On the Revival of Dynamic Languages*

Oscar Nierstrasz, Alexandre Bergel, Marcus Denker, Stéphane Ducasse,
Markus Gälli and Roel Wuyts

Software Composition Group, University of Bern
www.iam.unibe.ch/∼scg

Abstract. The programming languages of today are stuck in a deep rut that has developed over the past 50 years. Although we are faced with new challenges posed by enormous advances in hardware and internet technology, we continue to struggle with old-fashioned languages based on rigid, static, closed-world file-based views of programming. We argue the need for a new class of dynamic languages that support a view of programming as constant evolution of living and open software models. Such languages would require features such as dynamic first-class namespaces, explicit meta-models, optional, pluggable type systems, and incremental compilation of running software systems.

1 Introduction

It is no exaggeration to say that mainstream programming languages of today are inherently *static*. That is to say, these languages tolerate change at compile time, but precious little at run-time. To state the case more strongly, most languages assume a *closed world view*: specifically, they assume that *the world is consistent, and it will not change.*

That this assumption is patently false is obvious to anyone who has experienced the development of real, large software systems. Nevertheless, it is a fact that virtually no programming language today provides specific language mechanisms to help developers cope with the fact that the systems they work on will, inevitably change [LB85].

As concrete examples, we can observe that it is hard to:

- modify a running system,
- make changes that impact the whole system,
- reason about consequences of change,
- introduce run-time reflection *on-demand*,
- keep code, documentation and tests synchronized.

Furthermore, we can observe that increasing trends towards open, distributed systems, and pervasive computing make these issues even more critical. In this paper we take the standpoint that:

* To appear, Proceedings Software Composition 2005, ed. Thomas Gschwind and Uwe Aßmann, LNCS 2005.

Inherently static languages will always pose an obstacle to the effective realization of real applications with essentially dynamic requirements.

We therefore conclude that research is urgently needed to develop a new class of *dynamic languages*, *i.e.*, languages that support change at run-time. With this in mind, we outline five complementary research tracks that explore support for change in dynamic programming languages.

In Section 2 we revisit the most basic assumption behind most programming languages today: that programs live in *files*. By challenging this assumption, we argue, we can make it easier to change a running system. In Section 3 we argue that *first-class namespaces* are a fundamental concept missing from most programming languages, yet are needed to properly manage the scope of change. In Section 4 we explore the theme of type systems for dynamic languages (an apparent *non sequitur*), and in particular argue in favour of *pluggable type systems* as a means to reason about change. In Section 5 we explore the notion of *reflection on demand* as a mechanism to support and control both run-time introspection and behavioural reflection. In Section 6 we argue that *examples* integrated into the programming language and environment offer an effective way to keep code, documentation and tests in sync. We conclude in Section 7 with some remarks on ongoing and future work.

2 Living objects

The static nature of most programming languages is immediately evident in the programming model these languages support. Programs lives in files. To change a system, we must edit these files, recompile them, and restart the system. Oh, and by the way, if the layout of any persistent data changes, we will have to have some *ad hoc* way to migrate the data. It is essentially *impossible to change a running system*. The system must be stopped and restarted for changes to take effect.

Surprisingly little effort has been invested over the years in developing languages that support run-time change to the persistent program state. Smalltalk [Gol84] and its descendants, like Self [US87], support a programming model in which all objects live in a persistent program *image*. This model of persistence, however, is rather weak, as images reside in memory, and must be explicitly saved to the file system. Although intermediate changes are logged, disasters can occur, and images may be corrupted.

Smalltalk and CLOS [Kee89,KdRB91] also support shape-changing of objects: when a class' format is changed (for example, instances are added), the memory layout of instances of this class (or its subclasses) is updated to follow suit [MLW05]. Even though this is a very rudimentary form of data migration, it is quite effective.

Considerable work was done in the mid to late 1980s on object-oriented databases and their integration with programming languages [MOP85,PS87]. This work also led to research on schema evolution [BKKK87], addressing the problem of schema changes to running systems. Considerable research has also

been carried out on so-called *database programming languages* [AB87], but these languages have only had a limited impact on programming language design in general.

Aside from various technical difficulties involved in resolving the dichotomy between databases and programming languages [TN88], Bloom and Zdonik noted as early as 1987 that there are numerous cultural differences that make it difficult for programming language and database designers to see eye to eye [BZ87].

Let us imagine what a truly dynamic and persistent object system would be like. In a such a system, one would have the illusion of directly interacting with software artifacts. Software entities and their meta-representations would be causally connected, so that changes would have an immediate effect. Furthermore, the histories of changes would be first-class entities so that change itself can be manipulated. In a distributed object system, local changes may even have a global impact.

Technically, none of these issues are especially problematic. For example languages such as Smalltalk and CLOS already offer dynamic and living object facilities with causally connected meta-representations. Several pragmatic issues must be addressed, however, in order to arrive at an effectively usable dynamic and persistent object system. How, for example, do we control the scope of change in an open, distributed and causally connected system? How do we reason about the impact of possible changes? How do we limit and control the cost of reflection? And how can we keep various software artifacts synchronized with tests and documentation? These are issues that we will touch on in the following sections.

3 First-class namespaces

Most programming languages are static in the sense that they assume the world is consistent. They do not tolerate inconsistency. As a consequence, changes must always be made in a way that restores consistency to the world.

Reality, however, dictates that in complex systems, consistency is an illusion. For this reason, workarounds are needed to maintain this illusion, such as deprecation, or ornate naming conventions to differentiate concurrent versions of software artifacts.

There is, in fact, a well-established programming language mechanism that supports inconsistent world views, but it is in most cases unfortunately realized at best as a second class citizen. *Namespaces* are well-defined boundaries providing a set of *definitions*, *i.e.* names bound to values. Every programming language supports various forms of namespaces, be they as fine-grained as the context of a procedure or a block, or as coarse-grained as packages or modules. With the notable exception of Scheme [Dyb03], virtually no mainstream programming languages exist that have all their namespaces as first-class citizens *i.e.* that can be passed and manipulated as any other value in the language.

As it turns out, first-class namespaces can be used to great effect to form the basis of a computational model for a programming language [ALSN01]. First-

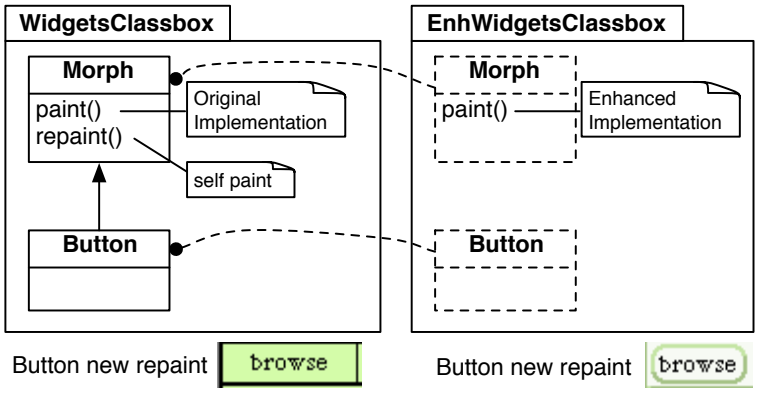

Fig. 1. Implicitly rebinding classes within classboxes.

class namespaces can be used to model objects, classes, metaobjects, software components, modules, and a variety of compositional abstractions, such as wrappers and mixin layers [AN00,NA00,NA05]. Furthermore, namespaces lend themselves well to formal specification using standard semantic modeling techniques, which can form the basis for reasoning about language constructs [AN05].

A particularly interesting application of namespaces in the context of change is to encapsulate *class extensions*. Languages like Smalltalk [GR89] or CLOS [Kee89] have traditionally supported the ability for programmers to define a set of extensions to existing classes. Extensions typically are used to add or redefine methods in situations where subclassing is not an option. (For example, by extending the class Object, one can ensure that the extension will be available to all classes, not just those that inherit from a particular subclass.)

Classboxes are namespaces that define both classes and class extensions [BDNW05]. A classbox may import a class from another classbox, and extend it locally. The *local rebinding* feature ensures that extensions remain local to the classbox introducing the extension, and other classboxes that (transitively) import the extended classes. A method addition or redefinition can be executed only within the classbox that defines this extension and to other classboxes that import the extended class. Within a given classbox, the world is always consistent, so collaborating classes are always well-defined. But multiple classboxes can support very different views of a universe full of inconsistencies.

The following example illustrates a method extension with local rebinding. Figure 1 depicts a classbox WidgetsClassbox that defines a class Morph, which is the root of the graphic element hierarchy in the Squeak system [IKM+97], and a subclass Button. Morph contains a paint() method and a repaint() method that calls paint(). The classbox EnhWidgetsClassbox imports Morph and redefines the paint() method. It also imports the subclass Button. In the context of WidgetsClassbox, invoking the repaint() method on an instance of Button invokes the definition of paint() in Morph defined by WidgetsClassbox. Within

EnhWidgetsClassbox, invoking repaint() triggers the enhanced implementation of paint() defined in EnhWidgetsClassbox. This is an illustration of the *local rebinding* facility.

Static classboxes can be used effectively to bundle a set of related class extensions that capture cross-cutting concerns, much in the way that mixin layers bundle sets of related features that can be applied in tandem [SB02]. *Dynamic* classboxes furthermore offer the possibility to dynamically apply (or disable) a set of related class extensions.

Imagine the situation in which a running system has to be upgraded *without being interrupted* and while *preserving behavior of its clients*. Dynamic classboxes offer a disciplined way out of this predicament: A patch consisting of classboxes can be dynamically applied to a running system without it being halted. Modifications, consisting of method additions and redefinitions, and encapsulated as classboxes, are locally visible to these classboxes and to new clients that rely on them. Former clients are guaranteed not to be impacted whereas new clients can rely on the new system.

Dynamic, first-class namespaces would appear to offer a number of further interesting and useful capabilities. First of all, a namespace can be used to restrict the scope of certain changes. Within a single running system, namespaces could help to indicate which clients may see a given set of changes. More interestingly, dynamic namespaces could broaden their scope at run-time, much in the way that dynamic classboxes can be applied or disabled. With dynamic namespaces, one could gradually introduce changes to a running system, extending the scope of change till it applies to all concerned clients. At any one point in time, however, there would be no need for different parts of the universe of namespaces to be mutually consistent.

4 Pluggable types

Generally speaking, static languages have *obligatory* and static type systems, that is, they attempt to use static type information to guarantee that no dynamic type errors may occur, and *refuse* any program that cannot be type-checked. The way this is achieved *always* entails a trade-off in the sense that any static type system will prevent you from writing certain "correct" programs simply because it cannot prove that no type error exists. The art of designing a usable type system is to make sure that no "interesting" program is forbidden (or that interesting programs can always be rewritten in an easy way to make them acceptable).

Static type systems, however, are the enemy of change. Reflective code, especially in statically typed object-oriented languages, can be especially cumbersome and verbose, since workarounds are needed for any operations that will not be known till run-time. Even languages that sport state-of-the-art type systems, such as (different variants of) ML or Haskell, struggle with overloading, polymorphism and reflection in the context of type-safety [Mac93]. For example, MetaOCaml, an extension of OCaml, provides a type-safe quasi-quoting mech-

anism that can be used to generate type-safe code at runtime [Tah03], but has no support for reflection. Extensions of polymorphism (such as first-class polymorphism in ML [Rus00] or in Haskell [Jon97]) exist but do not always allow for separate compilation (unless the type-preservation rules are relaxed) [KS04].

Furthermore, static type systems can produce a false sense of security. Runtime type-checks (*i.e.* "downcasts") in Java, for example, can hide a host of type-errors.

The issue of static typing is a divisive one, often splitting programmers into two camps: those who believe that dynamic languages are evil because they are "untyped" (not true — they are *dynamically* typed), and those who believe that static languages are evil because they prevent you from writing interesting programs without catching any interesting errors.

Instead of having static types hinder change, we would like to use them to support change. In particular, we want more, rather than less, expressiveness, fewer constraints, and more kinds of checks. We believe there exists a comfortable middle ground. At the center of this middle ground is a simple principle:

> *A type system should never be used to affect the operational semantics of a programming language.*

Once this principle is out of the way, we can entertain various notions of *optional type systems*, such as that of the Strongtalk language [BG93], [Str], which introduce static typechecking without compromising flexibility. We can even go one step further and explore the notion of multiple, *pluggable type systems* proposed by Gilad Bracha [Bra04].

Considerable research has been carried out in recent years on non-standard type systems such as (for example) alias types [SWM00], confined types [GPV01], [ZPV03], flow-sensitive type qualifiers [Fos02,FTA02], proxy inference [PSH04], scoped types [ZNV04], and demand-driven type inference with subgoal pruning [SS04].

It is clearly unrealistic to expect that static programming languages will or even could be developed to take advantage of all these new developments. A much more reasonable, and interesting alternative, is to envisage a dynamic programming language into which various non-standard type systems could be plugged. For example, a heuristics-based type inferencer can enable program understanding of dynamically typed programs [Wuy01]. Or a pluggable type-system dealing with worst-case execution times for methods or components can check runtime properties for programs intended to be run in hard real-time systems [HBW02,WDN05].

5 Reflection on-demand

Reflection enables the changing of systems without the need to rebuild or even restart them. This is an important basis for building the dynamic sytems of the future: Mobile, Ubiquitous, Always-On.

To change a running system means that we must *reify* behavioural aspects, *interact* with them to indicate the desired changes, and *reflect* changes to obtain their effects in the running system.

Reflection is a well-understood research topic with a long tradition of support in various programming languages and within various paradigms [FJ89,KdRB91] but with limited support in most static languages [Chi95]. Nevertheless, totally reflective systems suffer from many disadvantages:

- *Security:* If a language is reflective, the client that uses reflection can do anything.
- *Stability:* The effects of reflection are global: In a system with multiple clients, one client using reflection on a system service impacts all other clients.
- *Performance:* Full reflection is costly: To enable it, all behavioral aspects need to be reified in such a way that clients have the opportunity to change them.

These disadvantages all stem from the absence of a scope concept in the context of reflection. Scope is needed to:

- Separate the meta- from the base-layer.
- Define *where* and *when* reflection should be available.
- Limit the reflective interface to certain clients.
- Constrain the effects of reflection to certain clients.

Ideally, we would like to have *scoped reflection on demand*, that is to control *when* and *where* and *for whom* reflective services should be available. Such a reflective language would first of all be more secure, as untrusted clients could be given restricted reflective access. It would also be more stable, since changes made using reflection could be limited to the client that made them. Last but not least it could be made faster, since the reification would only be done for those clients that need it. Two recent research activities give us some hints how this may be achieved.

Mirrors in Self and Strongtalk provide structural reflection on demand [BU04]. In order to reflect on a particular object, a mirror object will be created at runtime. The mirror reifies the reflective services for the object under study. Thus mirrors provide a dedicated interface for the reflective services: meta- and base-layer are separated, the particular interface handed to a client can be defined by the object and it can differ between multiple clients. So mirrors provide some of the properties we need. But they have shortcomings, as the support they offer for full behavioral reflection is limited. They do not support fine-grain reflection below the method level, nor mirror-based intercession. And mirrors do not provide a way to scope the effects of reflective change.

Reflex provides fine-grained behavioural reflection for Java entities [TBN01] [TNCC03]. The entities to be reified can be selected by time (enabled/disabled by the program), or space. For spatial selection we can specify the entity (e.g., a class or an object), the operation (*e.g.* message send or a field access) or

combine these to select a specific operation (*e.g.* a certain message sent to a certain object).

Reflex uses bytecode transformations for reifying Java execution entities like instance variable access, method calls, exceptions and typecasts. Java as a static system does not allow bytecode to be modified at runtime, it needs to be done statically at load-time: Only those entities selected at load time can be reflective at runtime.

Geppetto is an implementation of Reflex for Squeak that provides the same fine-grained behavioural reflection for Squeak language entities. It supports reification of variables (instance variables and temps), message sends and message receive. Like Reflex, Geppetto uses bytecode transformation, but as Squeak is a dynamic system, these modifications are done at runtime. With Geppetto we want to explore the ideas of Reflex in a dynamic language, especially how to combine Geppetto with the idea of dynamic classboxes as outlined in Section 3.

Classboxes can be used to package reflective aspects of objects. When reflection is needed, the corresponding classbox can be dynamically loaded. Only clients that need reflection will see those services. Classboxes are used to extend the system without making the effect of this extension global. In the same way, classboxes could scope the effect of reflection.

6 Example Objects

Object-oriented code can be hard to understand, extend and adapt. One source of this difficulty is the disconnect between run-time architecture and source code: whereas at run-time we have a collection of interacting objects, the source code merely presents us with a class hierarchy. As a consequence it can be hard to identify the run-time structures in the code [Nie04]. Furthermore, architectural constraints and contracts tend to be implicit in the code, so it may be hard to tell whether given changes are consistent with the existing contracts in place.

Examples are a well-established medium for communicating how things works in virtually all domains. Dictionaries like the Oxford Dictionary of Current English [Soa01] provide the reader with lots of concrete examples of current usage of a given word. Curiously the use of examples is not widespread in running software systems, though they would offer many benefits. In particular, examples are run-time entities that can be manipulated, examples document usage scenarios, examples can form the basis of executable tests, and finally examples (that fulfil their tests) are guaranteed to be in sync with the running system.

A particularly useful notion is that of *one-method commands*. These are argument-free methods that serve as examples for methods and objects, focus on some given method under test, and return an example object. One-method commands may be composed to form suites of tests.

As an example, consider the following Smalltalk method (defined on the class side of the `Account` class:

```
Account class >> deposit100On123
```

```
|anAccount|
anAccount:= Account accountNumber123.
self test: [ anAccount deposit: 100 ].
self assert: [ anAccount balance==100 ].
^anAccount
```

This code is evaluated by sending the message `Account deposit100On123`. The method makes use of another one-method command of the `Account` class, called `accountNumber123`, which presumably returns an example `Account.deposit-100On123` then focuses on the `deposit:` method, thus providing an example usage of this method. (`self test: aBlock` performs `aBlock`, while documenting what is actually under test). A test is performed, (`self assert: ...`) and the modified example object is returned.

Here we can see how tests are composed from one-method commands, and explicitly link tests with methods under test, tests with classes under test, and tests with other tests.

Taivalsaari [Tai97] gives an overview about the philosophical differences between prototypical and class-based languages. We believe that a class-based language with a built-in facility for composing and fetching examples can help to bridge the gap between these two paradigms.

Deursen *et al.* [Deu01] discuss several benefits and drawbacks of unit tests for program comprehension. They do not discuss how one can navigate between tests and programs but it is clear that they should be together as close as possible.

In [DMBK01] Deursen *et al.* discuss several bad smells of test code. They describe the bad smell of *eager tests* which which test several methods of an object at the same time, and are hard to comprehend. They therefore suggest to apply the extract method refactoring to separate tests into what we call one-method commands which exemplify exactly only one method. Other bad smells include "general fixture" and "test code duplication", which we suggest to clean using de- and recomposition of the test code into one-method commands.

Edwards [Edw04] coined the term "example centric programming". Based on user provided examples (which we again call one-method commands) the developer can browse abstract methods side by side with concrete calls of these methods triggered by the one-method commands. Edwards does not provide means of composing and linking these one-method commands as we do.

7 Conclusions

In many ways, we are still in the dark ages of programming language design. Consider, for example, the great innovations in programming languages over the past fifty years. To a large extent, most of these innovations were achieved in the 1950s and 1960s. It is harder and harder to identify significant contributions over the past 20 years. It is also hard to identify truly radical language designs in recent years.

One may interpret this as a sign that the state-of-the-art in programming language design is stabilizing, or even that research in programming languages

is essentially dead. Another interpretation, however, is that language design is in a rut due to our fixation with a certain style of language design. We have argued in this paper that static languages have hampered innovation, and furthermore that the death of file-based languages is the first step towards a new generation of dynamic languages.

We need to come to terms with persistency, inconsistency and change in programming languages. This means that dynamic programming languages should support the notion of software as living, changing systems, they should provide support multiple and possibly inconsistent viewpoints of these systems. Static type systems still have their place, but they should serve rather than hinder expressiveness. To support dynamic change, behavioural reflection is needed, but it should be provided only on-demand, when and where it is needed. Finally, examples integrated into the language run-time can help one to document and test the software in a synchronized fashion.

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "A Unified Approach to Composition and Extensibility" (SNF Project No. 200020-105091/1, Oct. 2004 - Sept. 2006).

References

AB87. Malcolm P. Atkinson and O. Peter Buneman. Types and persistence in database programming languages. *ACM Comput. Surv.*, 19(2):105–170, 1987.

ALSN01. Franz Achermann, Markus Lumpe, Jean-Guy Schneider, and Oscar Nierstrasz. Piccola — a small composition language. In Howard Bowman and John Derrick, editors, *Formal Methods for Distributed Processing — A Survey of Object-Oriented Approaches*, pages 403–426. Cambridge University Press, 2001.

AN00. Franz Achermann and Oscar Nierstrasz. Explicit Namespaces. In Jürg Gutknecht and Wolfgang Weck, editors, *Modular Programming Languages*, volume 1897 of *LNCS*, pages 77–89, Zürich, Switzerland, September 2000. Springer-Verlag.

AN05. Franz Achermann and Oscar Nierstrasz. A calculus for reasoning about software components. *Theoretical Computer Science*, 331(2-3):367–396, 2005.

BDNW05. Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Classboxes: Controlling visibility of class extensions. *Computer Languages, Systems and Structures*, September 2005. To appear.

BG93. Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, volume 28, pages 215–230, October 1993.

BKKK87. Jay Banerjee, Won Kim, H-J. Kim, and H.F. Korth. Semantics and implementation of schema evolution in object-oriented databases. In *Proceedings ACM SIGMOD '87*, volume 16, pages 311–322, December 1987.

Bra04. Gilad Bracha. Pluggable type systems, October 2004. OOPSLA Workshop on Revival of Dynamic Languages.

BU04. Gilad Bracha and David Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of OOPSLA '04, ACM SIGPLAN Notices*, pages 331–344, New York, NY, USA, 2004. ACM Press.

BZ87. Toby Bloom and Stanley B. Zdonik. Issues in the design of object-oriented database programming languages. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 441–451, December 1987.

Chi95. Shigru Chiba. A metaobject protocol for C++. In *Proceedings of OOPSLA '95*, volume 30 of *ACM SIGPLAN Notices*, pages 285–299, October 1995.

Deu01. Arie van Deursen. Program comprehension risks and opportunities in extreme programming. In *Working Conference on Reverse Engineering*, pages 176–, 2001.

DMBK01. Arie van Deursen, Leon Moonen, Alex van den Bergh, and Gerard Kok. Refactoring test code. In M. Marchesi, editor, *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes (XP2001)*, pages 92–95. University of Cagliari, 2001.

Dyb03. Kent Dybvig. *The Scheme Programming Language.* MIT Press, 2003.

Edw04. Jonathan Edwards. Example centric programming. In *OOPSLA 04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 124–124. ACM Press, 2004.

FJ89. Brian Foote and Ralph E. Johnson. Reflective facilities in Smalltalk-80. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, volume 24, pages 327–336, October 1989.

Fos02. Jeffrey Scott Foster. *Type Qualifiers: Lightweight Specifications to Improve Software Quality.* Ph.D. thesis, University of California, Berkeley, December 2002.

FTA02. Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proceedings of PLDI '02 Conference on Programming Language Design and Implementation*, pages 1–12. ACM Press, 2002.

Gol84. Adele Goldberg. *Smalltalk 80: the Interactive Programming Environment.* Addison Wesley, Reading, Mass., 1984.

GPV01. Christian Grothoff, Jens Palsberg, and Jan Vitek. Encapsulating objects with confined types. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 241–255, New York, NY, USA, 2001. ACM Press.

GR89. Adele Goldberg and Dave Robson. *Smalltalk-80: The Language.* Addison Wesley, 1989.

HBW02. E. Yu-Shing Hu, G. Bernat, and A. Wellings. A Static Timing Analysis Environment Using Java Architecture for Safety Critical Real-Time Systems. *In Proceedings of 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS-2002)*, pages 64–71, January 2002.

IKM+97. Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: The story of Squeak, A practical Smalltalk written in itself. In *Proceedings OOPSLA '97, ACM SIGPLAN Notices*, pages 318–326. ACM Press, November 1997.

Jon97. Mark P. Jones. First-class polymorphism with type inference. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 483–496. ACM Press, 1997.

KdRB91. Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

Kee89. Sonia E. Keene. *Object-Oriented Programming in Common-Lisp*. Addison Wesley, 1989.

KS04. Andrew Kennedy and Don Syme. Transposing f to c#: Expressivity of polymorphism in an object-oriented language. *Concurrency and Computation: Practice and Experience*, 16(7), 2004.

LB85. Manny M. Lehman and Les Belady. *Program Evolution – Processes of Software Change*. London Academic Press, 1985.

Mac93. David B. MacQueen. Reflections on standard ml. In *Functional Programming, Concurrency, Simulation and Automated Reasoning*, pages 32–46. Lecture Notes in Computer Science, 1993.

MLW05. Eliot Miranda, David Leibs, and Roel Wuyts. Parcels: a fast and feature-rich binary deployment technology. *Computer Languages, Systems and Structures*, September 2005. To appear.

MOP85. David Maier, Allen Otis, and Alan Purdy. Object-oriented database development at servio logic. *IEEE Database Engineering*, 8(4):58–65, December 1985.

NA00. Oscar Nierstrasz and Franz Achermann. Supporting Compositional Styles for Software Evolution. In *Proceedings International Symposium on Principles of Software Evolution (ISPSE 2000)*, pages 11–19, Kanazawa, Japan, November 2000. IEEE.

NA05. Oscar Nierstrasz and Franz Achermann. Separating concerns with first-class namespaces. In Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and sit Mehmet Ak editors, *Aspect-Oriented Software Development*, pages 243–259. Addison-Wesley, 2005.

Nie04. Oscar Nierstrasz. Software evolution as the key to productivity. In A. Knapp M. Wirsing and S. Balsamo, editors, *Radical Innovations of Software and Systems Engineering in the Future*, volume 2941 of *LNCS*, pages 274–282. Springer-Verlag, 2004.

PS87. D. Jason Penney and Jacob Stein. Class modification in the gemstone object-oriented DBMS. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 111–117, December 1987.

PSH04. Polyvios Pratikakis, Jaime Spacco, and Michael Hicks. Transparent proxies for java futures. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 206–223, New York, NY, USA, 2004. ACM Press.

Rus00. Claudio V. Russo. First-class structures for standard ml. *Nordic J. of Computing*, 7(4):348–374, 2000.

SB02. Yannis Smaragdakis and Don Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM TOSEM*, 11(2):215–255, April 2002.

Soa01. Catherine Soanes, editor. *Oxford Dictionary of Current English*. Oxford University Press, July 2001.

SS04. S. Alexander Spoon and Olin Shivers. Demand-driven type inference with subgoal pruning: Trading precision for scalability. In *Proceedings of ECOOP'04*, pages 51–74, 2004.

Str. The strongtalk type system for smalltalk. http://bracha.org/nwst.html.

SWM00. Frederick Smith, David Walker, and J. Gregory Morrisett. Alias types. In *ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems*, pages 366–381, London, UK, 2000. Springer-Verlag.

Tah03. Walid Taha. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation*, pages 30–50, 2003.

Tai97. Antero Taivalsaari. Classes versus prototypes: Some philosophical and historical observations. *Journal of Object-Oriented Programming (JOOP)*, 10(7):44–50, 1997.

TBN01. Éric Tanter, Noury Bouraqadi, and Jacques Noyé. Reflex — towards an open reflective extension of java. In *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, volume 2192 of *LNCS*, pages 25–43. Springer-Verlag, 2001.

TN88. Dennis Tsichritzis and Oscar Nierstrasz. Fitting round objects into square databases. In S. Gjessing and K. Nygaard, editors, *Proceedings ECOOP '88*, volume 322 of *LNCS*, pages 283–299, Oslo, April 1988. Springer-Verlag.

TNCC03. Éric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In *Proceedings of OOPSLA '03, ACM SIGPLAN Notices*, pages 27–46, nov 2003.

US87. David Ungar and Randall B. Smith. Self: The power of simplicity. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 227–242, December 1987.

WDN05. Roel Wuyts, Stéphane Ducasse, and Oscar Nierstrasz. A data-centric approach to composing embedded, real-time software components. *Journal of Systems and Software — Special Issue on Automated Component-Based Software Engineering*, 74(1):25–34, 2005.

Wuy01. Roel Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, 2001.

ZNV04. Tian Zhao, James Noble, and Jan Vitek. Scoped types for real-time java. In *RTSS '04: Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS'04)*, pages 241–251, Washington, DC, USA, 2004. IEEE Computer Society.

ZPV03. Tian Zhao, Jens Palsberg, and Jan Vite. Lightweight confinement for featherweight java. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 135–148, New York, NY, USA, 2003. ACM Press.