2006. ACM. ISBN 1-59593-348-4. doi: 10.1145/1167473.1167476.

[124] M. Odersky and M. Zenger. Scalable component abstractions. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 41–57, New York, NY, USA, 2005. ACM. ISBN 1-59593-031-0. doi: 10.1145/1094811.1094815. URL http://doi.acm.org/10.1145/1094811.1094815.

[125] Oracle. Jsr 292: Supporting dynamically typed languages on the java platform. URL http://jcp.org/en/jsr/detail?id=292.

[126] M. Oriol. Primitives for the dynamic evolution of component-based applications. In *Proceedings of the 2007 ACM symposium on Applied computing*, SAC '07, pages 1122–1123, New York, NY, USA, 2007. ACM. ISBN 1-59593-480-4. doi: 10.1145/1244002.1244246. URL http://doi.acm.org/10.1145/1244002.1244246.

[127] A. Orso, A. Rao, and M. J. Harrold. A Technique for Dynamic Updating of Java Software. *Software Maintenance, IEEE International Conference on*, 0:0649+, 2002. doi: 10.1109/ICSM.2002.1167829. URL http://dx.doi.org/10.1109/ICSM.2002.1167829.

[128] K. Ostermann. Dynamically composable collaborations with delegation layers. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, ECOOP '02, pages 89–110, London, UK, 2002. Springer-Verlag. ISBN 3-540-43759-2. URL http://portal.acm.org/citation.cfm?id=646159.680026.

[129] J. Östlund, T. Wrigstad, D. Clarke, and B. Åkerblom. Ownership, uniqueness, and immutability. In R. F. Paige and B. Meyer, editors, *Objects, Components, Models and Patterns, 46th International Conference, TOOLS EUROPE 2008*, volume 11 of *Lecture Notes in Business Information Processing*, pages 178–197. Springer, 2008. ISBN 978-3-540-69823-4. doi: dx.doi.org/10.1007/978-3-540-69824-1_11.

[130] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *CACM*, 15(12):1053–1058, Dec. 1972. doi: 10.1145/361598.361623. URL http://www.cs.umd.edu/class/spring2003/cmsc838p/Design/criteria.pdf.

[131] B. Pernici. Objects with roles. Object oriented development, Centre Universitaire d'Informatique, University of Geneva, July 1989.

[132] M. Piccioni, M. Orioly, B. Meyer, and T. Schneider. An ide-based, integrated solution to schema evolution of object-oriented software. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 650–654, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3891-4. doi: 10.1109/ASE.2009.100. URL http://dx.doi.org/10.1109/ASE.2009.100.

[133] L. Pina and J. P. Cachopo. Atomic dynamic upgrades using software transactional memory. In *HotSWUp*, pages 21–25, 2012. doi: dx.doi.org/10.1109/HotSWUp.2012.6226612.

[134] F. Pluquet, S. Langerman, and R. Wuyts. Executing code in the past: efficient in-memory object graph versioning. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 391–408, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-766-0. doi: doi.acm.org/10.1145/1640089.1640118. URL http://doi.acm.org/10.1145/1640089.1640118.

[135] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In *Proceedings of the 1st international conference on Aspect-oriented software development*, pages 141–147. ACM Press, 2002. ISBN 1-58113-469-X. doi: 10.1145/508386.508404.

[136] D. R. Prasanna. *Dependency Injection*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2009. ISBN 193398855X, 9781933988559.

[137] M. Pukall, C. Kästner, and G. Saake. Towards unanticipated runtime adaptation of java applications. In *APSEC '08: Proceedings of the 2008 15th Asia-Pacific Software Engineering Conference*, pages 85–92, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3446-6. doi: 10.1109/APSEC.2008.66.

[138] M. Pukall, C. Kästner, W. Cazzola, S. Götz, A. Grebhahn, R. Schröter, and G. Saake. JavAdaptor—flexible runtime updates of Java applications. *Software: Practice and Experience*, pages n/a–n/a, 2012. ISSN 1097-024X. doi: 10.1002/spe.2107. URL `http://dx.doi.org/10.1002/spe.2107`.

[139] J. Ressia. *Object-Centric Reflection*. Phd thesis, University of Bern, Oct. 2012. URL `http://scg.unibe.ch/archive/phd/ressia-phd.pdf`.

[140] J. Ressia, T. Gîrba, O. Nierstrasz, F. Perin, and L. Renggli. Talents: Dynamically composable units of reuse. In *Proceedings of International Workshop on Smalltalk Technologies (IWST 2011)*, pages 109–118, 2011. doi: 10.1145/2166929.2166940. URL `http://scg.unibe.ch/archive/papers/Ress11a-Talents.pdf`.

[141] J. Ressia, T. Gîrba, O. Nierstrasz, F. Perin, and L. Renggli. Talents: an environment for dynamically composing units of reuse. *Software: Practice and Experience*, 2012. ISSN 1097-024X. doi: 10.1002/spe.2160. URL `http://scg.unibe.ch/archive/papers/Ress12eTalentsSPE.pdf`.

[142] F. Rivard. Smalltalk: a reflective language. In *Proceedings of REFLECTION '96*, pages 21–38, Apr. 1996.

[143] G. Salvaneschi, C. Ghezzi, and M. Pradella. Context-oriented programming: A software engineering perspective. *J. Syst. Softw.*, 85(8): 1801–1817, Aug. 2012. ISSN 0164-1212. doi: 10.1016/j.jss.2012.03.024. URL `http://dx.doi.org/10.1016/j.jss.2012.03.024`.

[144] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable units of behavior. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'03)*, volume 2743 of *LNCS*, pages 248–274, Berlin Heidelberg, July 2003. Springer Verlag. ISBN 978-3-540-40531-3. doi: 10.1007/b11832. URL `http://scg.unibe.ch/archive/papers/Scha03aTraits.pdf`.

[145] N. Schärli, S. Ducasse, O. Nierstrasz, and R. Wuyts. Composable encapsulation policies. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'04)*, volume 3086 of *LNCS*, pages 26–50. Springer Verlag, June 2004. ISBN 978-3-540-22159-3. doi: 10.1007/b98195. URL `http://scg.unibe.ch/archive/papers/Scha04aEncapsulationPolicies.pdf`.

[146] M. Seemann. *Dependency Injection in .NET*. Manning Publications, sep 2011.

[147] I. Sergey and D. Clarke. Gradual ownership types. In *ESOP*, pages 579–599, 2012.

[148] M. Serrano. Wide classes. In R. Guerraoui, editor, *Proceedings ECOOP '99*, volume 1628 of *LNCS*, pages 391–415, Lisbon, Portugal, June 1999. Springer-Verlag. URL `http://www.ifs.uni-linz.ac.at/~ecoop/cd/papers/1628/16280391.pdf`.

[149] C. Smith and S. Drossopoulou. Chai: Typed traits in Java. In *Proceedings ECOOP 2005*, 2005.

[150] R. B. Smith and D. Ungar. A simple and unifying approach to subjective objects. *TAPOS special issue on Subjectivity in Object-Oriented Systems*, 2(3):161–178, Dec. 1996. doi: 10.1002/(SICI)1096-9942(1996)2:3%3C161::AID-TAPO3%3E3.0.CO;2-Z. URL `http://www.mip.sdu.dk/~bnj/library/Us_Ungar.pdf`.

[151] F. Steimann. On the representation of roles in object-oriented and conceptual modelling. *Data Knowl. Eng.*, 35(1):83–106, Oct. 2000. ISSN 0169-023X. doi: 10.1016/S0169-023X(00)00023-9. URL `http://dx.doi.org/10.1016/S0169-023X(00)00023-9`.

[152] T. S. Strickland, S. Tobin-Hochstadt, R. B. Findler, and M. Flatt. Chaperones and impersonators: Run-time support for reasonable interposition. In *OOPSLA '12: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, Oct. 2012. To appear.

[153] S. Subramanian, M. Hicks, and K. S. McKinley. Dynamic software updates: a VM-centric approach. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 1–12, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1. doi: 10.1145/1542476.1542478. URL `http://doi.acm.org/10.1145/1542476.1542478`.

[154] J. Sunshine, K. Naden, S. Stork, J. Aldrich, and E. Tanter. First-class state change in Plaid. In *Proceedings of the 2011 ACM international*

conference on Object oriented programming systems languages and applications, OOPSLA '11, pages 713–732, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0940-0. doi: 10.1145/2048066.2048122. URL http://doi.acm.org/10.1145/2048066.2048122.

[155] É. Tanter. Expressive scoping of dynamically-deployed aspects. In *Proceedings of the 7th ACM International Conference on Aspect-Oriented Software Development (AOSD 2008)*, pages 168–179, Brussels, Belgium, Apr. 2008. ACM Press. URL http://pleiad.dcc.uchile.cl/papers/2008/tanter-aosd2008.pdf.

[156] E. Tanter. Beyond static and dynamic scope. In *Proceedings of the 5th symposium on Dynamic languages*, DLS '09, pages 3–14, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-769-1. doi: 10.1145/1640134.1640137.

[157] É. Tanter. Execution levels for aspect-oriented programming. In *Proceedings of AOSD'10)*, pages 37–48, Rennes and Saint Malo, France, Mar. 2010. ACM Press. Best Paper Award.

[158] É. Tanter, J. Noyé, D. Caromel, and P. Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In *Proceedings of OOPSLA '03, ACM SIGPLAN Notices*, pages 27–46, nov 2003. doi: 10.1145/949305.949309. URL http://www.dcc.uchile.cl/~etanter/research/publi/2003/tanter-oopsla03.pdf.

[159] R. Toledo, A. Núñez, É. Tanter, and J. Noyé. Aspectizing java access control. *IEEE Trans. Software Eng.*, 38(1):101–117, 2012.

[160] E. Truyen, N. Cardozo, S. Walraven, J. Vallejos, E. Bainomugisha, S. Günther, and T. DHondt. Context-oriented programming for customizable saas applications. In *Symposium on Applied Computing*, SAC'12. ACM press, 2012.

[161] D. Ungar and R. B. Smith. Self: The power of simplicity. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 227–242, Dec. 1987. doi: 10.1145/38765.38828.

[162] T. Van Cutsem and M. S. Miller. Proxies: design principles for robust object-oriented intercession apis. In *Proceedings of the 6th symposium on Dynamic languages*, DLS '10, pages 59–72, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0405-4. doi: 10.1145/1869631.1869638. URL http://doi.acm.org/10.1145/1869631.1869638.

[163] J. Viega, B. Tutt, and R. Behrends. Automated delegation is a viable alternative to multiple inheritance in class based languages. Technical report, University of Virginia, Charlottesville, VA, USA, 1998.

[164] J. Voigt, W. Irwin, and N. Churcher. Intuitiveness of class and object encapsulation. In *6th International Conference on Information Technology and Applications*, 2009.

[165] M. von Löwis, M. Denker, and O. Nierstrasz. Context-oriented programming: Beyond layers. In *Proceedings of the 2007 International Conference on Dynamic Languages (ICDL 2007)*, pages 143–156. ACM Digital Library, 2007. ISBN 978-1-60558-084-5. doi: 10.1145/1352678.1352688. URL `http://scg.unibe.ch/archive/papers/Loew07aPyContext.pdf`.

[166] M. Wang, J. Gibbons, and N. Wu. Incremental updates for efficient bidirectional transformations. *SIGPLAN Not.*, 46:392–403, sep 2011. ISSN 0362-1340. doi: 10.1145/2034574.2034825. URL `10.1145/2034574.2034825`.

[167] A. Warth, M. Stanojević, and T. Millstein. Statically scoped object adaptation with expanders. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 37–56, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-348-4. doi: 10.1145/1167473.1167477.

[168] E. Wernli. Theseus: Whole updates of Java server applications. In *Proceedings of HotSWUp 2012 (Fourth Workshop on Hot Topics in Software Upgrades)*, pages 41–45, June 2012. doi: 10.1109/HotSWUp.2012.6226616. URL `http://scg.unibe.ch/archive/papers/Wern12b.pdf`.

[169] E. Wernli, D. Gurtner, and O. Nierstrasz. Using first-class contexts to realize dynamic software updates. In *Proceedings of International Workshop on Smalltalk Technologies (IWST 2011)*, pages 21–31, 2011. URL `http://scg.unibe.ch/archive/papers/Wern11a-ActiveContext.pdf`. http://esug.org/data/ESUG2011/IWST/Proceedings.pdf.

[170] E. Wernli, M. Lungu, and O. Nierstrasz. Incremental dynamic updates with first-class contexts. In *Objects, Components, Models and Patterns, Proceedings of TOOLS Europe 2012*, pages 304–319, 2012. doi: 10.1007/978-3-642-30561-0_21. URL `http://scg.unibe.ch/archive/papers/Wern12a.pdf`.

[171] E. Wernli, P. Maerki, and O. Nierstrasz. Ownership, filters and crossing handlers: flexible ownership in dynamic languages. In *Proceedings of the 8th symposium on Dynamic languages*, DLS '12, pages 83–94, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1564-7. doi: 10.1145/2384577.2384589. URL `http://scg.unibe.ch/archive/papers/Wern12c.pdf`.

[172] E. Wernli, M. Lungu, and O. Nierstrasz. Incremental dynamic updates with first-class contexts. *Journal of Object Technology*, 12(3):1:1–27, Aug. 2013. ISSN 1660-1769. doi: 10.5381/jot.2013.12.3.a1. URL `http://scg.unibe.ch/archive/Wern13a.pdf`.

[173] T. Wrigstad, F. Pizlo, F. Meawad, L. Zhao, and J. Vitek. Loci: Simple thread-locality for java. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 445–469, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-03012-3. doi: 10.1007/978-3-642-03013-0_21.

[174] T. Würthinger, C. Wimmer, and L. Stadler. Unrestricted and safe dynamic code evolution for Java. *Science of Computer Programming*, July 2011. ISSN 01676423. doi: 10.1016/j.scico.2011.06.005. URL `http://dx.doi.org/10.1016/j.scico.2011.06.005`.

[175] Y. Zibin, A. Potanin, M. Ali, S. Artzi, A. Kie, un, and M. D. Ernst. Object and reference immutability using java generics. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 75–84, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-811-4. doi: 10.1145/1287624.1287637. URL `http://doi.acm.org/10.1145/1287624.1287637`.

Erklärung

gemäss Art. 28 Abs. 2 RSL 05

Name/Vorname: .....

Matrikelnummer: .....

Studiengang: ……

Bachelor ☐ Master ☐ Dissertation ☐

Titel der Arbeit: .....

.....

.....

LeiterIn der Arbeit: .....

.....

Ich erkläre hiermit, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe r des Gesetztes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist.

.....

Ort/Datum

.....

Unterschrift

Erwann Wernli

Software Composition Group
Institut für Informatik
Universität Bern

Phone: + 41 78 768 72 56
Email: wernli@iam.unibe.ch
Homepage: www.ewernli.com

Personal

Born on September 23, 1980.

Swiss Citizen.

Education

MSc in Computer Science, *Swiss Federal Institute of Technology Lausanne*, 1999–2004.

Ph.D. in Computer Science, *University of Bern*, 2009–2013.

Employment

Research Assistant, *University of Bern*, 2009–2013.

Software Engineer, *IMTF*, 2006–2009.

Consultant, *Blue-Infinity*, 2004–2006.

Publications

Journal Articles

Erwann Wernli, Mircea Lungu, and Oscar Nierstrasz. Incremental Dynamic Updates with First-class Contexts, 2013, *In Journal of Object Technology* 12(3) p. 1–27

Proceedings

Erwann Wernli, Mircea Lungu, and Oscar Nierstrasz. Incremental Dynamic Updates with First-class Contexts, 2012, *In Proceedings of TOOLS Europe 2012*, p. 304–319.

Erwann Wernli, Pascal Maerki, and Oscar Nierstrasz. Ownership, filters and crossing handlers: flexible ownership in dynamic languages, 2012, *In Proceedings of the DLS 2012*, p. 83–94

Erwann Wernli. Theseus: Whole updates of Java server applications. *In Proceedings of HotSWUp 2012*, 2012, p. 41-45.

Awards

Best Paper Award at TOOLS 2012 for Incremental Dynamic Updates with First-class Contexts

Last updated: September 18, 2013