UNIVERSITY OF CALIFORNIA,
IRVINE

Component-Oriented Programming Languages: Why, What, and How

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Peter Hans Fröhlich

Dissertation Committee:
Professor Michael Franz, Chair
Professor André van der Hoek
Professor Isaac Scherson

2003

The dissertation of Peter Hans Fröhlich is approved
and is acceptable in quality and form for
publication on microfilm:

_____

_____

_____
Committee Chair

University of California, Irvine
2003

Dedication

In memory of Kimberly Haas
(October 29, 1968 – April 26, 2002)

Dich bedecken
nicht mit Küssen
nur einfach
mit Deiner Decke
(die Dir
von der Schulter
geglitten ist)
daß Du
im Schlaf nicht frierst

Später
wenn Du
erwacht bist
das Fenster zumachen
und Dich umarmen
und Dich bedecken
mit Küssen
und Dich
entdecken

Nachtgedicht, Erich Fried

Contents

List of Figures

List of Tables

Acknowledgments

A dissertation reflects the work of many people, most of whom deserve more credit than they customarily receive. However, keeping track of all the hallway conversations and newsgroup postings that end up being important after five years is just too tedious. My only recourse is to thank everyone who ever had some sort of conversation with me, not necessarily even about the topic of this dissertation, and to apologize to all not listed below.

I am deeply grateful to my advisor, Prof. Dr. MICHAEL FRANZ, for inviting me to UC Irvine as a researcher and a student, for constantly sharing his insight and wisdom, and for suffering patiently under my bizarre working habits. Michael kept me going when I was about to give up and then nudged me out of the door I was so afraid to step through. I cannot thank him enough for all he has given me.

I am also indebted to Prof. Dr. ANDRÉ VAN DER HOEK and Prof. Dr. ISAAC SCHERSON for serving on my dissertation committee and for providing heaps of helpful feedback on my work. Prof. Dr. LUBOMIR BIC and Prof. Dr. ABEL KLEIN deserve "extra credit" for sitting through my candidacy exam. Prof. Dr. DAVID ROSENBLUM introduced me to many advanced issues in software engineering and made me write a term paper that already contained the kernel of this dissertation. Prof. Dr. RICHARD LATHROP provided useful advice on giving presentations.

Outside the Irvine campus, Prof. Dr. CLEMENS SZYPERSKI (Queensland University of Technology and Microsoft Research) turned out to be an almost infinite source of inspiration and aspiration. All of our sizable email exchanges ended up somewhere in this dissertation, and I am particularly grateful for the long talk we had at OOPSLA 2001 in Florida. In addition, Clemens was kind enough to serve as

an (unofficial) outside member on my dissertation committee. I also want to thank Prof. Dr. KLAUS KÖHLER (University of Applied Sciences, Munich, Germany) who sparked my interest in programming languages, advised my diploma thesis, and encouraged me to persue still higher goals on this side of the Atlantic. Last but not least, I thank Prof. Dr. THOMAS PAYNE and Prof. Dr. BRETT FLEISCH (University of California, Riverside) for supporting me during the final months of writing.

I benefitted from countless discussions with fellow graduate students NIALL DALTON, JOACHIM FEISE, ANDREAS GAL, VIVEK HALDAR, Dr. THOMAS KISTLER, ZIEMOWIT LASKI, CHRIS LÜER, and CHRISTIAN STORK. While visiting our research group, Dr. WOLFRAM AMME and Dr. FERMÍN REIG were kind enough to share their insights as well. Short email exchanges with Prof. Dr. MARTÍN ABADI, Dr. MARTIN BÜCHI, Prof. Dr. KIM BRUCE, Prof. Dr. ERIK ERNST, Prof. Dr. PETER GROGONO, RICCARDO PUCCELLA, Prof. Dr. MARKKU SAKKINEN, and Dr. WOLFGANG WECK were also very helpful.

Finally, I want to thank my friends (especially NAOMI CARPENTER, MATTHEW DAVIES, REBECCA HARRIS, MICHAEL SHAFAE, LEILA THAROK, and CHRISTIAN VOGEL), Kim's family (DIANE and ROBERT BUCK, and LYNNE EDDINGTON), and of course my parents (ANTONIE and HANS FRÖHLICH), for keeping me at least somewhat sane during the insanity of graduate school.

Funding. The National Science Foundation (NSF) partially funded the work described in this dissertation under grants EIA-9975053 and CCR-0105710.

Publications. Parts of this dissertation were published previously [FFK99, FF00, Frö00, FF01, Frö02] and benefitted from the comments of numerous (often anonymous) reviewers.

Curriculum Vitae

Peter Hans Fröhlich

Born September 20, 1971 in Munich, Germany.
Citizen of Germany.

Education

Diplom-Informatiker (1997)

Department of Computer Science and Mathematics
University of Applied Sciences, Munich, Germany

Master of Science (2000)

Department of Information and Computer Science
University of California, Irvine

Doctor of Philosophy (2003)

School of Information and Computer Science
University of California, Irvine

Academic Experience

Winter 1998 – Spring 1998

Visiting Researcher
Department of Information and Computer Science
University of California, Irvine

Fall 1998 – Winter 1999

Teaching Assistant
Department of Information and Computer Science
University of California, Irvine

Academic Experience (continued)

Spring 1999 – Spring 2001

Research Assistant
Department of Information and Computer Science
University of California, Irvine

Summer 2001

Lecturer
Department of Information and Computer Science
University of California, Irvine

Fall 2001 – Summer 2002

Research Assistant
Department of Information and Computer Science
University of California, Irvine

Fall 2002 – current

Lecturer and Research Programmer
Department of Computer Science and Engineering
University of California, Riverside

Industrial Experience

September 1991 – February 1992

Technical Intern
Daimler-Benz Aerospace AG, Munich, Germany

September 1993 – February 1994

Technical Intern
Sisymed Software GmbH, Munich, Germany

March 1997 – December 1997

Software Engineer
NEXUS informatics GmbH, Munich, Germany

July 1998 – September 1998

Software Engineer
COCOSS GbR, Munich, Germany

Abstract of the Dissertation

Component-Oriented Programming Languages: Why, What, and How

by

Peter Hans Fröhlich

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 2003

Professor Michael Franz, Chair

In this dissertation, I investigate the notion of *component-oriented* programming languages. Simply put, a programming language is component-oriented if (and only if) it *facilitates* software development following the *paradigm* of component-oriented programming. Although this definition might not seem well-founded at first, it is *exactly* in this sense that we commonly speak of structured, modular, or object-oriented programming languages as facilitating software development following *their* respective paradigms.

The central question I address here is *how* component-oriented programming languages *differ* from programming languages for those earlier paradigms. This obviously requires an explanation of component-oriented programming as a software development paradigm as well. Given the "object hype" of the early 1990s, it should come as no surprise that even a paradigm that seems "revolutionary" at

first actually leads to mostly "evolutionary" improvements over earlier programming languages. While these improvements *derive* from the ideas of component-oriented programming, their applicability is *not* restricted to that setting. In this respect, they are similar to "classic" advances in programming languages such as the proscription against `goto`. the `case` instruction, or the introduction of explicit `module` constructs.

The main result I present in this dissertation is a *framework* of design decisions for component-oriented programming languages. This framework can be applied either to revisions of existing languages or to the design of new ones. I focus on the development of this framework, particularly on the development of the two novel language mechanisms it is based on: *stand-alone messages* and *generic message forwarding*. Using the example of Lag*oo*na, I illustrate how the framework can be applied to the design and implementation of an actual programming language. Finally, I evaluate the framework (and thus Lag*oo*na) in terms of new solutions to—sometimes long-standing—design and implementation problems drawn from both object-oriented and component-oriented programming.

Chapter 1

Introduction

> *At the present time I think we are on the verge of discovering at last what programming languages should really be like. I look forward to seeing many responsible experiments with language design during the next few years; and my dream is that by 1984 we will see a consensus developing for a really good programming language (or, more likely, a coherent family of languages).*
>
> — DONALD E. KNUTH [Knu74]

Programming languages are the bridge connecting software and hardware, the conceptual and the tangible poles of computer science. In a first approximation, software engineering drives programming language *design* while computer architecture drives programming language *implementation* [GJ97]. Thus, as long as these disciplines continue to evolve, programming languages will continue to evolve as well.

In this dissertation, I study the impact of *component-oriented programming*, an emerging software development paradigm [SGM02], on the design of programming languages. I contribute a novel framework for the macroscopic structure of component-oriented programming languages, the result of another "responsible experiment with language design" which hopefully brings us closer to KNUTH's dream. In this chapter, I give an overview of the "experiment" and the dissertation.

1.1 Problem

The classic idea of "mass-produced software components" has seen a resurgence of interest in recent years, although not in its "classic" form. In the wake of McIlroy's landmark paper [McI69], software components were primarily understood as units of *reuse*. Produced and sold by component vendors, bought and integrated by application vendors, components would end up on a user's machine as invisible parts of a "binary blob" called "application."

In their "modern" form, software components are understood as units of *extension* instead and the complementary notion of a *component framework* has appeared [SGM02]. Components extend the functionality of frameworks, while frameworks provide execution environments for components. Furthermore, components and frameworks can be produced as well as integrated by *any* interested party at *any* time.[1] No longer invisible, "modern" software components retain an autonomous character as "binary blobs" in their own right, even after they are deployed on a user's machine.

Figure 1.1 on the following page illustrates this approach to component software. A web browser, instead of being a monolithic application, is a framework responsible for managing network access and screen estate on behalf of components that provide user functionality. Composition is *hierarchical* since components of one framework can themselves be frameworks for further components. Composition is restricted by *interfaces* to ensure a functioning software system. Composition is *dynamic* since new components can be added to the system at runtime.

Software engineering has long recognized components as one of the few "silver bullets" that could alleviate the "software crisis" [Bro87]. Established software development paradigms have in fact consistently identified components with *their* primary abstraction mechanism (see Figure 1.2 on the next page):

- In structured programming, components are individual *operations* (i.e. *procedures* or *functions*); this includes McIlroy's original paper [McI69].

[1] Note that the "modern" view subsumes the "classic" view: Reuse is still practiced, but not exclusively by application vendors anymore. See Chapter 2 for the details.

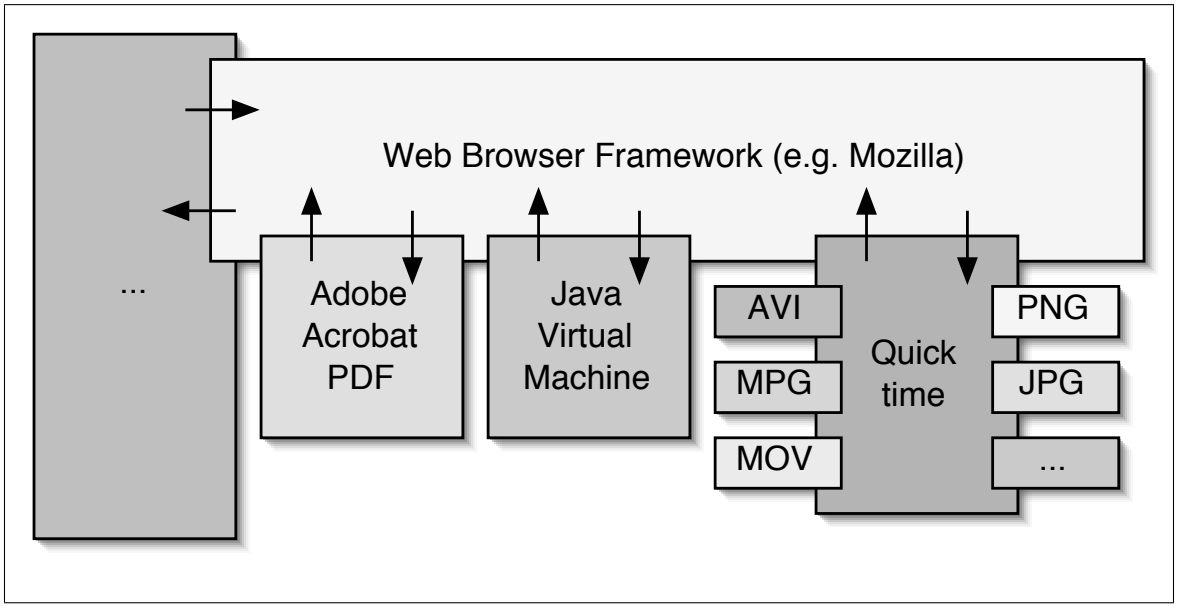


Figure 1.1: Schematic view of a web browser in terms of components, frameworks, and interfaces (shown as arrows, omitted within Quicktime for clarity).

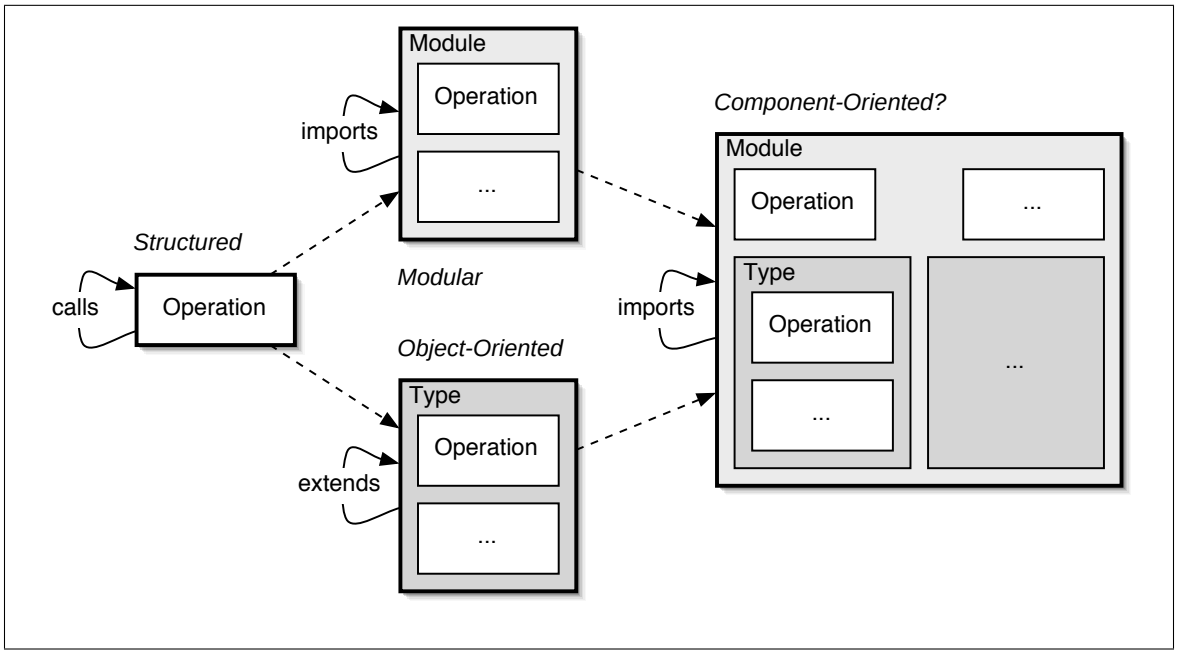


Figure 1.2: Evolution of programming language abstractions for components through various software development paradigms. (Dashed arrows indicate evolution, repeated arrows for recursive relationships omitted for clarity.)

- In modular programming, components are *modules* that encapsulate a collection of related operations.
- In object-oriented programming, components are *types* (i.e. *classes*), that again encapsulate a collection of related operations.

Of course, none of the established paradigms has in fact achieved McIlroy's original vision in this way.

For the emerging paradigm of *component-oriented programming*, a combination of all these abstraction mechanisms has been suggested instead [SGM02]: *Components are modules that encapsulate a collection of operations as well as a collection of types.* Component-oriented programming can thus be characterized as a combination of modular and object-oriented programming, and I will refer to this as the "standard model" of a component-oriented programming language (see Chapter 2). There are, however, several problems with this approach, for example:

- Interfaces must frequently be *combined* to enable composition with multiple frameworks. In the standard model, this can lead to *interface conflicts* which prevent otherwise legal compositions (see Chapter 3).
- Components must frequently be *adapted* to enable composition with frameworks they were not explicitly designed for. In the standard model, this can lead to the *fragile base class problem* which prevents successful adaptation (see Chapter 4).
- Conformance of components to interfaces must be (at least partially) *structural*. This is not commonly supported in either the modular or the object-oriented languages the standard model is based on (see Chapter 5).

More generally, while there is a consensus that *some* of the mechanisms from modular and object-oriented programming are necessary for component-oriented programming, the *exact* nature of their combination is rarely spelled out.

1.2 Approach

In order to solve the problems outlined above, I propose a novel design framework for the structure of component-oriented programming languages. I develop this framework from specific example problems that lead to two novel language mechanisms: stand-alone messages and generic message forwarding. The properties of these mechanisms allow their integration into a coherent framework which solves the problems outlined above:

- Any combination of two or more interface types is itself a *valid* interface type preserving *all* constituent messages.
- Implementation types can be adapted *conveniently* and *without* risk of the fragile base class problem.
- Conformance between interface and implementation types is *structural* yet *safe* down to the level of constituent messages.

The resulting design framework also supports *minimal typing* of parameters at component boundaries as well as *retroactive supertyping*, two concepts that support software evolution and refactoring. Using the framework, the problems of *component reentrance* can be addressed as well, and there are further applications in the areas of *iteration abstractions*, *component framework extensibility*, and design guidelines for *behavioral subtyping*.

1.3 Evaluation

Evaluating programming languages and language mechanisms objectively is notoriously difficult. While there is *general* agreement on the desirable qualities, no two books or articles seem to agree on the details. I therefore apply two complementary approaches to evaluate the language mechanisms developed in this dissertation, one comparative and one qualitative.

Comparative evaluations study related mechanisms in other programming languages as well as related design patterns and idioms. For the most part, I rely on established and validated programming languages instead of academic prototypes. These evaluations thus provide detailed analyses of the advantages and disadvantages relative to known standards.

Qualitative evaluations are based in part on the results of the comparative evaluations. I discuss the mechanisms I develop in terms of four core qualities, namely efficiency, flexibility, safety, and simplicity. To avoid misunderstandings, following are the definitions used herein:

Efficiency: An efficient programming language tries to associate fixed and preferably constant runtime costs with each mechanism it offers. Similarly, it tries to avoid mechanisms for which no such guarantee can be made.[2]

Flexibility: A flexible programming language allows programmers freedom in combining and exploiting language mechanisms and provides mechanisms that are expressive enough to lead to straightforward solutions.

Safety: A safe programming language tries to detect as many programming errors as possible at compile time. Furthermore, it tries to avoid language mechanisms for which safety can not be enforced in this way. When a mechanism can neither be analyzed statically nor removed from the language, a safe language will at least guarantee detecting the error at runtime.

Simplicity: A simple programming language tries to minimize the number of language mechanisms necessary to write useful software. Simple languages are easy to learn, mostly because they have fewer special cases that must be remembered. Simple languages also often have particularly reliable compilers.

As illustrated in Figure 1.3 on the following page, the interplay between these four qualities drives much of the research in programming language design. These

[2] In the context of programming languages, efficiency is frequently not just an asymptotic concern: Every instruction the processor has to execute on behalf of the language itself—and not the client program—is considered one instruction too many.

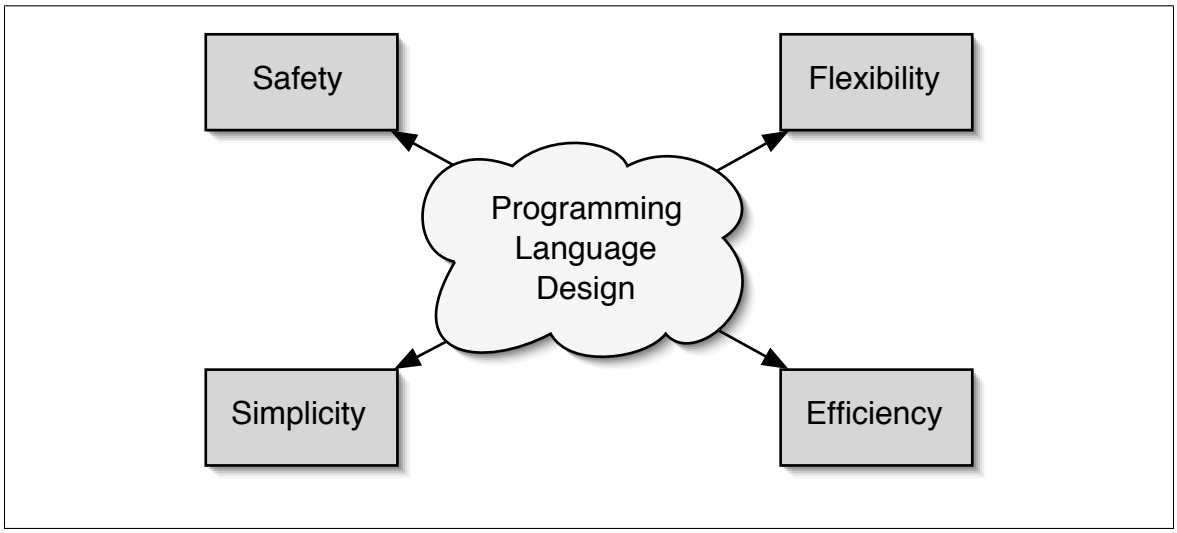


Figure 1.3: The context for programming language design in terms of four core language qualities.

qualities are not completely orthogonal. Safety and simplicity, for example, often influence efficiency in a positive way.

1.4 Benefits

Besides solving a number of technical problems, the benefits derived from my design framework fall into three major areas. First, the framework covers a previously unexplored region in the design space of programming languages and sheds new light on the exact combination of modular and object-oriented features required for component-oriented programming.

Second, it extends previous results on programming language design, namely the separation of interface types from implementation types [Sny86] and the separation of modules from types [Szy92]. Both of these results are by now widely accepted, and my contribution is to show that messages and methods should be separated as well, binding messages to modules instead of types.

Third, the framework clarifies a number of the tradeoffs involved in the design of component-oriented—and often object-oriented—programming languages:

- The tradeoff in expressive power between *forwarding* and recursive mechanisms for code reuse such as *inheritance* and *delegation*.
- The tradeoff between the level of *extensibility* required for component software and the level of *type safety* that can be guaranteed for it.
- The tradeoff between the *efficiency* of purely *static* inheritance in class-based languages and purely *dynamic* delegation in prototype-based languages.

By providing these insights and clarifications, the design framework developed in this dissertation should enable future research on component-oriented programming languages to proceed with better focus and thus more productively.

1.5 Roadmap

Research publications tend to be "rational reconstructions" of the actual research performed, and this dissertation is no exception. Instead of presenting my work in chronological order, I "fake a rational design process" [PC86] and discuss questions and findings in topical groupings. Chapters 2 – 5 constitute the core of my dissertation and focus on language design for component-oriented programming. Chapters 6 and 7 describe implementation issues and future work addressing the shortcomings that remain. Note that I discuss related work throughout the thesis where it is most appropriate instead of collecting it in a separate chapter.

In Chapter 2, I introduce component-oriented programming as a software development paradigm. I discuss the "classic" understanding of components as units of centralized reuse as well as the "modern" understanding of components as units of distributed extension. Following these preliminaries, I describe the standard model for the design of component-oriented programming languages, on which I improve in the remainder of the dissertation.

In Chapter 3, I develop *stand-alone messages*, the first novel language mechanism in my design framework. I introduce the problem of *interface conflicts* and show that programming languages following the standard model can not resolve them.

In the subsequent analysis, I trace this shortcoming to the status of messages in object-oriented programming languages and argue that they must be independent of types. Finally, I evaluate stand-alone messages by comparing them to existing approaches solving similar problems.

In Chapter 4, I develop *generic message forwarding*, the second novel language mechanism in my design framework. I introduce the problem of *fragile base classes* and show that programming languages following the standard model are prone to this problem as well. In the subsequent analysis, I trace this shortcoming to established results on the recursive binding of self references in object-oriented programming languages. I argue that recursive mechanisms such as inheritance and delegation must be abandoned in favor of forwarding and exhibit a flexible mechanism for achieving this. Finally, I evaluate generic message forwarding by comparing it to existing approaches solving similar problems.

In Chapter 5, I introduce the programming language Lag*oo*na, which is based on the language mechanisms developed earlier. I outline my design framework for component-oriented programming languages and relate it to Lag*oo*na's base language Oberon. I then review the individual design decisions made in applying the framework and describe Lag*oo*na in detail. Finally, I evaluate the design framework—and thus Lag*oo*na—by exhibiting novel solutions to several design and implementation problems drawn from both object-oriented and component-oriented programming.

In Chapter 6, I discuss implementation aspects of Lag*oo*na and component-oriented programming languages in general. I focus on efficient techniques for the problem of *message dispatch*, an area where languages following my design framework require more general solutions than those commonly adopted for established object-oriented programming languages.

In Chapter 7, I outline several directions for future work, addressing shortcomings that remain in Lag*oo*na as well as promising extensions. Finally, in Chapter 8, I summarize the contributions made in this dissertation and offer my conclusions.

Chapter 2

Background

> *I would like to see components become a dignified branch of software engineering. . . . I think there are considerable areas of software ready, if not overdue, for this approach.*
>
> — M. DOUGLAS MCILROY [McI69]

> *A component is a unit of composition with contractually specified interfaces and explicit context dependencies only. . . . A component can be deployed independently and is subject to composition by third parties.*
>
> — CLEMENS SZYPERSKI [SGM02]

In this chapter, I provide the necessary background information on which the remainder of the dissertation is built. I first review the idea of "software components" in the various forms it has taken over the years and show how it relates to the design of paradigmatic programming languages (Section 2.1). I then introduce the basic design elements of component-oriented programming languages, the issue I focus on for the remainder of the dissertation (Section 2.2). I conclude the chapter with a brief discussion of related work in the area of component software, mainly to properly explain the scope of my work (Section 2.3).

2.1 Component-Oriented Programming

As evidenced by McIlroy's quote from 1968, the idea of software components has been around for a long time. One problem with ideas as old as this one is that everybody has their own understanding of what a "component" is (or ought to be). A similar situation existed for "objects" and "object-oriented programming" until the late 1980s, when one particular approach to "object-oriented programming"—in the style of Smalltalk [GR83], Eiffel [Mey92], and C++ [Str00]—finally became the widely accepted understanding [Weg87].

To gain a clearer understanding of component software, I review the "classic" as well as the "modern" understanding of the term in detail and then relate both the design of paradigmatic programming lanuages.

2.1.1 Classic Perspective: Centralized Reuse

The idea of *assembling* software systems out of existing components instead of *building* software systems from scratch was first described by McIlroy in 1968 [McI69]. He envisioned nothing less than an "industrial revolution" of software production. Drawing analogies to production processes in established industries, he called for "catalogs" that would list the software components available from certain vendors, including descriptions of their system requirements and quality characteristics. Component vendors would specialize in certain areas of expertise, while application vendors—in the business of producing software systems for users—would select required components from such catalogs and buy them, instead of developing the equivalent functionality themselves.

The resulting "market of software components" would then in due time run its course, leaving only vendors of "high-quality" components that are available at "reasonable" prices behind. The end result would be beneficial for all parties:

- Component vendors could concentrate on their areas of expertise without having to actually produce applications to survive.

- Application vendors could concentrate on the needs of their users without having to become experts in all the areas their application touches upon.
- Users could expect higher-quality applications at lower prices since the cost savings and productivity gains would "trickle down" to them.

Even in retrospect, armed with the knowledge that MCILROY's vision still has not been realized on any noticable scale, there is an inherent attraction to this idea in which "free markets" feature so prominently for the benefit of everyone involved.[1]

Software engineering [Som02], which was essentially "born" as a discipline at the same conference where MCILROY presented his vision [NR69], has indeed come back to the idea of software components time and again. This is understandable since—as BROOKS put it twenty years later [Bro87]—a flourishing market of software components is one of the few "silver bullets" that have the potential to actually alleviate the software crisis. The reason is simply that software components are, by design, meant to be *reused*. They thus reduce the amount of software development necessary, and development efforts that can be avoided are development efforts that cannot go wrong. I will refer to this emphasis on "reuse" as the "classic" understanding of software components.

Figure 2.1 on the next page illustrates how the resulting market of software components is supposed to work. Component vendors produce software components and sell them on a component market. Application vendors buy these components in order to produce applications, which they in turn sell on an application market. Users, finally, buy these applications and presumably use them to make their lives better.[2] In this model, reuse only occurs *within* individual application vendors. They decide—among other things—what application to produce, which components to buy, which components to develop internally, and how the

[1] It is interesting to speculate on the reasons for this "failure" of software components, but I will keep such speculations to a minimum. Simply put, they sooner or later involve economic, legal, and even political arguments, most of which have a tendency to be (at best) comfortably vague or (at worst) thoroughly misleading. Luckily there are still plenty of technical problems to be solved.

[2] Note that I refer to *roles played* by stakeholders in this model, I do not imply that component vendors, application vendors, and users are necessarily *distinct* entities. For example, the user of a spreadsheet application can simultaneously be the vendor of a spelling checker component and a 3D rendering application, the latter built using an OpenGL component.

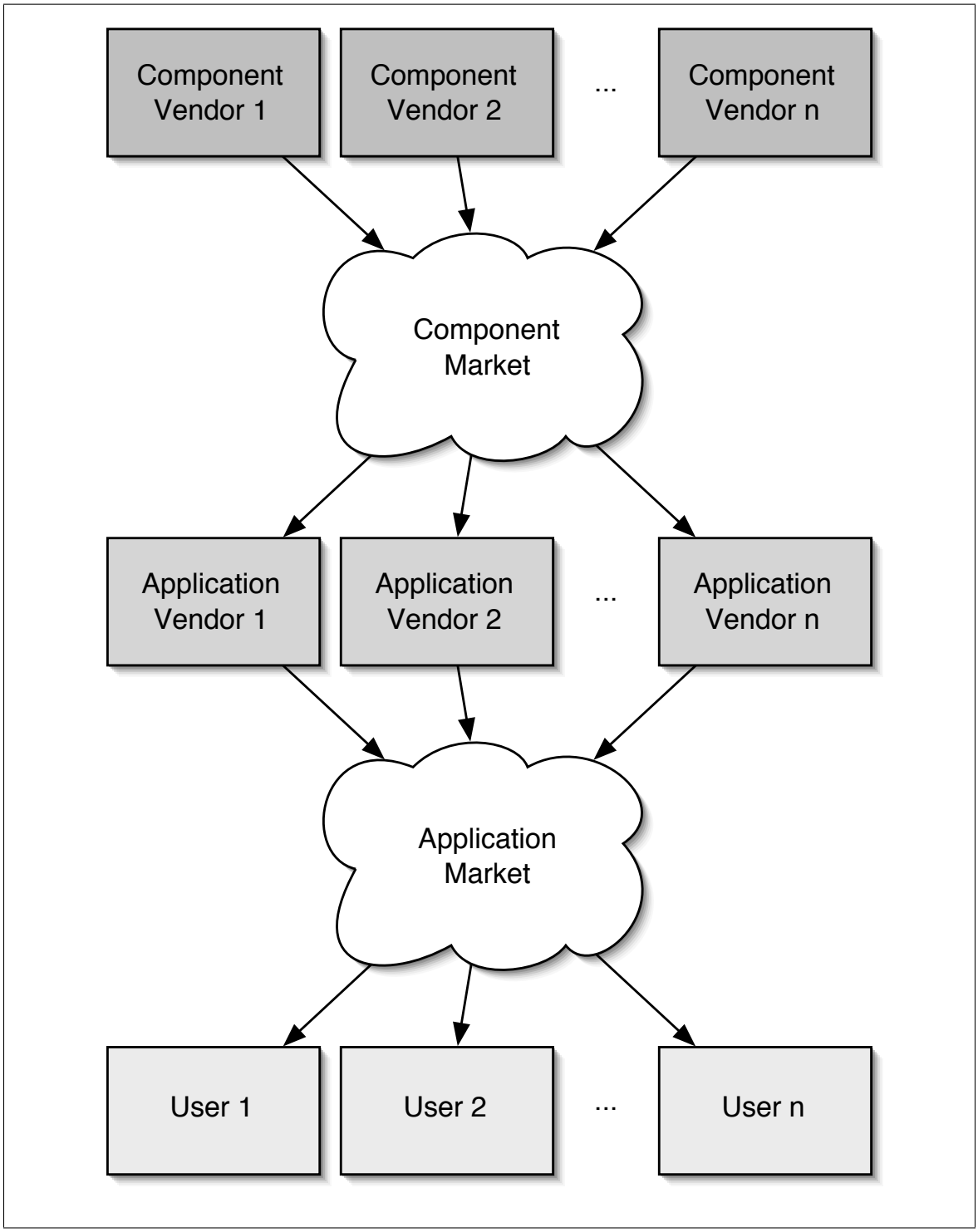


Figure 2.1: A model of the "classic" market for software components in which "centralized reuse" by application vendors dominates. Arrows indicate the flow of software artifacts.

resulting application is distributed and deployed. We can therefore summarize the "classic" understanding of software components as follows:

> **Classic Understanding:** Component software is primarily concerned with *reuse* of software artifacts in a *centralized* setting, where a *single* software vendor has *complete* control over the acquisition and integration of components.

2.1.2 Modern Perspective: Distributed Extensibility

As pointed out above, McIlroy's vision of an "industrial revolution" of software production has not yet been realized on any noticeable scale. In contrast to an "industrial revolution," the development of the modern understanding of component software can be described as a form of "neoliberalism" instead. The "market" is given more flexibility by opening it up to all participants, in hopes of increasing the chances for creating a viable component economy.[3]

Since "composition" is the central motivation for "components" in the first place, this requires breaking the dominance of application vendors. In McIlroy's vision, all composition takes place within application vendors, while users do not have any choice besides buying one or the other application. Figure 2.2 on the following page illustrates how the "modern" market of software components is supposed to work. Component vendors still produce software components and sell them on a component market. However, components are supposed to provide functionality that "something else" is lacking, giving rise to the notion of *component frameworks*. A component framework provides the basic services needed in a certain application domain and prescribes how various components should interact to form a functioning software system (i.e. it provides a domain-specific software architecture). In other words, components extend the functionality of frameworks, while frameworks provide execution environments for components. Software components therefore become units of *extension* instead of units of reuse.

[3] While this description is sensible in retrospect, it is not historically accurate. The developments leading to the renewed interest in component software are often technical rather than economical in nature [SGM02].

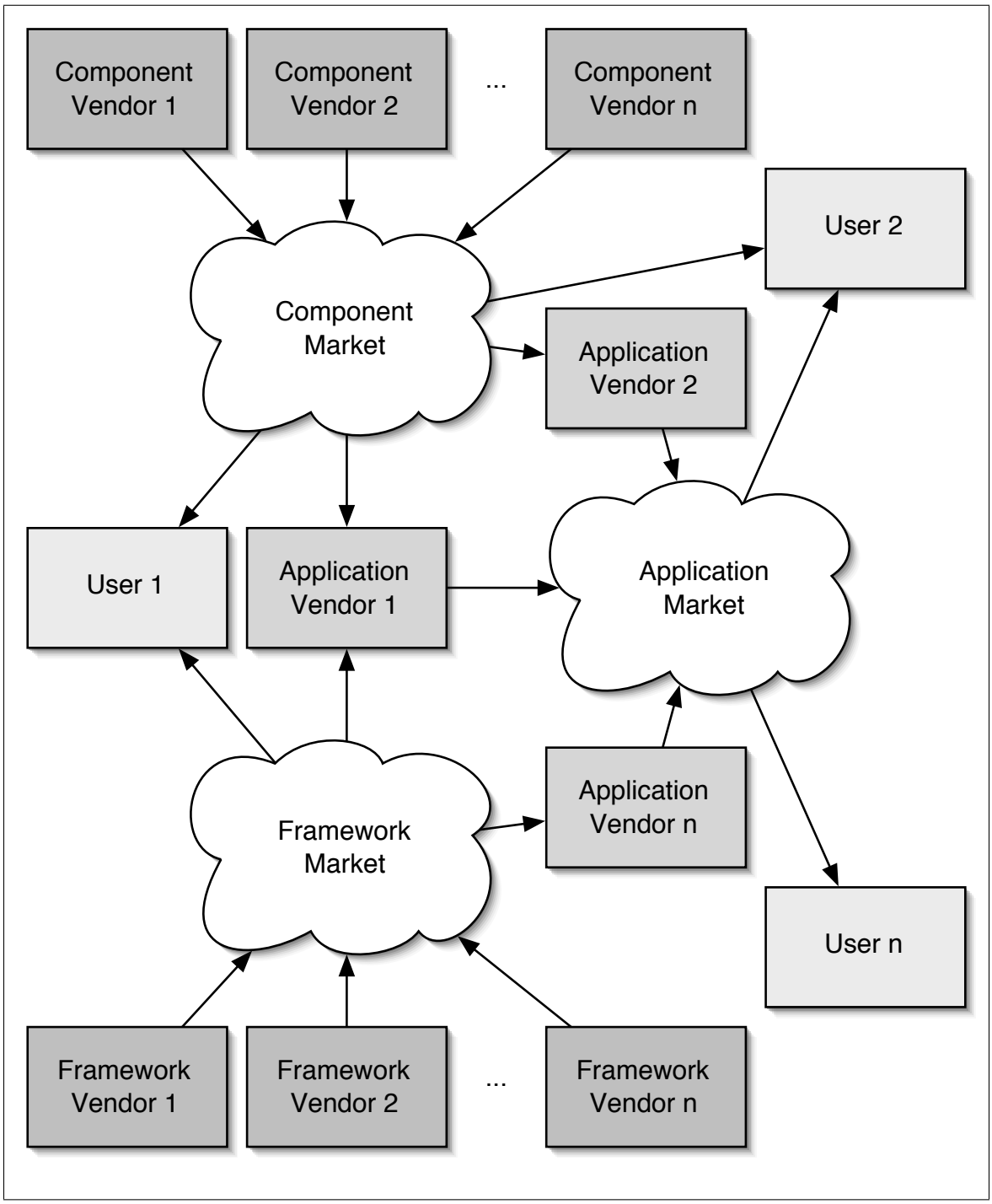


Figure 2.2: A model of the "modern" market for software components in which "distributed extensibility" dominates. Arrows indicate the flow of software artifacts.

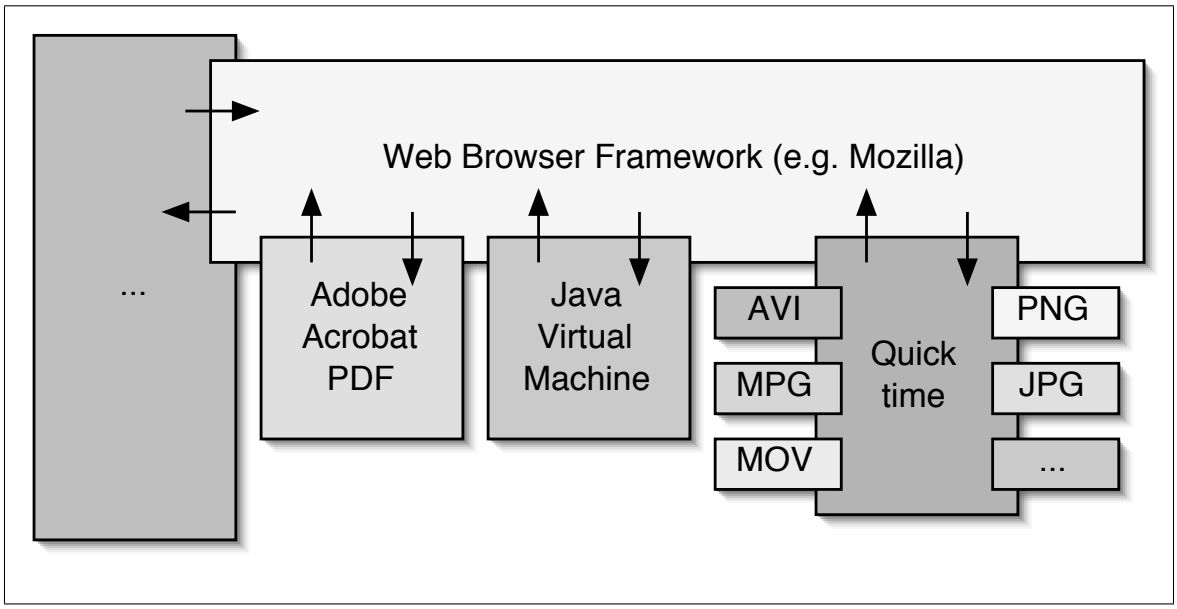


Figure 2.3: Schematic view of a web browser in terms of components and frameworks. Quicktime illustrates hierarchical composition, it is both a component *and* a framework in this example.

The presence of components and framework enables users, and not just application vendors, to purchase the "parts" for a desired software systems from various vendors. Specifically, users can in principle obtain a *complete* system without being tied to application vendors.

Figure 2.3 illustrates this approach to component software. Instead of being a monolithic application, a web browser is first of all a framework responsible for managing network access and screen estate between components that provide user functionality. Composition is *hierarchical* since components of one framework can in their own right be frameworks for further components. The web browser framework is a component for the framework "below" it and relies on its services, while the Quicktime component is a framework for components that provide functionality for specific multimedia file formats.

We can summarize this "modern" understanding of software components as follows:

Modern Understanding: Component software is primarily concerned with the *extensibility* of software systems in a *distributed* setting, where *any* interested party can develop extensions which can be acquired and integrated at *any* time.

2.1.3 Software Development Paradigms

Software engineering is concerned with techniques for the systematic and efficient production of high-quality software [GJM91]. As a discipline within computer science, software engineering covers a broad range of topics ranging from requirements analysis through configuration management to quality assurance. Regardless of the specific techniques employed, however, the result of any software development effort worth its name is—obviously—software, usually expressed as source code in some programming language.

Programming languages therefore share many of the goals that exist for software engineering in general, but they also have more specific goals of their own [GJ97]. Among these, the need for safety and efficiency are of primary importance. As tools for software development, programming languages need to aid programmers in expressing their designs accurately and consistently. Language implementations—compilers as well as interpreters [Wir96, App02]—achieve this goal by performing a variety of automated analyses on the source code supplied by programmers [WM95, NNH99]. Compilers use similar analyses to ensure that source code is translated into native code which makes efficient use of machine resources. In this way, programming languages are the bridge that connects software and hardware, software engineering and computer systems.

Programming languages are also often the most concrete and tangible form in which a particular approach to software development—a software development *paradigm*—is embodied. Structured programming [DDH72, DeM79], for example, encourages us to think of a software system as a process that transforms input data into output data, and which is refined into smaller and smaller subprocesses as development proceeds. Languages that support (i.e. encourage or even enforce)

structured programming, for example Algol 60 [Nau63] or Pascal [JW91], are called *structured* programming languages.

For the paradigms of modular and object-oriented programming—based on the notions of information hiding [Par72], abstract data types [Gut77], and inclusion polymorphism [CW85]—paradigmatic programming languages exist as well. Languages such as Modula-2 [Wir89] and the original version of Ada [Int95] are clearly *modular*, while languages such as Smalltalk [GR83] and Eiffel [Mey92] are clearly *object-oriented*. Programming languages supporting multiple paradigms include CLU [LSAS77, LG86], C++ [Str00], Java [GJSB00], Modula-3 [CDG+91], Oberon-2 [MW91], and Simula [BDMN73, Mag93].

For the emerging paradigm of component-oriented programming, however, no paradigmatic programming language has been developed so far. Instead, languages that combine concepts from modular programming and object-oriented programming—Java [GJSB00], Modula-3 [CDG+91], Oberon-2 [MW91], and Component Pascal [Obe97] for example—are commonly advocated for component-oriented programming [SGM02].[4]

Figure 2.4 on the following page illustrates the evolution of programming language paradigms that this view implies. Modular as well as object-oriented programming adopted certain concepts from structured programming while leaving others behind. For example, they still use a limited number of control structures, but they replace procedures as the sole abstraction mechanism by more advanced ones. Similarly, the paradigm of component-oriented programming can be expected to adopt concepts from previous paradigms while leaving others behind.

It is interesting to note that established software development paradigms have in fact consistently identified components with *their* primary abstraction mechanism (see Figure 2.5 on the next page):

- In structured programming, components are individual *operations* (i.e. *proce-*

[4]The example of Component Pascal [Obe97] is interesting in this regard. The language was designed—and designed well—specifically with component-oriented programming in mind, and it even has a commercial implementation. However, as I argue in Chapters 3–5, it is not paradigmatic in the above sense either.

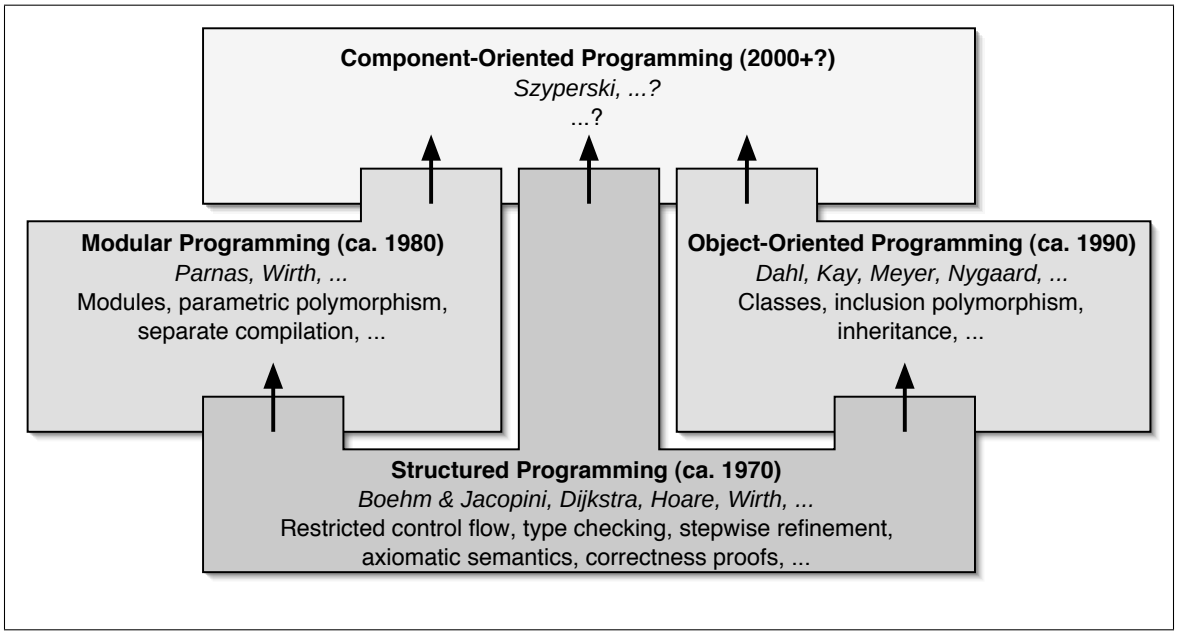


Figure 2.4: A biased view on the evolution of software development paradigms. Arrows indicate the flow of (certain) concepts, dates are approximate and (highly) debatable.

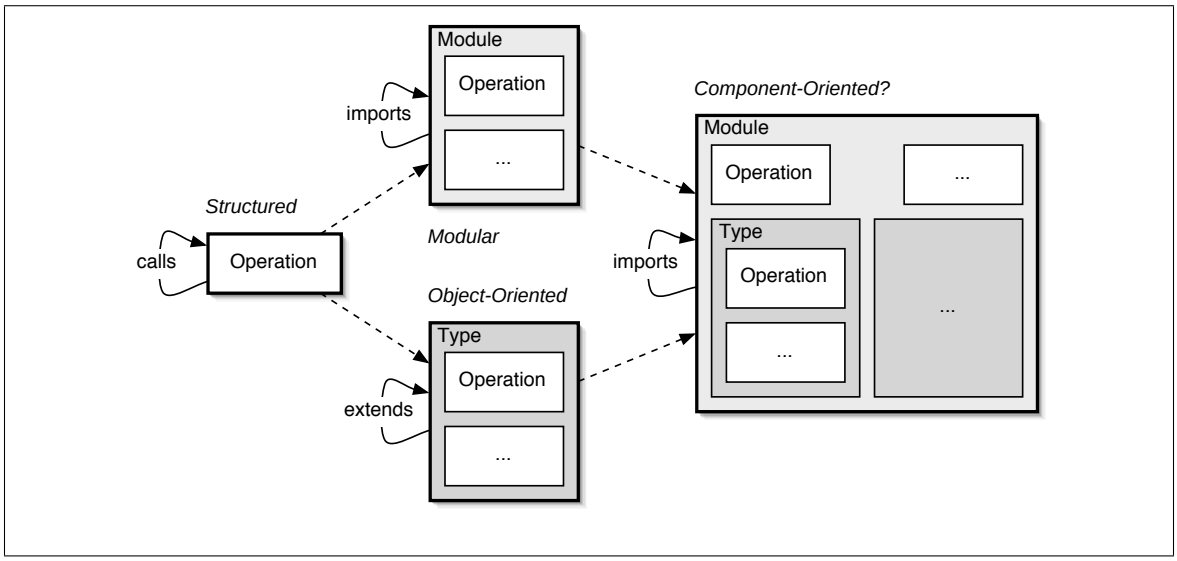


Figure 2.5: The evolution of programming language abstractions for software components. Components were originally considered operations in structured programming, then modules or types that contain operations in modular or object-oriented programming, and now modules that contain operations as well as types.

dures or *functions*); this includes McIlroy's original paper [McI69].

- In modular programming, components are *modules* that encapsulate a collection of related operations.
- In object-oriented programming, components are *types* (i.e. *classes*), that again encapsulate a collection of related operations.

For the emerging paradigm of *component-oriented programming*, a combination of all these abstraction mechanisms has been suggested instead [SGM02]: *Components are modules that encapsulate a collection of operations as well as a collection of types.*

2.2 Component-Oriented Programming Languages

Now that we have a characterization of component-oriented programming as a software development paradigm, we return to the design of component-oriented programming languages. In this section, I essentially repeat the development of the "standard model" for such a language, which can be inferred from [SGM02] as well. However, I present the issues that arise in a compressed and streamlined form suitable for the remainder of the dissertation, and add some remarks on modules that—to my knowledge—have not appeared before.

Support for a certain development paradigm requires a close correspondence between the paradigm's abstractions and those available in a suitable programming language [GJ97]. However, this correspondence does not have to be one-to-one. Structured programming [DeM79], for example, encourages us to think of a software system as a "process" that transforms input data into output data, and which is refined into smaller and smaller "subprocesses" recursively. As long as a programming language offers *some* abstraction capable of these transformations, for example basic procedures or actual processes, it is suitable for structured programming. Thus, while it is tempting to design a language full of explicit abstractions for components, frameworks, connectors, etc., we should first analyze which *existing* language mechanisms can support the necessary requirements. Relying on

proven concepts is an advisable strategy to keep language design manageable and well-founded [Hoa73].

As the essence of component-oriented programming, distributed extensibility should be able to explain the necessary features of an appropriate programming language. A first observation helps us to distinguish what is surely *not* important for such a language. As an *organizational* paradigm, component-oriented programming is concerned with the *composition* and *interaction* of components and frameworks through interfaces, it is not concerned with their *insides* in any way. The *computational* paradigm at the core of a component-oriented programming language is therefore not constrained, i.e. we can choose an imperative, a functional, or a logical core. However, we must restrict ourselves to statically typed languages, otherwise the use of interfaces to ensure the safety of a composition would be impossible to guarantee. In the functional domain, for example, we could choose ML [MTHM97] and Haskell [PJ03], but not Scheme [ADH+98]. In the imperative domain, we could choose Java [GJSB00] and Oberon [RW92], but not Python [vR01].

2.2.1 Modules

The principle of distributed extensibility implies a distinction between *extensions* themselves on the one hand, and whatever they *extend* on the other. In component-oriented programming, these notions are reified as *components* and *framework* respectively. An obvious requirement for this distinction is the ability to *isolate* components and frameworks in such a way that no implicit dependencies remain between them. In programming languages, this requirement can be addressed by *modules*. Modules define the static structure of a system by providing rigid boundaries which can not be crossed arbitrarily. They thus limit the interactions between components and frameworks and make dependencies explicit.

There are, however, a large variety of different module systems available in various programming languages, not all of which are suitable for component-oriented programming. One possible taxonomy for modules classifies them in terms of *access* and *membership* [Car89]:

- Open modules restrict neither access nor membership in any way. From outside a module, all its members can be accessed and new members can be added retroactively.
- Closed modules restrict access but do not restrict membership. From outside a module, only exported members can be accessed but new members can still be added retroactively.
- Sealed modules restrict access as well as membership. From outside a module, only exported members can be accessed and no new members can be added retroactively.

Following this taxonomy, modules must be *sealed* to be suitable for component-oriented programming. In both open and closed module systems, new dependencies that are not explicit in the original module can be created, which defeats distributed extensibility.

Note that Java's `package` construct [GJSB00] provides a closed module system in this sense and is therefore unsuitable as a basis for component-oriented programming. Interestingly, the problems caused by packages have been recognized in Java 1.2 with the introduction of *sealed* packages, which must be distributed as Java archive (jar) files. For a sealed package `A` contained in a file `A.jar` the Java virtual machine guarantees that all classes belonging to `A` have in fact been loaded from `A.jar`. Combined with the capability to cryptographically sign jar files, this achieves the same level of protection that is available in languages that provide sealed modules, but at a much higher complexity.

A number of further issues arise in regard to this basic construct, not the least of which is the confusion of modules and classes. It has been shown that although classes *can* play the role of modules, the two *should* be conceptually different because they serve different purposes [Szy92], and many recent language designs have indeed separated modules from classes. One major reason for this is that modules can package a number of related classes into a single deployable unit, which is required for component-oriented programming.

This in turn raises another question: Since certain components might exceed the complexity that can conceivably be packaged into a single module, should it not be possible to *nest* modules? Aside from a number of semantic difficulties with hierarchical module systems [CHP99]—or nested classes for that matter [IP00]—we have to consider what constitutes a deployable unit again. If nested modules are still deployed individually, nesting becomes irrelevant for distributed extensibility. On the other hand, if nested modules are deployed in one "super module," we might have to distribute the same (source-level) modules a number of times because they are part of different components. A flat module space is conceptually simpler and also has a number of other valuable properties for component-oriented programming [Szy00].

A final concern is the identity of components, and therefore that of modules. Distributed extensibility requires that the presence of a particular extension in a system can not preclude the presence of any other extension.[5] Two otherwise unrelated modules must therefore never have the same name, they must have unique identities. Since no form of "unique identity" can be achieved without *some* convention, our goal should be to make the conventions as unintrusive and transparent as possible. Microsoft's COM [Mic95] uses randomly generated identifiers for this purpose, but these are hardly transparent. A convention similar to that originally proposed for Java seems more convenient in this regard: module names are prefixed with "inverted" Internet domain names, such as `edu.uci.ics.Stack`. Although not enforcable, this convention is a good tradeoff, especially when coupled with an import declaration that can introduce abbreviations.

2.2.2 Types and Polymorphism

A component-oriented programming language needs constructs to express interfaces and implementations and must also support dynamic and independent extensibility. In programming languages, interfaces and implementations should be

[5] The exceptions to this rule are of dynamic nature and concern invariants the system needs to maintain in order to function properly, for example when the extensions are device drivers of an operating system.

modeled as *interface types* and *implementation types* respectively. In this manner, we can define the conformance of an implementation to an interface by the conformance of the corresponding types. Dynamic extensibility requires some form of polymorphism that allows different instances of implementation types to be bound to the same interface types at run-time. Inclusion polymorphism [CW85] in object-oriented languages such as Java [GJSB00] is one way to achieve this, although we prefer the term *implementation polymorphism* in this context.

An interface is an abstraction of all possible *implementations* that can fill a certain role in the composed system [LG86]. It thus describes minimal assumptions that frameworks and components can make about each other. Interfaces are essential to component-oriented programming because they are the only form of coordination between frameworks and components and the only means by which compositions can be validated. We can view interfaces as sets of *messages* (abstract operations) and implementations as sets of *methods* (concrete operations). Messages describe *what* effect is achieved by an operation, while methods describe *how* that effect is achieved. Multiple *instances* of an implementation can exist concurrently, and multiple implementations can be part of a component. We say that an implementation (or an instance) *conforms* to an interface if it provides methods for all messages in that interface. In programming languages, interfaces and implementations should be modeled as *interface types* and *implementation types* respectively. In this manner, we can define the conformance of an implementation to an interface by the conformance of the corresponding types.

Polymorphism supports the dynamic structure of a system by allowing different instances of different implementation types to be bound to the same interface type at runtime. Inclusion polymorphism [CW85] as known from object-oriented languages is one way to achieve this, although we prefer the term *implementation polymorphism* in this context.

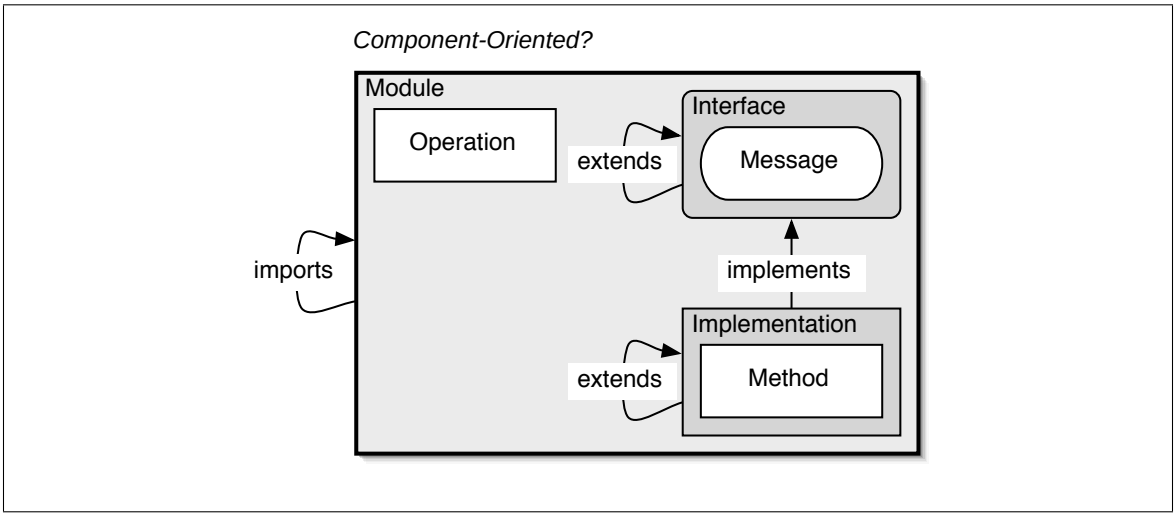


Figure 2.6: The "standard model" for component-oriented programming languages illustrated in the style of Figure 2.5 on page 19.

2.2.3 An Idealized Version of Java

Figure 2.6 summarizes the "standard model" for component-oriented programming languages developed in this section. Sealed modules serve as the elementary component notion, while interfaces and implementations are mapped to types.

Starting from Java [GJSB00] we can now propose a first approximation for a component-oriented programming language. The language is essentially an "idealized" version of Java and we adopt the name $\mathcal{IJ}$ for it for this reason. In $\mathcal{IJ}$, packages are replaced by sealed modules. Imported identifiers are *always* qualified fully by the name of the module that exports them. For convenience, the `import` declaration is modified to allow the introduction of local abbreviations. For example, after the declaration

```
import S = edu.uci.ics.phf.random;
```

we can refer to a class `Standard` exported by this module as `S.Standard` instead of using the more involved

```
edu.uci.ics.phf.random.Standard
```

everywhere. Furthermore, $\mathcal{IJ}$ separates the notions of subtyping and subclassing completely, allowing the hierarchy of interface types to have a different struc-

ture than the hierarchy of implementation types [Sny86, Ame87]. Implementation types declare their conformance to interface types explicitly, and following the Java approach we allow for multiple subtyping but only single subclassing. For completeness, we also replace the notion of static methods with proper procedures declared on the module level.

Note that $\mathcal{IJ}$, besides being a cleaner superset of Java, also subsumes Component Pascal [Obe97], Modula-3 [CDG+91], and Oberon-2 [MW91], which are often regarded as "close approximations" of component-oriented programming [SGM02].

2.3 Scope

The notion of software components, often in the "classic" sense as explained above, appears in a number of areas between software engineering, programming languages, and computer systems. I focus on programming languages in the following, and specifically on the concerns induced by the "modern" view of component software. However, to clarify the scope of my work, I briefly discuss several of the related areas in this section, mainly to explain what this dissertation does *not* address.

2.3.1 Component Models

Component models, such as Microsoft's COM [Mic95], OMG's CORBA [Obj99], and Sun's JavaBeans [Sun97], are industry standards designed to support software components. The main emphasis of these models lies on defining interoperability and packaging conventions in the form of design patterns rather than on providing comprehensive, paradigmatic support. Many component models also address aspects that are essentially unrelated to component-oriented programming itself—such as distribution, concurrency, cross-platform portability, and cross-language integration—but that nevertheless increase their complexity significantly.

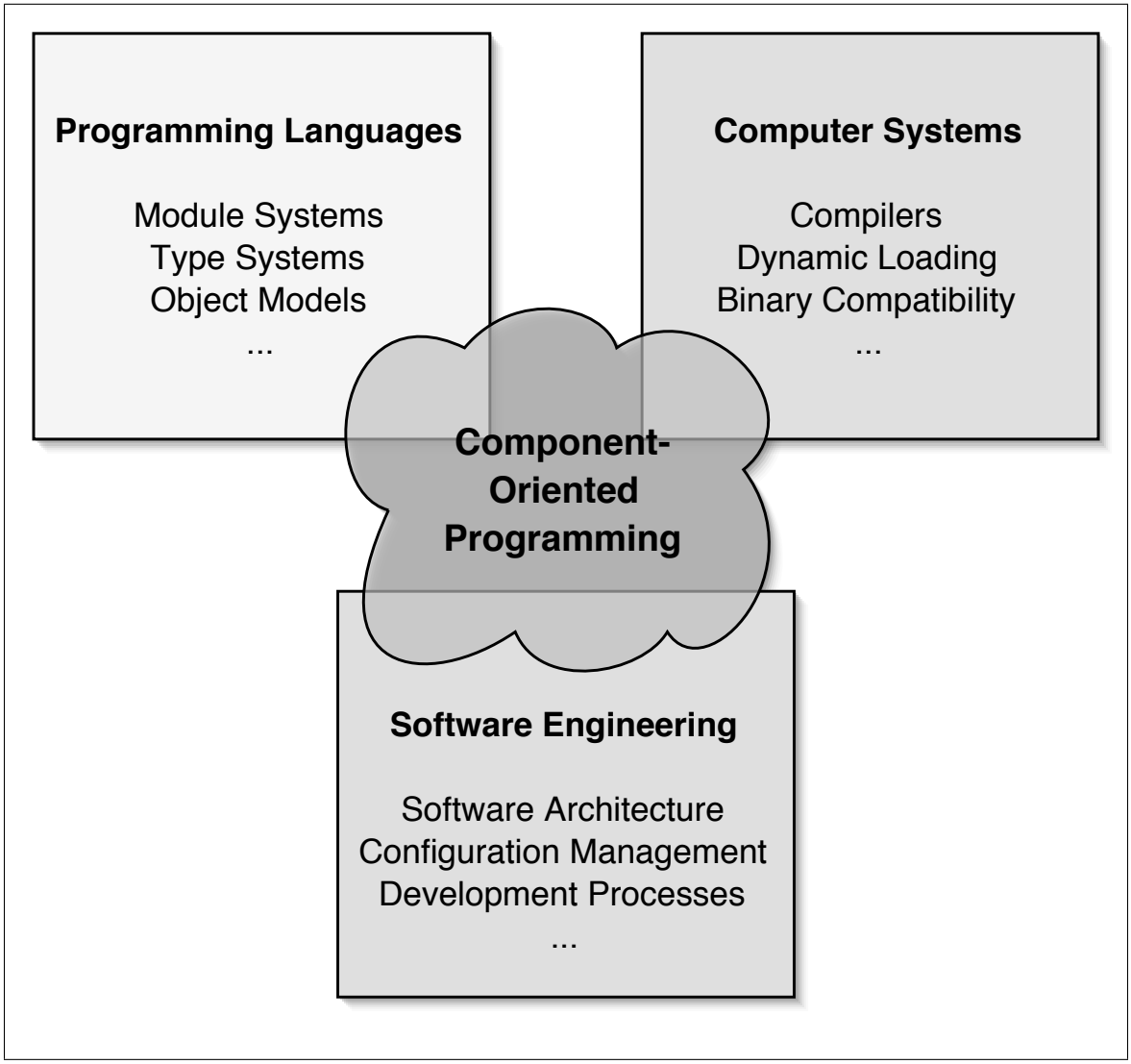


Figure 2.7: Component-oriented programming affects three main "dimensions" of computer science research: programming languages, software engineering, and computer systems.

From the perspective of this dissertation, component models serve a temporary purpose until more comprehensive ways for component-oriented programming emerge. Some of the capabilities offered—especially by COM [Mic95] and its descendant .NET [ECM01]—are indeed valuable and should find their way into programming languages as well. I will discuss these concepts and their relationship to my work in more detail in later chapters.

2.3.2 Generative Programming

The paradigm of generative programming (GP) [CE00] is based on a number of ideas, namely domain-specific programming languages, aspect-oriented programming (AOP), and generic programming. In GP, software systems are described in terms of domain-specific languages that are used to encode domain knowledge on a high level. These descriptions are used to drive AOP [KLM+97] tools that integrate various reusable and basically unrelated "components" and aspects to produce customized applications automatically. The functional "components" are implemented using generic programming techniques (i.e. parametric polymorphism).

While GP provides an interesting approach to source-level reuse and maintenance, its "components" are not components in the sense of component-oriented programming [SGM02]. In GP (and AOP), "components" are reusable and parameterized abstractions that only exist on the programming language level, but not in the deployed application. Thus, once an application has been produced using GP, the "components" it consists of can not be reused or updated separately from the application they were compiled into.

2.3.3 Composition Environments

Composition environments are—frequently graphical—tools focusing on the issue of software composition [LvdH02]. The lines between "regular" software development environments and composition environments are quite fuzzy. However,

the general emphasis of composition environments is not on the development of individual components but rather on their composition into applications or subsystems.

Information about components, especially in terms of interfaces, is used to enforce certain consistency requirements. In this regard composition environments partially compete with the idea of component frameworks, in which consistent composition is enforced through the design of the framework itself and the communication patterns it allows between components. Architecture description languages share similar goals and are sometimes used as part of composition environments, either to guide composition or to record the details about a particular configuration of components.

Composition environments are dominated by higher-level concerns than those I discuss in this thesis. In developing a programming language for component-oriented programming, I focus on the possible foundation that such environments could be built on. In other words, instead of making composition *easier*, I investigate how to make composition *possible at all*, especially in the way mandated by distributed extensibility.

Chapter 3

Stand-Alone Messages

> *1. Everything is an* object. *2. Objects communicate by sending and receiving* messages *(in terms of objects). 3. Objects have their* own memory *(in terms of objects). . . .*
>
> — ALAN C. KAY [Kay96]

In this chapter, I develop the concept of *stand-alone messages*, the first novel language mechanism in my design framework for component-oriented programming languages. I start by motivating the need for software components to conform to multiple interfaces using a realistic example (Section 3.1). Switching to a simpler example for clarity, I then introduce the problem of *interface conflicts* and exhibit several shortcomings of programming languages following the standard model (Section 3.2). I trace these shortcomings to the status of messages in object-oriented programming languages and argue that messages should be independent of types, leading to the concept of stand-alone messages (Section 3.3). Finally, I evaluate stand-alone messages by comparing them to a variety of existing approaches for resolving interface conflicts (Section 3.4).

3.1 Motivation

As discussed in Section 2.1.2, interfaces play a central role in component-oriented programming. Components rely on interfaces implemented by frameworks to ac-

cess their services, while frameworks in turn rely on interfaces implemented by components to access theirs.

For technical as well as economic reasons, software components often need to conform to *multiple* interfaces. Consider, say, a component that presents the result of a database query within a compound document [Wec96], a scenario illustrated schematically in Figure 3.1 on the next page On the technical side, instances of this component have to react to notifications from both the database management framework *and* the compound document framework to keep their presentations current:

- After a change in the database, the component must update its presentation (if the query is persistent).
- After a change in the document, the component must update its presentation (and potentially the database).

On the economic side, the component will increase its potential market if it can be composed with a variety of frameworks for database management and compound documents.

The principle of distributed extensibility requires that any interested party can develop a component extending the functionality of any given framework. In particular, it neither rules out components that extend multiple frameworks simultaneously, nor does it restrict such components to extend only certain subsets of possible frameworks.[1]

For component-oriented programming languages, this requires that an implementation type can conform to *any number* of interface types or, equivalently, that *any combination* of interface types is again a valid interface type. As I am about to show, this requirement is *not* fulfilled by the standard model for component-oriented programming languages (see Section 2.2).

[1] The problem of *framework combination* [MB97] starts from slightly different assumptions but can be reduced to the same underlying issue.

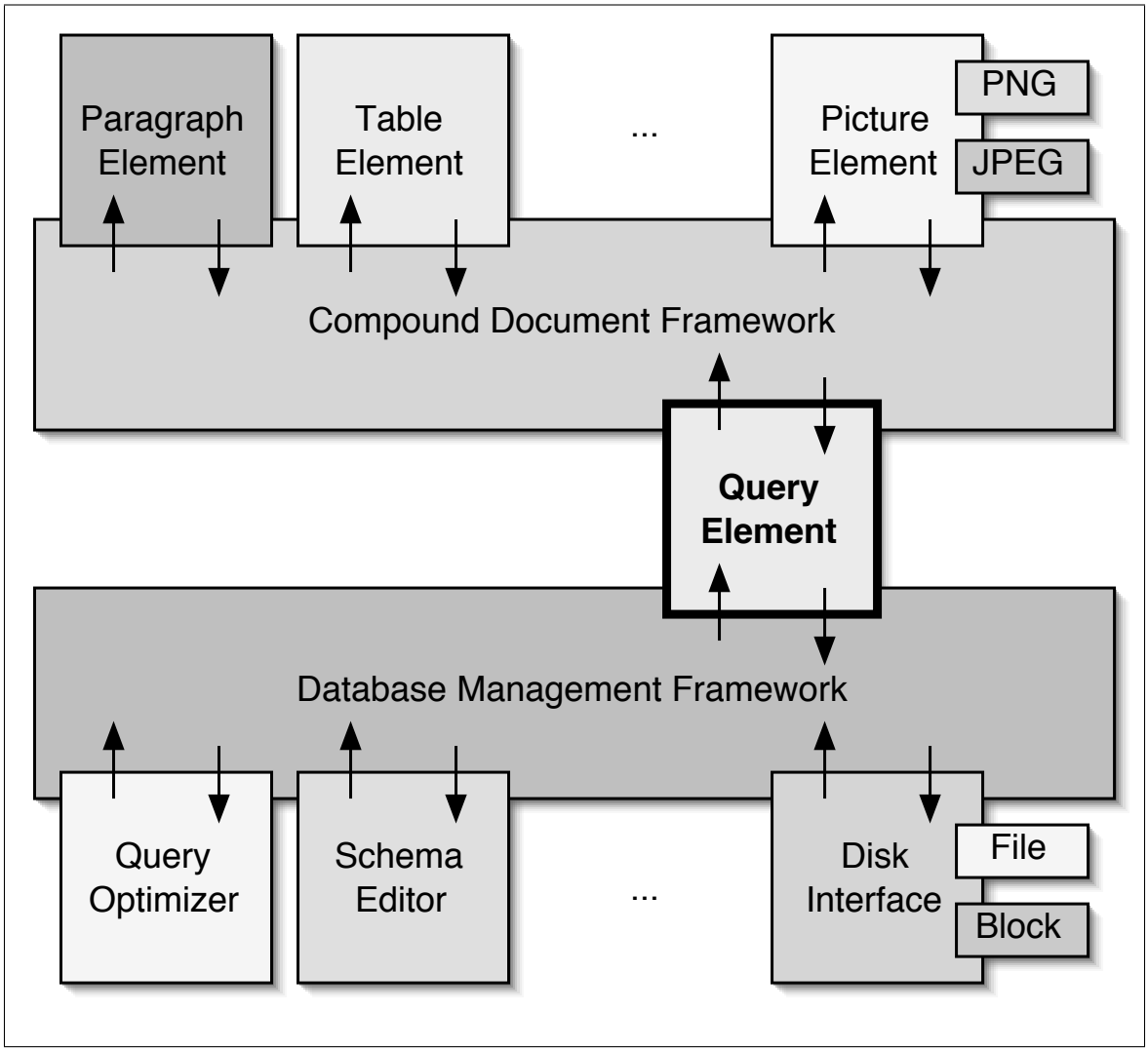


Figure 3.1: Schematic view of a software component that needs to conform to a compound document framework and a database management framework simultaneously. Other components are for illustration only.

adt Stack **aka** UnboundedStack

uses
Any, Boolean

defines
Stack<Element: Any>

operations
new: $\rightarrow$ Stack<Element>
empty: Stack<Element> $\rightarrow$ Boolean
push: Stack<Element> $\times$ Element $\rightarrow$ Stack<Element>
pop: Stack<Element> $\nrightarrow$ Stack<Element>
top: Stack<Element> $\nrightarrow$ Element

preconditions
pop(s): not(empty(s))
top(s): not(empty(s))

axioms
empty(new())
not(empty(push(s, e)))
top(push(s, e)) = e
pop(push(s, e)) = s

Figure 3.2: An algebraic specification of the abstract data type Stack. Except for the type parameter Element with its obvious meaning, the notation follows [Mey97].

3.2 Interface Conflicts

In the following, I use a simple example based on the "infamous" abstract data type (ADT) Stack to illustrate the problem of interface conflicts in detail.[2] For reference, Figure 3.2 provides a standard algebraic specification of this ADT, using a variation of MEYER's notation [Mey97]. The code examples below are given in $\mathcal{IJ}$, the idealized version of Java outlined in Section 2.2.

Consider a component vendor who decides to specialize in `Stack` components. Given the ubiquity of `Stack` implementations in existing libraries and even textbooks, our vendor has to support a *very* large number of frameworks to sell any `Stack` components at all. Assume the first framework defines the interface shown in Figure 3.3 on the next page. The design of this interface follows the textbook

[2] While Stack is "infamous" for having been "overused" in the past, it still serves as an easily understood abstraction exhibiting most of the problems also found in more complex scenarios.

```
module edu.uci.framework {
  public interface Stack {
    // pre o != null; post top() == o;
    public void push( Object o );
    // pre !empty();
    public void pop();
    // pre !empty(); post return != null;
    public Object top();
    // "no elements?"
    public boolean empty();
  }
}
```

Figure 3.3: An interface for the basic stack abstraction in $\mathcal{IJ}$. It is meant to express the semantics from Figure 3.2 on the page before, but closer to an actual implementation.

definition ADT Stack closely, and developing an implementation of the interface, for example in terms of a linked list, is straightforward.

The interface defined by the second framework is given in Figure 3.4 on the following page. Instead of relying on an `empty` message, this interface works with the `size` of the stack, i.e. the number of elements it currently contains. To support this interface in addition to the one from Figure 3.3, our component vendor must add a `size` method which is again straightforward. The interfaces are *compatible* because they only differ in their use of `empty` and `size` respectively, and we can express one in terms of the other using the identity

```
empty() == (size() == 0)
```

as an abstraction function [LG86]. If we apply this abstraction function to the specification, the precondition and postconditions listed as comments become identical. For reference, Figure 3.5 on page 36 gives an implementation of these first two interfaces.

```
module gov.nsa.framework {
  public interface Stack {
    // pre o != null; post top() == o;
    public void push( Object o );
    // pre size() > 0
    public void pop();
    // pre size() > 0; post return != null;
    public Object top();
    // post return >= 0; "how many elements?"
    public int size();
  }
}
```

Figure 3.4: Another stack abstraction in $\mathcal{IJ}$, compatible with the previous one (see Figure 3.3 on the preceding page). Since `empty()` can be expressed in terms of `size()`, a single implementation type can conform to both interface types.

3.2.1 Syntactic Conflicts

Our vendor now decides to support the interface shown in Figure 3.6 on page 37 in addition to the previous two. This new interface follows an alternative specification of the ADT Stack, in which the pop operation not only removes the top element but also returns it. Compared to the previous two interfaces, there is no `top` message, and the signature of `pop` has changed. To support this interface as well, the `Stack` implementation would need *two* methods for `pop` with *different* signatures. Even if we assume that $\mathcal{IJ}$ includes Java's overloading mechanism, it is impossible to add this interface to the previous two.

For good reasons, Java does *not* allow methods to be overloaded on their return type, which is what would be required here.This is an example for a *syntactic* interface conflict, violating the principle that any combination of interface types should again be a valid interface type. Programming languages such as $\mathcal{IJ}$, which follow the standard model from Section 2.2, therefore do not support distributed extensibility and are not suitable for component-oriented programming.

```
module org.bloat.components {
  import US = edu.uci.framework, GS = gov.nsa.framework;
  class Link {
    Object object; Link next;
  }
  public class Stack implements US.Stack, GS.Stack {
    Link top; int sz;
    public void push( Object o ) {
      Link x = new Link(); x.object = o;
      x.next = this.top; this.top = x;
      this.sz += 1;
    }
    public void pop() {
      this.top = this.top.next;
      this.sz -= 1;
    }
    public Object top() {
      return this.top.object;
    }
    public int size() {
      return this.sz;
    }
    public boolean empty() {
      return this.size() == 0;
    }
  }
}
```

Figure 3.5: An implementation of the two compatible interfaces from Figure 3.3 on page 34 and Figure 3.4 on the preceding page in $\mathcal{IJ}$.

```
module com.sun.framework {
  public interface Stack {
    // pre o != null; post top() == o;
    public void push( Object o );
    // pre !empty(); post return != null;
    public Object pop();
    // "no elements?"
    public boolean empty();
  }
}
```

Figure 3.6: A Stack abstraction causing a syntactic conflict in $\mathcal{IJ}$.

3.2.2 Semantic Conflicts

Having failed to support one interface, our component vendor now desperately tries to support another. This fourth and final interface is given in Figure 3.7 on the following page. Except for the additional `size` message, this interface is identical to the first from Figure 3.3 on page 34. Unlike `size` in the second interface, however, this one returns the *number of remaining* `push` *operations* before some presumably expensive internal restructuring occurs.[3] While both `size` messages have identical signatures—and are therefore syntactically indistinguishable—their semantics are quite different. To support this interface as well, the `Stack` implementation would need two *different* methods for `size`, one returning the number of elements and one returning the number of remaining slots, but both having *identical* signatures.

Obviously, no amount of overloading in $\mathcal{IJ}$ will allow our vendor to accomplish this feat. This is an example of a *semantic* interface conflict, and like syntactic conflicts before, it violates the principle of distributed extensibility. When interface types are combined, the resulting interface type must *preserve all constituent messages*, which is not the case in languages that follow the standard model.

[3] This information might be necessary in a framework with real-time constraints, and implementations based on incrementally growing arrays can supply it easily.

```
module org.cthulhu.framework {
  public interface Stack {
    // pre o != null; post top() == o;
    public void push( Object o );
    // pre !empty();
    public void pop();
    // pre !empty(); post return != null;
    public Object top();
    // "no elements?"
    public boolean empty();
    // post return >= 0; "how many pushes?"
    public int size();
  }
}
```

Figure 3.7: A stack abstraction introducing a semantic conflict in $\mathcal{IJ}$.

3.2.3 Discussion

The stack example I have used above to illustrate interface conflicts might seem overly simplistic. On the one hand, few vendors would ever consider actually entering the "market" for stack components, and most likely such a "market" would not even exist in the first place. However, the complexity of the example used does not affect the validity of the conclusions drawn. If a problem can be demonstrated using a small example, it is obviously possible to find bigger examples that exhibit it as well.

On the other hand, a number of "obvious" solutions for avoiding interface conflicts immediately come to mind, some of which I discuss in more detail below (see Section 3.4). For example, we could use the Adapter pattern [GVJH95] and implement five classes inside the stack component, four of which would simply act as "placeholders" for the fifth, which contains the actual implementation (see Figure 3.8 on the following page). However, the point here is not whether it is possible to resolve the problem in other ways once it is detected, or even that we can make it "less likely" to occur. Instead, we must prevent it from *ever* occurring. If any chance for an interface conflict remains, it will rule out some combination

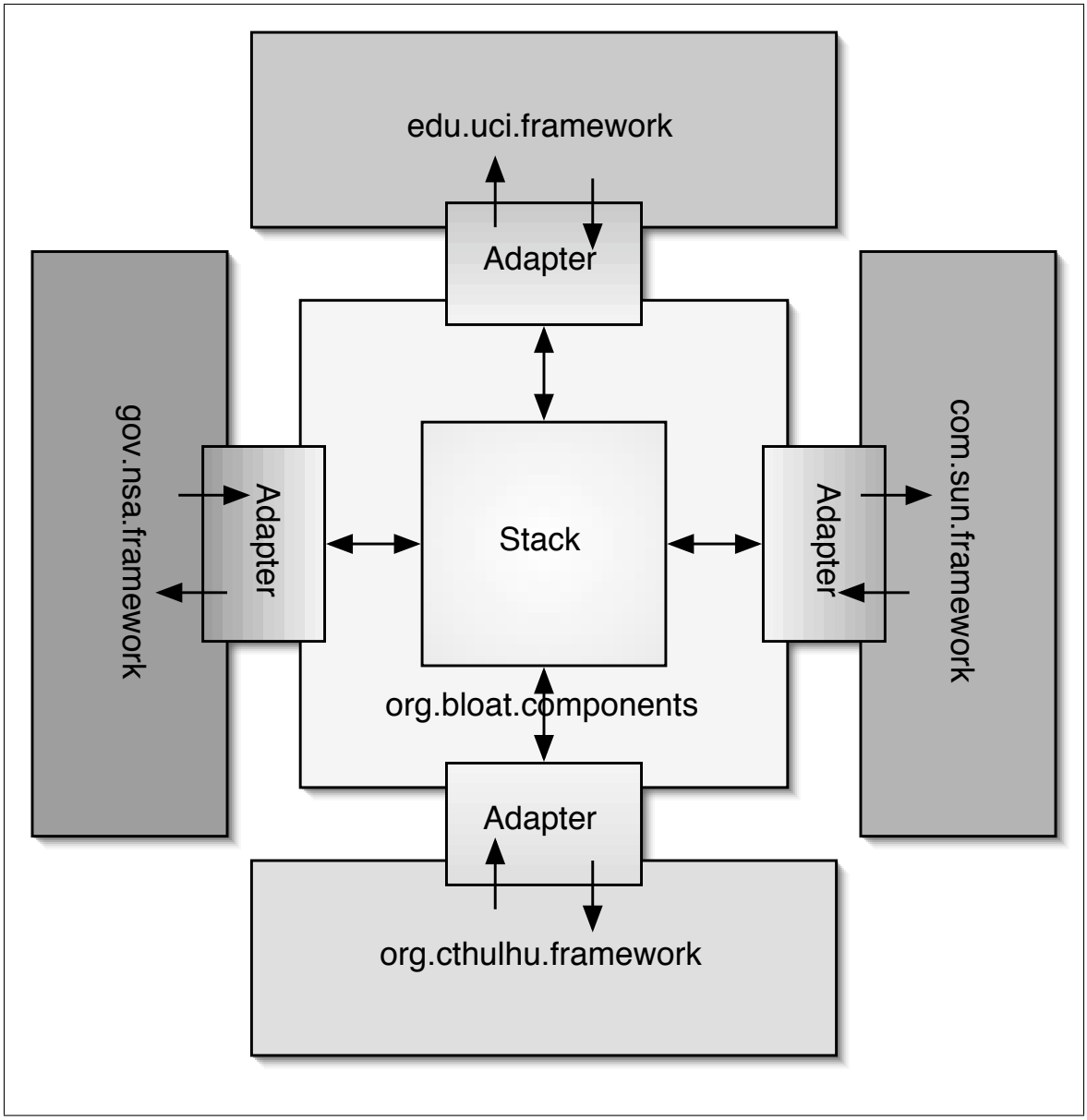


Figure 3.8: Resolving interface conflicts using adapters [GVJH95]. The stack component now consists of five classes, one for the actual implementation (center), and four adapter classes, one for each framework.

of interfaces that—sooner or later—someone will want to perform, thus violating distributed extensibility.

3.3 Rethinking Messages

The problem of interface conflicts discussed in Section 3.2 is not specific to $\mathcal{IJ}$. I made only very general assumptions about the "ingredients" for component-oriented programming languages in Section 2.2 where the basics of $\mathcal{IJ}$ were outlined. The following analysis therefore applies to many existing programming languages.

3.3.1 Analysis

Interface conflicts, both syntactic and semantic ones, can arise whenever two or more interfaces are combined into a new one. Looking at this process in terms of messages, we observe the following:

- Syntactic conflicts can only arise between messages with *identical* names and *different* signatures.
- Semantic conflicts can only arise between messages with *identical* names and *identical* signatures.

The problem can therefore be reduced to the issue of *naming* messages: Under what conditions can we *identify* a message *uniquely* given its *name*?

In most object-oriented programming languages—certainly in established ones such as C++ [Str00], Eiffel [Mey92], Java [GJSB00], and Smalltalk [GR83]—the name of a message only identifies it uniquely *within* the type containing its declaration. When we combine several types $T_1, \ldots, T_n$ to form a new type T, we therefore have to require that *all* constituent messages can *still* be identified uniquely in T, regardless which type introduced them originally. Figure 3.9 on the next page illustrates this approach. Interface types are "boxes" containing messages, and messages have unique identities inside their interface types (a). During interface

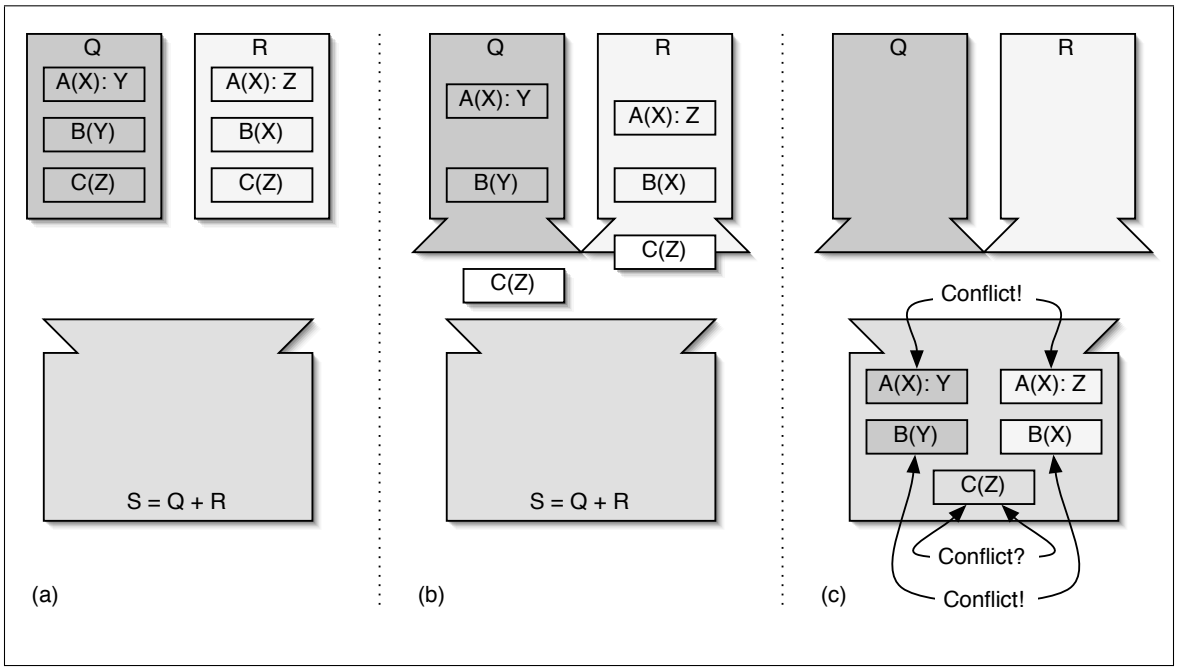


Figure 3.9: Interface combination in object-oriented programming languages. Messages "fall" out of their respective interface types Q and R into a new interface type S, losing their identity in the process.

combination, messages "fall" out of their respective interface types, and lose their unique identity in the process (b). When they "land" inside the new interface type, syntactic as well as semantic conflicts can occur (c). It should be obvious that giving messages unique identities only *within* types by not *across* types is the cause for syntactic as well as semantic interface conflicts.

Before proposing a new approach to the identity of messages *across* types, it is worth pointing out that there is still a need for identity *within* types, namely in the case of methods and implementation types. Consider the example given in Figure 3.10 on the following page. After we bind an instance of `ArrayStack` to the interface reference `stack`, we expect the message `push` to invoke the *specific* `push` method declared for `ArrayStack`. Similarly, after we rebind an instance of `ListStack` to the reference, we expect the *same* message `push` to invoke a *different* `push` method declared for `ListStack`. In other words, whenever the implementation type of the instance bound to the `stack` reference changes, we

```
...
edu.uci.framework.Stack stack;
...
stack = new edu.uci.components.ArrayStack( 16 );
stack.push( new Integer( 1 ) );
...
stack = new edu.uci.components.ListStack();
stack.push( new Integer( 1 ) );
...
```

Figure 3.10: An example for implementation polymorphism $\mathcal{IJ}$. When we send a `push` message through an interface reference, we expect the `push` *method* invoked to change depending on the *implementation type* of the instance.

want the identity of the methods invoked through that reference to change as well. In fact, it is this kind of *implementation polymorphism* that motivated the choice of object-oriented concepts for component-oriented programming languages in the first place (see Section 2.2).

3.3.2 Synthesis

Returning to messages and their identities, our goal must be to somehow "detach" messages from interface types. Since methods have to remain relative to implementation types for polymorphism to work, this will break the traditional "symmetry" between messages and methods.

In the standard model for component-oriented programming languages (see Section 2.2), the only reasonable language construct other than types to "attach" messages to is the *module*. To emphasize the difference to messages in existing object-oriented languages, we choose the name *stand-alone* messages for this concept. Figure 3.11 on the next page illustrates how stand-alone messages would be used to express the first `Stack` interface from Figure 3.3 on page 34. At first, this example does not seem very different from the original form of the interface. However, in client modules that import `edu.uci.framework`, the type `Stack` will now appear as shown in Figure 3.12 on the next page, with each constituent

```
module edu.uci.framework {
  // pre o != null; post top() == o;
  public message void push( Object o );
  // pre !empty();
  public message void pop();
  // pre !empty(); post return != null;
  public message Object top();
  // "no elements?"
  public message boolean empty();
  public interface Stack { push, pop, top, empty }
}
```

Figure 3.11: An interface for the basic stack abstraction using stand-alone messages. In contrast to Figure 3.3 on page 34, messages are declared in the `module` scope, not in the `interface` scope.

```
interface edu.uci.framework.Stack {
  edu.uci.framework.push, edu.uci.framework.pop,
  edu.uci.framework.top, edu.uci.framework.empty
}
```

Figure 3.12: The interface type from Figure 3.11 as it appears in client modules. All constituent messages are qualified by a `module` name.

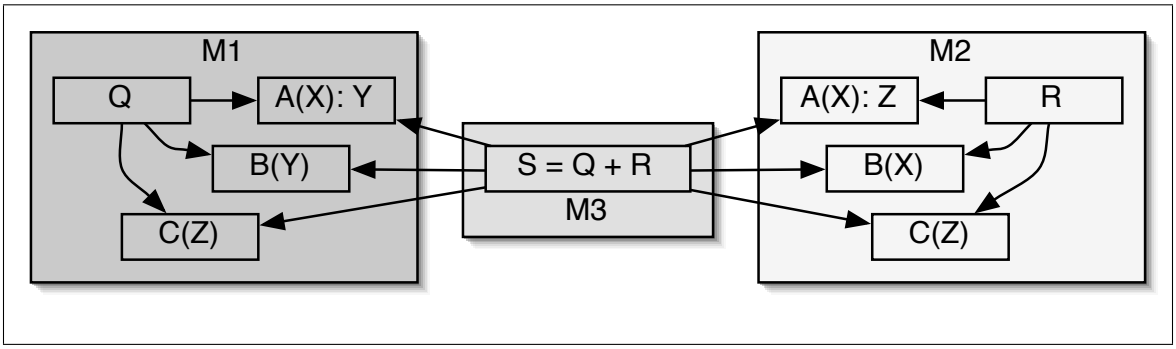


Figure 3.13: Interface combination for component-oriented programming languages using stand-alone messages.

message qualified by a module name. At this point, it should be obvious that stand-alone messages solve the problem of interface conflicts, and that any combination of interface types is indeed again a valid a valid interface type preserving all constituent messages. Figure 3.13 illustrates the process of interface combination in a language that supports stand-alone messages.

3.4 Evaluation

To evaluate the concept of stand-alone messages, I compare them to a number of existing approaches for resolving the problem of interface conflicts. I focus on approaches that do not introduce language mechanisms beyond object-oriented programming first: component models, programming conventions, and design patterns. Then I turn to approaches that do require mechanisms beyond the basic ingredients of object-oriented programming: explicit qualification of messages, renaming messages, and overloading messages. Finally, I summarize my results.

3.4.1 Component Models

Microsoft's COM is the component model that is most similar to our approach [Mic95]. Instead of assigning unique identities to messages, COM assigns unique identities to interface types. Instead of relying on a transparent naming convention for modules, COM associates an automatically generated *globally unique identifier*

(GUID) with each interface type. Contrary to most object-oriented programming languages, COM allows an implementation type to conform to multiple interface types *without* any conflicts. Combined interface types can also be expressed using COM's *category* mechanism.

While we emphasize explicit programming language support and the associated advantages, the two approaches are equivalent as far as interface conflicts are concerned. In particular, we could map stand-alone messages to singleton COM interfaces and interface types to COM categories.

3.4.2 Programming Conventions

A variety of programming conventions can be suggested to address interface conflicts. Defining naming conventions for messages is one of the simplest. The message `push` in the interface `Stack` in the module `edu.uci.framework` could by convention be named `edu_uci_framework_Stack_push`. While theoretically possible, we do not believe that such a convention is acceptable in practice. Additional mechanisms for introducing short local names for messages would be needed, complicating the resulting language. However, even if we accept this complication, we must define *new* conventions on how names should be abbreviated if we are concerned about readability. More complex programming conventions have been suggested as well [BW00].

A general problem with programming conventions is that they are not enforcable by the compiler. This applies to programming languages based on stand-alone messages as well, since we rely on module names that are unique by convention. However, no form of "globally unique identity" can be achieved without *some* convention, so our goal should be to make the conventions as unintrusive and transparent as possible. We believe that, in light of these considerations, conventions for module names are a good tradeoff.

3.4.3 Design Patterns

Certain design patterns can be used to resolve interface conflicts [GVJH95]. In a variation of the Command pattern, "messages" are modelled as a hierarchy of classes containing "parameter slots," while "message sends" are calls to a universal dispatch method. The dispatch method performs explicit run-time type-tests and calls the actual method corresponding to the dynamic type of the "message." This approach relies on the compiler to generate unique type descriptors for each class and thus prevents any conflicts between messages. However, static type-checking is not possible to the desirable extent.[4]

Variations of the Adapter, Bridge, and Proxy patterns can be used to map multiple conflicting interface types to a single implementation type. The idea is to insert additional forwarding classes between clients of an interface type and its implementation type. Messages sent to the forwarding class are routed to the corresponding method in the implementation. While this approach preserves static type-checking, it can be tedious to write the required forwarding classes without tool support.

3.4.4 Explicit Qualification

C++ supports the explicit qualification of member functions by classes to avoid name clashes [Str00]. In our terminology, message sends can be qualified by the implementation type in which a method should be invoked. As defined in C++, this mechanism does not support implementation polymorphism as required for component-oriented programming.

However, we can generalize the idea of explicit qualification by allowing message sends to be qualified by interface types. Although this does not restrict polymorphism anymore, even a qualified message of the form `Stack.pop` is not necessarily unique, since multiple interface types with identical names could exist.

[4] Interestingly, stand-alone messages were originally inspired by this design pattern from the Oberon system [WG92]. Language constructs for messages appeared in Object Oberon [MTG89], the protocols extension for Oberon [Fra95], and finally Lag*oona* [Fra97b].

Therefore, qualification must be extended to include module names as well, at which point the mechanism becomes equivalent to stand-alone messages, except for the redundant interface type.

3.4.5 Renaming Messages

In Eiffel, features inherited from ancestor classes can be *renamed* in a descendant class to avoid name clashes [Mey92]. In our terminology, an implementation type conforming to multiple interface types can explicitly choose new local names for conflicting messages. Note that clients still use the messages declared in the original interface type, but the messages are "rerouted" in a way similar to the Adapter design pattern described above.

Although renaming can be used to resolve interface conflicts, the approach has two major drawbacks. First, renaming clutters up the name space of the implementation type. We may have to invent a new name for a message that is less expressive than the original one, define naming conventions to keep readability up, and repeat this "renaming excercise" whenever we want to conform to an additional interface type. Second, renaming must be extended to combined interface types in addition to implementation types. This becomes particularly clumsy in terms of syntax if we also want to support anonymous interface types.

3.4.6 Overloading Messages

Overloading is a form of ad-hoc polymorphism [CW85] supported by a number of programming languages such as Java [GJSB00] and C++ [Str00]. In our terminology, overloading essentially encodes parts of the signature of a message within its name and uses contextual information available when a message is sent to determine which *actual* message is being referred to.

Although overloading helps to avoid some interface conflicts, it has two major limitations. First, semantic conflicts can not be avoided by overloading since the semantics of a message can not be expressed by type systems in which type check-

ing is decidable [Sch95]. Second, avoiding *all* syntactic interface conflicts requires *all* combinations of parameter and return types to be distinct. This is not generally possible in the presence of subtyping and the coercions it implies.

3.4.7 Summary

Stand-alone messages break the symmetry between messages and methods that exists in object-oriented languages. Binding messages to sealed modules instead of binding them to extensible types allows interface combination without *any* possibility for interface conflicts. It also leads to the following interesting property:

> **Interface Combination:** Any combination of interface types is again a valid interface type preserving all constituent messages.

In other words, using stand-alone messages, the set of interface types is *closed* under interface combination.

Stand-alone messages provide a *simpler* solution to the problem of interface conflicts than those commonly available in other languages. Neither overloading of messages nor explicit qualification in the style of C++ [Str00] provide a general solutions in the first place. The latter can be extended to the point where it becomes equivalent to stand-alone messages if we disregard the redundant interface type name. Renaming allows all interface conflicts to be resolved, at the price of requiring a "fresh" supply of names every now and then. None of these mechanisms, however, actually solves the problem in the right way component-oriented programming, namely by avoiding it.

Stand-alone messages might, however, affect *flexibility* in a negative way. Since messages are now globally unique, it is impossible to "unify" any two messages retroactively, even if they specify identical syntax (signature) and semantics (specification). This could conceivably lead to an "explosion" of messages in the long run. There are a number of points to be made about this. A first observation is that this is simply the price we have to pay to avoid interface conflicts. If there was a way to "unify" messages explicitly, this would necessarily introduce the potential for semantic conflicts through the back door. A second observation is that

under the market assumptions of component-oriented programming, a relatively stable number of widely known and used messages will form sooner or later. A third and final observation is that "unification" of messages has *no* problematic consequences if such a decision remains strictly local within modules. I explore this third option further in Chapter 5 and Chapter 7.

In terms of *safety* and *efficiency*, stand-alone messages do not have any particular advantages or disadvantages.

In retrospect, it seems that KAY's 1972 summary of object-oriented programming quoted at the beginning of this chapter had the status of messages "right" for component-oriented programming, while most object-oriented programming languages—including KAY's own Smalltalk—have it "wrong" to varying degrees.

Chapter 4

Generic Message Forwarding

> *Though delegation has been the minority viewpoint in object oriented languages, it is slowly becoming recognized as important for its added power and flexibility.*
>
> — HENRY LIEBERMAN [Lie86]

In this chapter, I develop the concept of *generic message forwarding*, the second novel language mechanism in my design framework for component-oriented programming languages. I start by motivating the need to adapt and customize existing software components to conform to new interfaces using a realistic example (Section 4.1). Switching to a simpler example for clarity, I then introduce the *fragile base class* problem and exhibit several shortcomings of programming languages following the standard model (Section 4.2). I trace these shortcomings to the use of inheritance and delegation in object-oriented programming languages and argue that these mechanisms should be replaced, leading to the concept of generic message forwarding (Section 4.3). I then compare the expressiveness of forwarding as a mechanism for component adaptation to inheritance and delegation (Section 4.4). Finally, I evaluate generic message forwarding by comparing it to a variety of existing approaches for solving the fragile base class problem (Section 4.5).

4.1 Motivation

In Chapter 3, our focus was on enabling interface combination in a way that preserves distributed extensibility. In component-oriented programming, interfaces are the primary means of coordination between otherwise independent component vendors and framework vendors. Interfaces ensure—to the extent possible—that compositions of frameworks and components result in properly functioning software systems.

For any number of reasons, however, a software component might not support the *exact* interface required by some framework we would like to compose it with:

- The framework involved might not be widely used and the component vendor therefore had no incentive to support it explicitly.
- The framework or the component involved might be "legacy" software in the sense that no party is maintaining them anymore.
- A sufficiently powerful component vendor might decide *not* to support certain frameworks for political reasons.

Consider, say, a (very) sophisticated spell checking component that detects defective proofs in doctoral dissertations. We might need this capability within an existing compound document framework, but the interfaces provided by the component do not conform to the spell checking interfaces required by the framework. This scenario is illustrated in Figure 4.1 on the next page. After studying the interfaces involved, we might decide that it would indeed be possible to use the component with the framework, but that some of the messages exchanged have to be altered while others have to be added.

Component-oriented programming languages therefore have to provide support for adapting and extending existing components retroactively. Given the presence of object-oriented concepts in the standard model (see Section 2.2) mechanisms such as *inheritance* or *delegation* might seem to be good candidates for this.

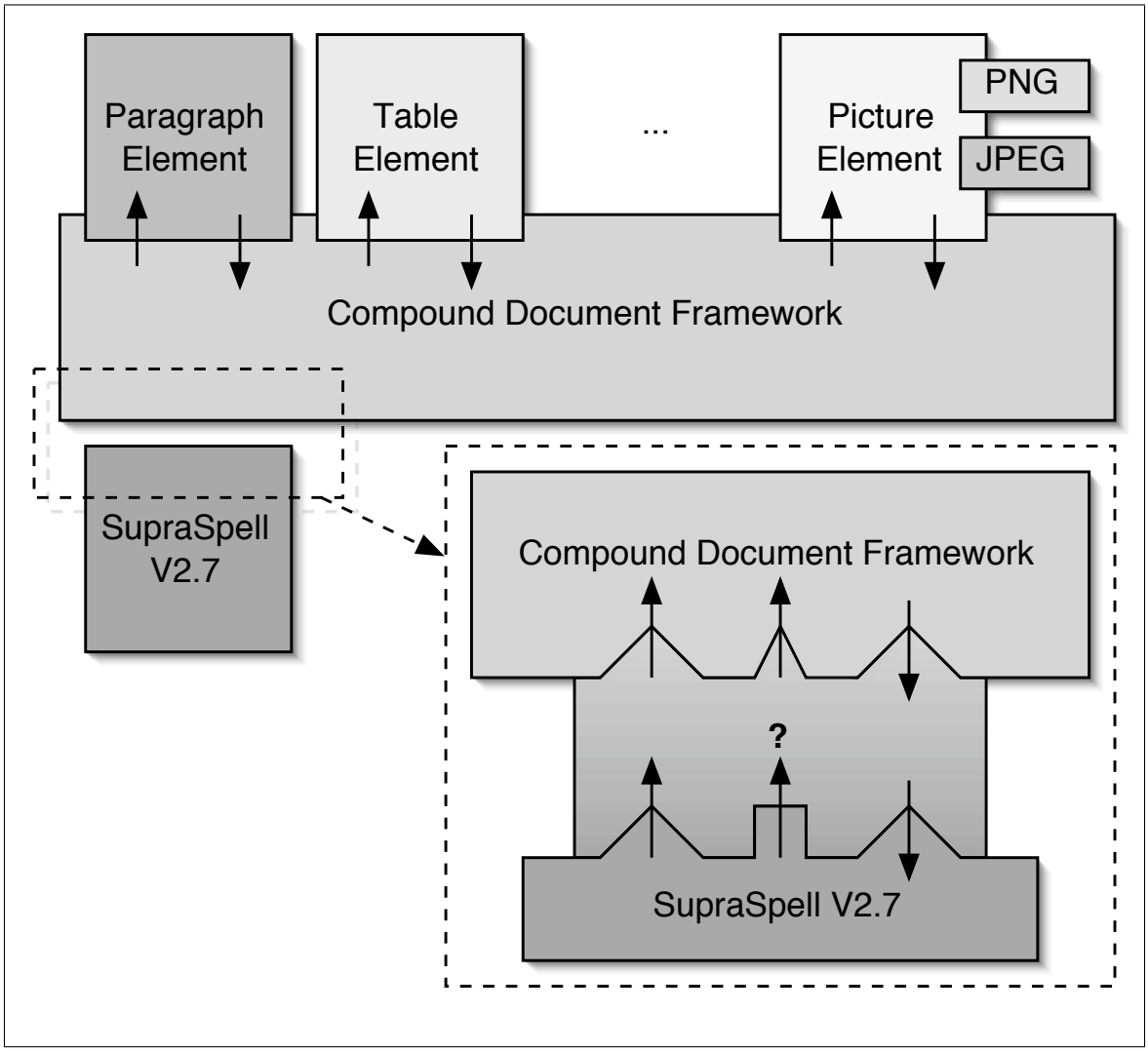


Figure 4.1: Schematic view of a software component that requires adaptation and extension to conform to a compound document framework. Other components are for illustration only.

4.2 The Fragile Base Class Problem

In the following, I once again use a simple example based on the ADT Stack to illustrate the fragile base class problem in detail. As in Chapter 3, code examples are given in $\mathcal{IJ}$, the idealized version of Java outlined in Section 2.2.

Consider a variation of Stack that offers an operation multi_pop to remove $n > 0$ elements at once in addition to the "regular" operations push, pop, top, and empty. Figure 4.2 on the following page gives a possible implementation of this version of the data structure. Note how the `multi_pop` method simply sends `pop` messages to `this` for the required number of times to remove several elements.

Assuming we have a `MultiStack` at our disposal, how can we use objects of this class with the interface shown in Figure 4.3 on the next page? The obvious difference is the message `size` which not provided by `MultiStack`. In order to use `MultiStack` where a `Stack` is expected, we somehow have to add a `size` method to it. But since we consider `MultiStack` to be (part of) a component in this example, we can not simply edit the source code. In $\mathcal{IJ}$, however, we can use *inheritance* to achieve our goal without access to source code, as illustrated in Figure 4.4 on page 55. The `Adapter` class extends `MultiStack` and adds a field `sz` to maintain the current size. It also overrides the methods `push` and `pop` in a way that updates this field whenever the corresponding operations are called. Finally, it adds a method `size` to return the current size of the stack.

4.2.1 Syntactic Aspect

Ignoring problems of instantiation, what we have achieved is exactly what we set out to do, the existing `MultiStack` was adapted to a framework it was not designed for. However, there are still two problems, both having to do with the principle of distributed extensibility again.

Consider what happens when the vendor of `MultiStack` actually adds a `size` operation and (for efficiency reasons maybe) decides to apply the `final` modifier to it (see Figure 4.5 on page 56). Once we install this new version of `MultiStack`,

```
module org.bloat.components {
  class Link {
    Object object; Link next;
  }
  public class MultiStack {
    Link top;
    public void push( Object o ) {
      Link x = new Link(); x.object = o;
      x.next = this.top; this.top = x;
    }
    public void pop() {
      this.top = this.top.next;
    }
    public void multi_pop( int n ) {
      while (n > 0) { this.pop(); n--; }
    }
    public Object top() {
      return this.top.object;
    }
    public boolean empty() {
      return this.top == null;
    }
  }
}
```

Figure 4.2: An $\mathcal{IJ}$ implementation of a stack supporting the `multi_pop` operation to `pop` $n > 0$ elements at once. (Error handling omitted for clarity.)

```
module gov.cia.framework {
  public interface Stack {
    public void push( Object o );
    public void pop();
    public void multi_pop( int n );
    public Object top();
    public int size();
  }
}
```

Figure 4.3: A mismatched stack interface that is not supported by `MultiStack` from Figure 4.2.

```
module net.lagoona.adapters {
  import BS = org.bloat.components, CS = gov.cia.framework;
  public class Adapter
    extends BS.MultiStack implements CS.Stack
  {
    int sz;
    // override push and pop
    public void push( Object o ) {
      super.push( o ); this.sz += 1;
    }
    public void pop() {
      super.pop(); this.sz -= 1;
    }
    // add size
    public int size() {
      return this.sz;
    }
    // multi_pop, top, and empty unchanged
  }
}
```

Figure 4.4: An $\mathcal{IJ}$ adapter that allows `MultiStack` (from Figure 4.2 on the page before) to be used where `Stack` (from Figure 4.3 on the preceding page) is expected. (Error handling omitted for clarity.)

```
module org.bloat.components {
  ...
  public class MultiStack {
    ...
    public void push( Object o ) { ... }
    public void pop() { ... }
    public void multi_pop( int n ) { ... }
    public Object top() { ... }
    public boolean empty() { ... }
    public final int size() { ... }
  }
}
```

Figure 4.5: A vendor change causing the syntactic fragile base class problem (change emphasized).

the `Adapter` class attempts to *override* instead of add the `size` method. Because it is declared `final`, however, this breaks our solution. This is commonly referred to as the *syntactic fragile base class problem*.

By extending a class like `MultiStack` through inheritance, we create a syntactic dependency on its interface, making not only assumption about what is contained in this interface, but also about what is *not* contained in it. Note, however, that this part of the problem could easily be resolved by stand-alone messages as described in Chapter 3 or by one of the more common mechanisms described therein.

4.2.2 Semantic Aspect

A problem that is much more difficult to resolve is the following: Consider what happens when the vendor of `MultiStack` changes `multi_pop` to update the linked list *directly* instead of sending `pop` messages.

As with the use of `final` for `size` above, this change might be motivated by performance considerations. If the new version of `MultiStack` is used through the `Adapter` now, the counter `sz` works fine as long as `multi_pop` is never used.

```
module org.bloat.components {
  ...
  public class MultiStack {
    Link top;
    public void push( Object o ) { ... }
    public void pop() { ... }
    public void multi_pop( int n ) {
      while (n > 0) { this.top = this.top.next; n--; }
    }
    public Object top() { ... }
    public boolean empty() { ... }
  }
}
```

Figure 4.6: A vendor change causing the semantic fragile base class problem (change emphasized, error handling omitted for clarity).

Once it *is* used, however, the counter value will bear *no relation* to the actual state of the stack abstraction anymore. The `Adapter` only overrides `pop` for counting, not `multi_pop`, so if `pop` is not called, the counter is not updated. This is an example for the *semantic fragile base class problem*.

By extending a class like `MultiStack` through inheritance, we create a semantic dependency on the call patterns *within* that class. If those call patterns are not documented, the use of inheritance for component adaptation becomes a guessing game. Even if our adapter works for a particular version of the adapted component, there is no guarantee that it will work for the next version. Note that we can not simply choose to override `multi_pop` as well and hope to solve the problem. If we do, we would not count accurately for the original implementation anymore because we would now update the counter *twice* instead of not at all.

4.3 Rethinking Inheritance and Delegation

The fragile base class problem discussed in Section 4.2 is not specific to $\mathcal{IJ}$. Indeed, I made only very general assumptions about the "ingredients" for component-

oriented programming languages in Section 2.2 where the basics of $\mathcal{IJ}$ were outlined. The following analysis therefore applies to many existing programming languages.

4.3.1 Analysis

Given the problems that inheritance causes in $\mathcal{IJ}$, we could consider variations of the mechanism used in other object-oriented programming languages and hope that they provide a better solution to the problem of component adaptation. I briefly compare three different approaches in this section (see [Tai96] for a more detailed survey):

1. Inheritance based on *overriding* methods of a *super*class.
2. Inheritance based on *augmenting* methods with a *sub*class.
3. *Delegation* of messages from one object to another object.

The first approach is the mechanism in $\mathcal{IJ}$ that we already studied above. It is also used in languages following the tradition of Simula [BDMN73, Mag93], including in Smalltalk [GR83], Eiffel [Mey92], C++ [Str00], Java [GJSB00], Modula-3 [CDG+91], and Oberon-2 [MW91].

The second approach is used in the programming language Beta [MMPN93] and its derivatives. Figure 4.7 on the following page compares these mechanisms in terms of the call patterns that arise. In the case of overriding, calls "start" from the "most derived" class and only reach superclasses when explicitly directed to the super reference. Even self calls in base classes bind to the "most derived" class. In the case of augmentation, calls "start" from the base class and only reach subclasses when explictly directed to the inner reference. Even self calls in subclasses bind the the base class.

In terms of component adaptation, overriding allows us to adapt components in ways that were not foreseen by the original component vendor. However, we pay the price for this as illustrated by the semantic fragile base class problem. Augmentation, on the other hand, only allows us to adapt components in ways that

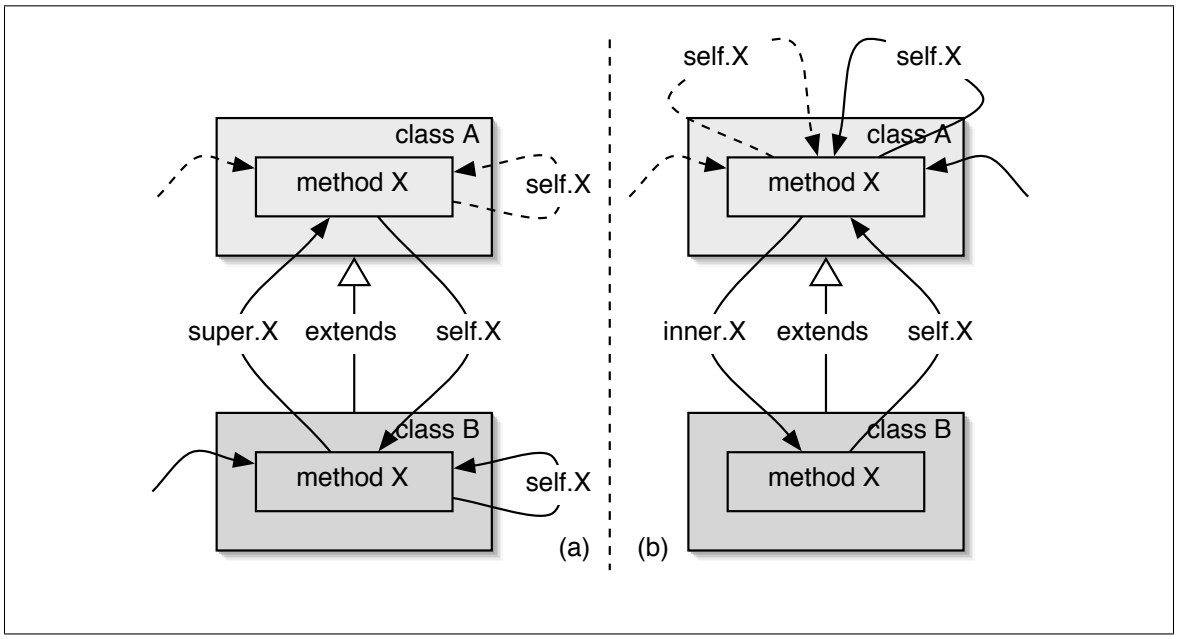


Figure 4.7: The two basic inheritance mechanisms based on overriding as in Java (a) and augmentation as in Beta (b). Delegation as in Self works according to (a) as well. (Arrows represent calls, dashed arrows before inheritance is applied.)

were foreseen by the original component vendor through a "proper" placement of inner calls. However, while providing a strictly weaker mechanism for adaptation, it does not solve the semantic fragile base class problem either.[1]

The third and final approach, delegation [Lie86], is used in the programming language Self [US87]. Instead of being defined on classes, delegation is defined between individual objects. Objects "delegate" messages they do not "understand" to other objects who "answer" on their behalf. While there are a number of advantages to delegation, especially being able to determine "super classes" at runtime, the mechanism is equivalent to overriding inheritance in regard to the call patterns that arise [Ste87]. Delegation therefore suffers from the fragile base class problem as well. In summary, established "alternative" approaches to inheritance do not allow us to properly adapt components for new interfaces either.

[1] Even if we augment all three relevant methods from our example, we still could not tell whether `pop` is used by `multi_pop` or not.

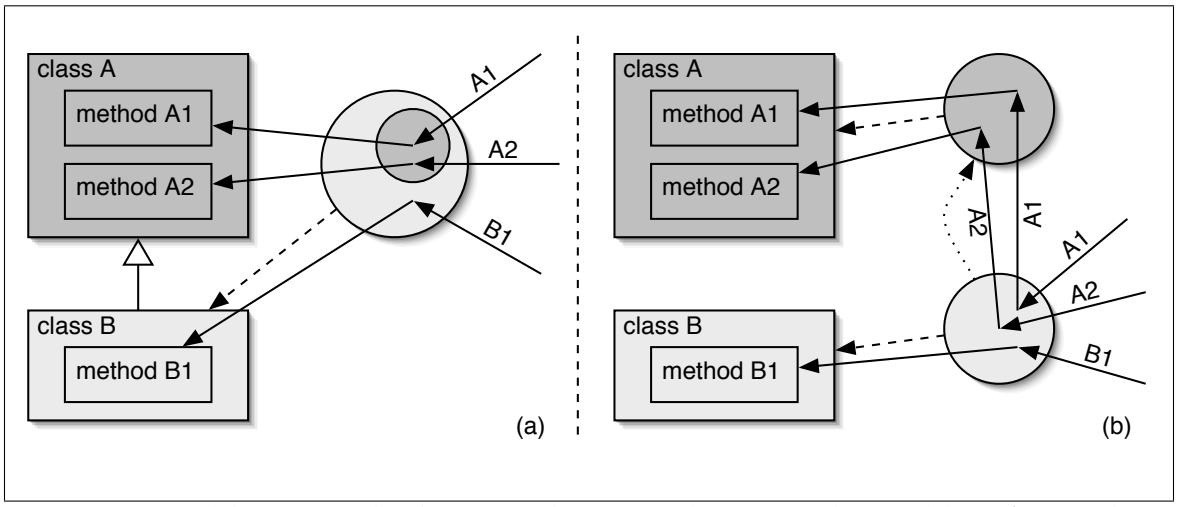


Figure 4.8: Adding a method using inheritance between classes (a) or forwarding between objects (b). (Arrows are "calls," dashed arrows are "instance-of" relationships, dotted arrows are object references).

4.3.2 Synthesis

It should now be obvious that the problem of all mechanisms discussed above is the recursive binding of the self reference. To avoid the fragile base class problem, we have to avoid allowing this reference to change in non-local ways while still allowing us to adapt components as necessary, either by *adding* operations not previously supported or by *overriding* operations to perform slightly different tasks. The idea of delegation actually points the way to a straightforward solution for both of these problems.

Consider the task of adding an operation not previously supported by a component first. In Figure 4.8, we contrast the use of inheritance and *forwarding* in this regard. Instead of a single object in which inheritance and thus recursive binding of self are performed, we maintain two objects of "unrelated" classes. Instead of B extending A, it keeps a reference to an A object which can handle the messages A1 and A2. Since we want to add support for a message B1, class B implements a method for it. However, B also implements methods for A1 and A2, but these methods simply "forward" the messages they receive to the A reference (not shown in Figure 4.8).

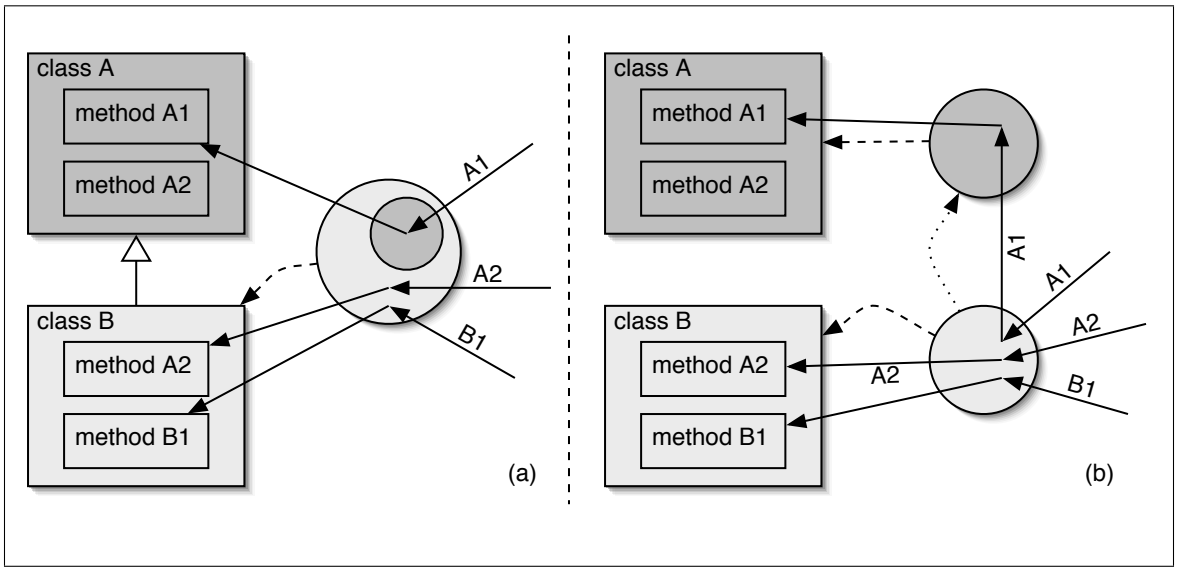


Figure 4.9: Overriding a method using inheritance between classes (a) or forwarding between objects (b). (Arrows are "calls," dashed arrows are "instance-of" relationships, dotted arrows are object references).

Now consider the task of overriding an operation. In Figure 4.9, we again contrast the use of inheritance and forwarding in this regard. To override A2 in B, we simply implement the changes we need instead of forwarding the message to A unchanged. If we do need the equivalent of a super call, we send a message to the A reference explicitly from within B.A2 to achieve this.

Used in this manner, forwarding allows us to perform the necessary component adaption tasks. Since all involved classes remain "self contained" it also avoids the fragile base class problem. However, we now have to write a number of methods that just contain code to send a received message on to another object, a task that will become quite tedious at some point. Using inheritance, those messages that were not overridden in a subclass would "automatically" be sent to the implementation in the superclass.

In other words, we would prefer a mechanism as *convenient* as inheritance in that it allows us to express the forwarding relationships necessary *without* repeating ourselves for each message, while at the same time being *safer* in the sense that it avoids the fragile base class problem. We obviously have to express the forward-

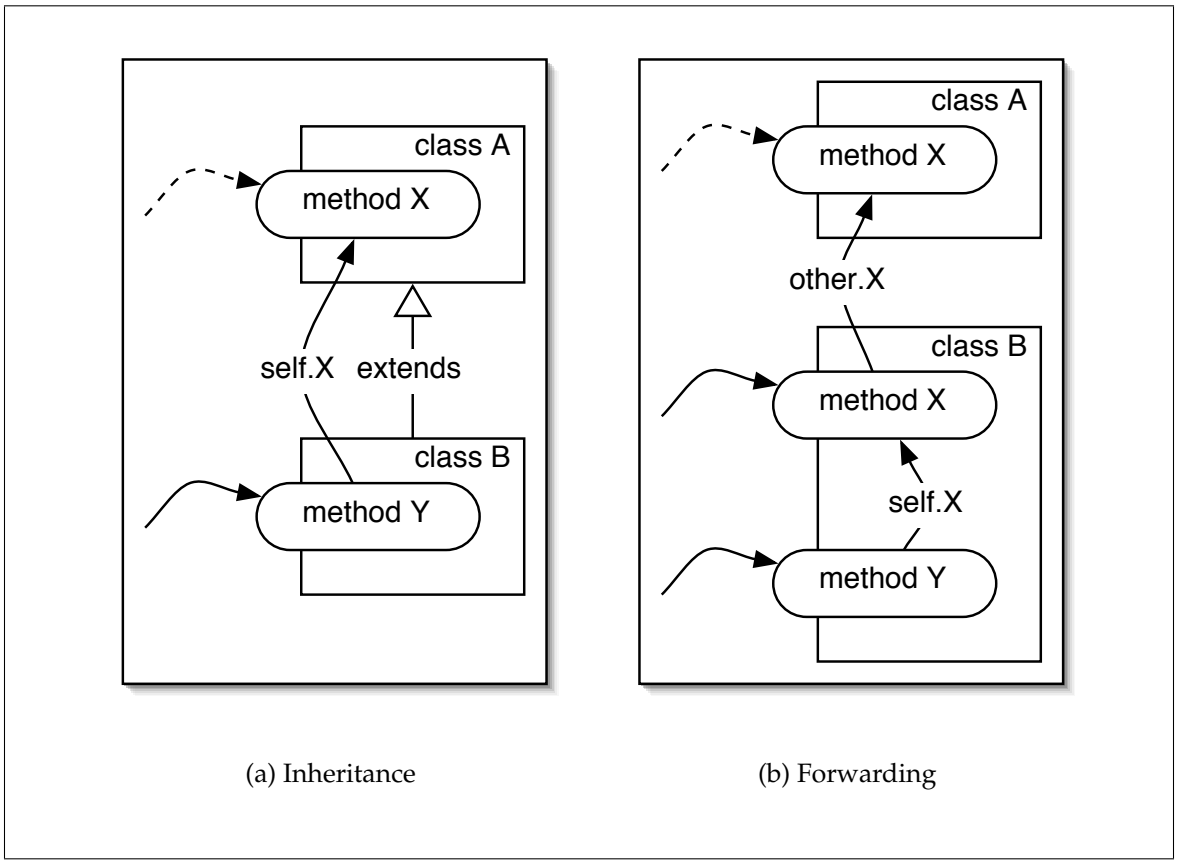
ing relationship at least *once*, however we can take inheritance as an example for how to approach the problem. Instead of listing all the forwarded messages explicitly, we can introduce a single method default that is invoked whenever an object receives a message that it does not handle explicitly. Inside this default method, we can then forward this message to another object. To keep the this mechanism simple and close to the behavior of inheritance, we should leave the identity of the actual message received *opaque* during this process, leading to the name *generic message forwarding* for the resulting mechanism.

4.4 The Expressiveness of Forwarding

Generic message forwarding as introduced above enables us to perform *unforeseen* and *safe* component adaptation in a *convenient* manner. However, since it does not allow for recursive binding of self references, it must clearly be a weaker mechanism than the various forms of inheritance discussed in Section 4.3. While we do not require the full power of inheritance for component-oriented programming, the question of how much expressiveness we lose in dropping inheritance mechanisms is still of interest. In this section, I therefore demonstrate to what extent various forms of inheritance can be "decomposed" into the more basic mechanisms of object composition and message forwarding using several examples. I also study a number of common object-oriented design patterns to verify how common those uses of inheritance that can *not* be decomposed are in practice.

4.4.1 Decomposing Inheritance

The goal for this section is to decompose various forms of inheritance into the more basic mechanisms of object composition and message forwarding. To keep the discussion relatively brief, a number of restrictions that will not change the applicability of the results are imposed. First, we do not discuss the problem of shared state since it can always be expressed in terms of message sends. Furthermore, we ignore the problem of object identity and the related question of transparency to



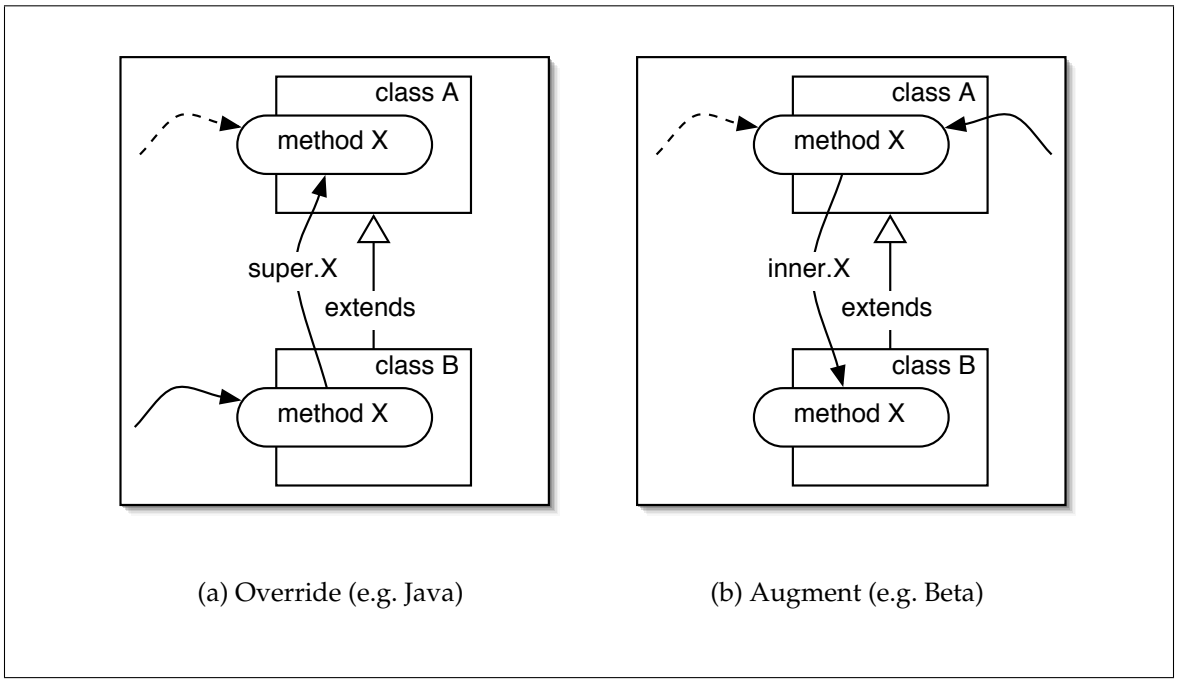
(a) Inheritance (b) Forwarding

Figure 4.10: Call patterns for adding methods using inheritance and fowarding (dashed arrows are "before adding," others are "after adding").

clients. Finally, since our discussion will focus on call patterns exclusively, we can safely disregard issues such as encapsulation, aliasing, and further details about the type or module system.

The "base case" for our discussion of call patterns is a single class A that has neither ancestors nor descendents. Since inheritance is not used at all, we (trivially) do not need to decompose it either.

Next, we consider a class A that defines a method X and a descendent class B that extends A and defines a Y method. This scenario is depicted in Figure 4.10 where we also show the resulting decomposition. On the one hand, since the message Y was not known when the method X was implemented, the call pattern resulting when we send a message X to an instance of B can not involve the Y method. This case can thus easily be decomposed by implementing X in B to for-



(a) Override (e.g. Java)

(b) Augment (e.g. Beta)

Figure 4.11: Call patterns for overriding or augmenting methods through inheritance (dashed arrows are "before inheritance," others are "after inheritance").

ward to an instance of A. On the other hand, if we send a message Y to an instance of B, the method Y *could* send a X message to *self*. However, this case is already handled correctly by the forwarding method X in B. Thus we can accurately decompose inheritance relationships that *add* methods in descendent classes.

For the next two examples, we consider a class A that defines a method Y and a descendent class B which extends A and *also* defines a Y method. This scenario is depicted in Figure 4.11 for two different inheritance mechanisms.

Figure 4.11(a) illustrates the well-known "overriding" semantics of inheritance used in Smalltalk, Eiffel, Java, and most other object-oriented languages. If we send the message Y to an instance of A, the method $A.Y$ will be invoked ("before" in Figure 4.11(a)). However, if we send the same message to an instance of B, the method $B.Y$ will be invoked ("after" in Figure 4.11(a)). Inside the $B.Y$ method, we can invoke $A.Y$ by sending the message Y to *super*. To decompose the "overriding" semantics, the class B has to hold a reference to an A instance, and to emulate the receiver *super* we send messages to that instance.

Figure 4.11(b) on the preceding page illustrates the somewhat obscure "augmentation" semantics of inheritance used in Beta. As before, if we send the message Y to an instance of A, the method $A.Y$ will be invoked ("before" in Figure 4.11(b) on the page before). However, if we send the same message to an instance of B, the method $A.Y$ will *still* be invoked ("after" in Figure 4.11(b) on the preceding page). Inside the $A.Y$ method, we can invoke $B.Y$ by sending the message Y to *inner*.[2] To decompose the "augmentation" semantics, the class A has to hold a reference to a B instance, and to emulate the receiver *inner* we send messages to that instance.

A number of comments are in order at this point. First, note that our decomposition of these two inheritance mechanisms makes their differences more explicit. The "overriding" semantics allow us to redefine methods in ways that the author of the ancestor class did not anticipate. The "augmentation" semantics rely on the author of the ancestor class to provide suitable "hooks" for extension. We can thus view these inheritance mechanisms as corresponding to *wrappers* and *plugins* in the paradigm of component-oriented programming [SGM02]. Next, note that while the two inheritance mechanisms can not easily emulate each other, we can emulate both through composition and forwarding, even concurrently. That is, we can "extend" a class A through "overriding" by wrapping it in an instance of a class B and through "augmenting" by supplying it with a plugin instance of a class C as well. Finally, note that in our model, the concepts of *super*, *inner*, and *self* are very transparent and therefore easy to understand. Sending a message to *self* always invokes a method in the same class, sending a message to *super* always invokes a method in the "closest" ancestor class, and sending a message to *inner* always invokes a method in the "closest" descendent class. This is not the case in inheritance-based models, which will complicate the remaining decomposition.

Consider the example in Figure 4.12 on the next page. The class A initially defines two methods $A.X$ and $A.Y$ and the method $A.Y$ sends the message X to *self*, resulting in the dotted call pattern. The descendent class B extends A and

[2] In Beta, *inner* is actually a keyword and no message sending is involved; the semantics, however, are identical.

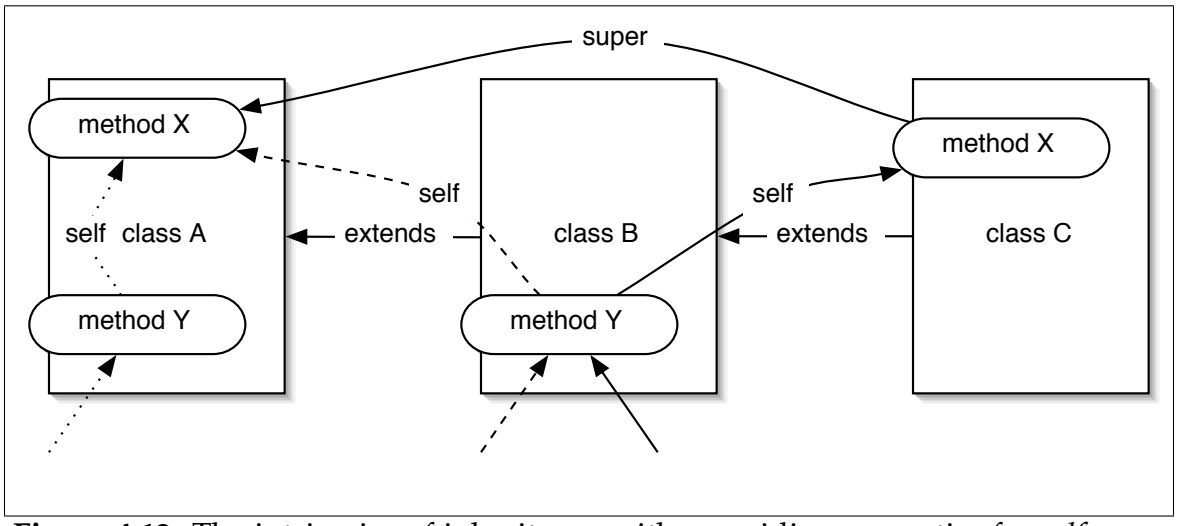


Figure 4.12: The intricacies of inheritance with overriding semantics for *self* message sends.

overrides $A.Y$ with a $B.Y$ method. This method does not call $A.Y$ through *super* but instead sends X to *self*, resulting in the dashed call pattern. We have already seen how to decompose the scenario up to now: we give B a reference to an A instance and defined $B.X$ as a forwarding method. However, we will now add a third class to the mix. The descendent class C extends B (and therefore A as well) and overrides $A.X$ with a $C.X$ method, which also calls $A.X$ through *super*. The resulting call pattern can not be easily decomposed. If we define a forwarding method $C.Y$ the send of X to *self* in $B.Y$ would cause in $C.X$ to be ignored. We cannot make C into a plugin for B either since $C.X$ uses *super* whereas plugins only work for *inner*. Also, this would make B dependent on a plugin even when none is required, i.e. when we only need B and not C instances.

The only viable approach is to pass the "right" *self* as an explicit parameter with each message. To see how this works, we need to "follow the message" through the resulting call pattern. Assume we send Y to an instance c of C with the parameter *self* identifying the (arbitrary) source instance. The $C.Y$ method forwards this message an instance of B and also replaces the existing *self* parameter with c instead. In $B.Y$ we send X to the *self* = c we were passed, but in turn we replace the *self* parameter with b again. Instead of the call through *super*, $C.X$ will send

X to the *self* = b parameter, passing *self* = c once again. The $B.X$ method simply forwards to $A.X$ and we have recreated the call pattern induced by inheritance in this example.

It should be quite obvious that the resulting implementation is very sensitive to changes in the composition. One possible conclusion to draw from this example is that inheritance should not be decomposed at all. However, we believe that the opposite is true. Our decomposition simply sheds some light onto the complexity of call patterns that object-oriented programs sometimes exhibit. Put another way, if a call pattern is complex to decompose, chances are that it is complex to understand as well. This complexity remains hidden from the programmer by "virtue" of the inheritance mechanism, which in turn makes systems exhibiting these call patterns difficult to understand. Finally we get to the question of how common these complex call patterns are in practice, which leads us to the next section.

4.4.2 Design Patterns

It should now be obvious that forwarding can *not* express all the call patterns that can be constructed using inheritance or delegation. However, it should also be obvious that we can not retain either mechanism if we want to ensure distributed extensibility, since both are affected by the fragile base class problem. At this point, an obvious question to ask is how often those call patterns "beyond" forwarding arise in practice.

We could obviously study this question by examining a large number of "real world" systems and measuring frequency of these call patterns. A drawback of such an approach is that we can never be sure whether the results we obtain are "representative" or just artifacts of the particular systems chosen. Therefore, instead of studying actual systems, we fall back on a more "abstract" set of examples, namely design patterns.

Object-oriented design patterns are "elements of reusable object-oriented software" that capture proven solutions to recurring software development problems [GVJH95]. Somewhat unintentionally, design patterns also provide examples for

Pattern	Abstract	Concrete	Notes
Abstract Factory	+	−	
Builder	+	−	
Factory Method	−	+	Replacing "inner" (Section 4.4.1)
Prototype	+	−	
Singleton	−	−	
Adapter	+	+/−	Class Adapter / Object Adapter
Bridge	+	−	
Composite	+	−	
Decorator	+	−	
Facade	−	−	
Flyweight	+	−	
Proxy	+	−	
Chain of Responsib.	+	−	
Command	+	−	
Interpreter	+	−	
Iterator	+	−	
Mediator	+	−	
Memento	−	−	
Observer	+	−	
State	+	−	
Strategy	+	−	
Template Method	−	+	Replacing "inner" (Section 4.4.1)
Visitor	+	−	

Table 4.1: Use of inheritance in design patterns [GVJH95] for interface (fully abstract ancestor) or implementation (partially concrete ancestor) reasons.

typical uses of object-oriented programming languages. Given that something is described as a design pattern, it must have occurred often enough to be identified as such. Thus, if many design patterns utilize a certain language mechanism, we can be reasonably sure that many "real world" software systems use the mechanism as well. Here we are particularly interested in how common design patterns make use of inheritance mechanisms.

In Table 4.1, we list the design patterns from [GVJH95] and classify them regarding their use of inheritance. A "+" in the column "Abstract" means that inheritance from a fully abstract ancestor class is used to establish a common interface

in the sense of subtyping. A "+" in the column "Concrete" means that inheritance from a (partially) concrete ancestor class is used in the sense of subclassing. Somewhat surprisingly, only three out of 23 design patterns critically depend on inheritance for subclassing. Two of these, *Factory Method* and *Template Method*, use inheritance to provide "hooks" that descendent classes are expected to override. As we saw in Section 4.4.1, the resulting call patterns can be decomposed using the "plugin" approach. Only one variation of the *Adapter* pattern resists any attempt at decomposition. A *Class Adapter* uses multiple inheritance for efficiency reasons: it allows adapting an existing class without the need for auxiliary objects. Obviously, we can not decompose this particular use of inheritance while staying true to the intent of the pattern.

Our sample of design patterns illustrates that most uses of inheritance can be decomposed easily. While it would be a fallacy to conclude that because some mechanism is *not* used in design patterns, it is also unused in real systems, we still get the impression that the importance of inheritance might be overrated to some extent.

4.5 Evaluation

To evaluate the concept of generic message forwarding, I compare it to a number of existing approaches for solving the fragile base class problem: component models, programming conventions, design patterns, and generic wrappers. Finally, I summarize my results.

4.5.1 Component Models

In the domain of component models, it is again Microsoft's COM that follows our approach most directly [Mic95]. There is not support for inheritance in COM, for the same reasons pointed out above. Instead, COM relies on forwarding of messages between individual objects, however it does not provide a generic mechanism for this and forwarding has to be performed on a "per message" basis. It

would, however, be straightforward to implement generic message forwarding on top of COM.

4.5.2 Programming Conventions

In their excellent analysis of the fragile base class problem, MIKHAJLOV and Sekerinksi develop an elaborate set of programming conventions to restrict inheritance mechanisms in a suitable way [MS98]. As with stand-alone messages in Chapter 3, the problem with such an approach is that it can not be enforced by the compiler, and thus is not reliable enough for component-oriented programming where we have to rule out the potential for the fragile base class problem to arise.

4.5.3 Design Patterns

The basic idea of forwarding is also at the root of many design patterns [GVJH95], for example the *Adapter* or *Proxy* patterns. As with programming conventions, these patterns *can* avoid the fragile base class problem, but they can *not* be enforced by the compiler. Regarding support for component-oriented programming, design patterns are therefore not reliable enough.

4.5.4 Generic Wrappers

Generic wrappers [BW00] provide an alternative to generic message forwarding that is type safe and allows for most of the component adaptation necessary. However, the mechanism can not be used to support the construction of flexible frameworks, in which generic message forwarding allows extensibility in terms of messages as well as component adaptation.

Unrelated to forwarding, generic wrappers also rely on several programming conventions that we can rule out through stand-alone messages. It seems promising to investigate the integration of generic wrappers with stand-alone messages.

4.5.5 Summary

Generic message forwarding provides a convenient mechanism for component adaptation that avoids the fragile base class problem.

The mechanism is *simpler* to understand than inheritance because it does not lead to recursive binding of `self` and the resulting non-local call patterns. While clearly not as powerful as inheritance in it's various forms, generic message forwarding is able to express a large number of typical uses for inheritance. In particular, it can be used to express all but one out of 23 common object-oriented design patterns examined.

Generic message forwarding is more *flexible* than class-based inheritance since it works along the object graph which can be changed at runtime. In this regard, generic message forwarding is similar to delegation, but again does not suffer from the fragile base class problem.

This flexibility does, however, come at a price in terms of *safety* and *efficiency*. In the presence of generic message forwarding, we can not guarantee complete static type safety anymore since the compiler lacks explicit information about the structure of the object graph and the forwarding relationships that will be imposed on it. I will return to this problem in Chapter 5 and Chapter 7, suggesting various ways in which it can be mitigated. Furthermore, forwarding messages along the object graph obviously requires more work than statically resolving these relationships in the presence of inheritance. Generic message forwarding is thus in the same position as delegation when it comes to performance. As I will discuss in Chapter 6, there are certain situations in which generic message forwarding actually beats the performance of explicitly coded forwarding relationships following design patterns.

Chapter 5

Lagoona

> *. . . 8. A programming language is low level when its programs require attention to the irrelevant. . . . 19. A language that doesn't affect the way you think about programming, is not worth knowing. . . .*
>
> — ALAN J. PERLIS [Per82]

In this chapter, I present the Lagoona design framework for the organizational structure of component-oriented programming languages, which is based on the mechanisms of stand-alone messages and generic message forwarding.

I start with a brief history of the Lagoona project and several remarks on the imperative language core (Section 5.1). Next I discuss the object model that languages following the Lagoona design framework exhibit and illustrate these abstract ideas with a number of simple code examples (Section 5.2). This leads into a discussion of various applications—technical as well as non-technical—of the object model, including novel solutions to several important design and implementation problems in component-oriented programming (Section 5.3). Finally, I evaluate Lagoona by comparing it to a number of existing proposals for component-oriented programming languages and related language mechanisms (Section 5.4).

5.1 Overview

The design framework I present below was developed as part of the Lagoona project which investigates programming language support for the paradigm of component-oriented programming. Besides this focus, however, the project is also concerned with language design and implementation "for its own sake," and several ideas unrelated to component-oriented programming have been explored. The "art of simplicity" as practiced by WIRTH has been an important guideline throughout the project and led to tradeoffs that might be a little surprising in this day and age [BGP00].

5.1.1 Historical Remarks

A complete account of the programming language developments that eventually lead to Lagoona would have to start with Algol 60 [Nau63, RR64], but such an account would hardly qualify as a "remark" anymore. I will therefore start with Oberon [RW92], which could be described as a "minimalist's" object-oriented programming language. Oberon was designed in the 1980s by WIRTH in the tradition and spirit of Pascal [JW91] and Modula-2 [Wir89], his previous and more well-known designs. Oberon dropped many of the mechanisms that were rarely used in Modula-2 with the goal of making the language truly minimal and simple. Only a few mechanisms were added, most importantly type-extension between record types, the basis for object-oriented programming.

Oberon retained Modula's module concept and could thus be described as the earliest language following the "standard model" for a component-oriented programming language (see Section 2.2). More importantly, however, the Oberon System [WG92] already contained many of the ideas that would lead to component-oriented programming as later formulated by SZYPERSKI and others [SGM02]. One of these ideas, namely the use of *message objects* to implement an extensible architecture for Oberon's user interface, eventually gave rise to the notion of stand-

alone messages.[1] The first language construct for messages, albeit still far from their current form, appeared in Object Oberon [MTG89], an experimental extension of the Oberon language that added "better" support for object-oriented programming. Curiously, the message construct is absent from Oberon-2 [MW91], which in turn developed out of Object Oberon. The next language construct for messages appeared in the "protocol extension" proposal for Oberon by FRANZ [Fra95]. At this point, the notion becomes first recognizable as the current concept, although the emphasis of the proposal is not on messages but rather on a form of "modular mixin inheritance" that allows new methods to be added to classes retroactively. In the first Lag*oo*na proposal, messages finally appear in pretty much their current form, although embedded in quite a different object model [Fra96, Fra97b]. The same is true for concept of generic message forwarding, which also has been refined further into the form described in this dissertation.

For the record, the central differences between the original Lag*oo*na proposal ("Lag*oo*na 97") and the object model of Lag*oo*na described here are the introduction of two message send operators leading to improved type safety, the introduction of structural conformance between interface types, and the removal of type-extension between implementation types.

5.1.2 Core Language

In spite of the Java-based surface syntax I have used throughout this dissertation, Lag*oo*na's imperative core language consists of a simplified version of Oberon. However, a number of genuine Java influences are present as well, for example the rule that instances of objects are always treated using reference semantics.

The core language is designed to be as simple as possible. It supports `int`, `float`, `boolean`, and `char` as basic data types, as well as type constructors for arrays and records (classes). Apart from assignment commands, the core supports the usual control structures such as `if`, a safe form of `switch` without `break`,

[1] Oberon's message objects would be classified as an application of the *Command* design pattern today [GVJH95].

`while`, `repeat`, and a bounded form of `for`. Sequences of commands or arithmetic expressions can be abstracted using a standard `procedure` mechanism, with parameter passing following the Ada [Int95] model of explicit `in`, `out`, and `inout` parameter modes. This allows describing the intended data-flow across a parameter explicitly, but without committing to a certain implementation of parameter passing.

As pointed out in Section 2.2, the computational core language is not important for the organizational structure and could just as well be in the form of a functional or logical languages. However, our experience with Lag*oo*na implementations is so far limited to imperative core languages, and I wanted to document the nature of the core I assume in the following for reference.

I would also like to point out that the core language has been explicitly designed to facilitate simple yet efficient code generation. The control flow structure is limited and enables the generation of advanced intermediate representations (such as SSA form) as well as common code generation tasks (such as register allocation) to be performed in straightforward ways [BM94, Tho98]. There is, for example, no `return` command that can be used to leave procedures at arbitrary points thus complicating the control flow.

5.2 Object Model

The object model at the core of the Lag*oo*na design framework separates many of the roles traditionally played by classes in object-oriented programming languages, turning them into individual language constructs. Table 5.1 on the next page provides a compact comparison of how different design concerns are mapped onto language constructs in traditional object-oriented languages and in Lag*oo*na.

At the lowest level of Lag*oo*na's object model are *messages* and *methods*. Messages are *abstract operations* that describe *what* effect they achieve, while methods are *concrete operations* that describe *how* a certain effect is achieved. In other words, messages are specifications for methods, and methods are implementa-

Concern	Traditional	Lagoona
Encapsulation	Class (modifiers)	Module
Specification	Class (abstract method) Class (abstract)	Message Interface Type
Implementation	Class (concrete method) Class (concrete)	Method Implementation Type
Modification	Class (inheritance)	Forwarding

Table 5.1: Design concerns and corresponding language constructs in traditional object-oriented languages and in Lagoona.

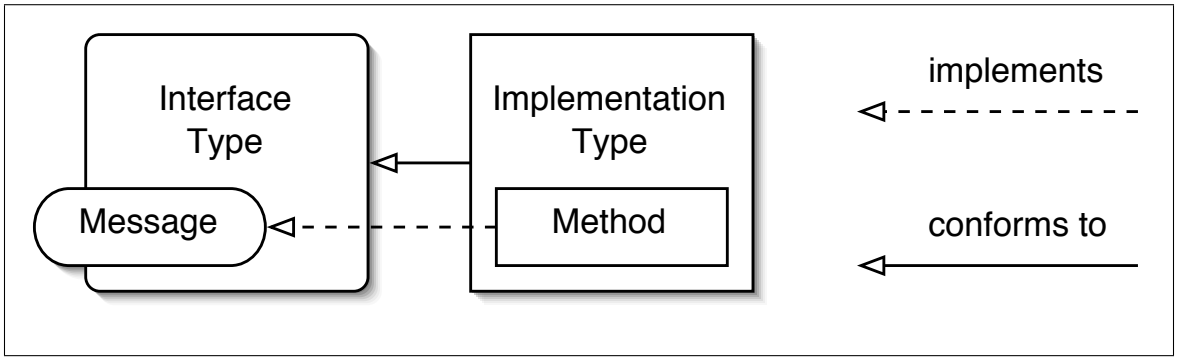


Figure 5.1: Notation for messages and interface types that include them, as well as for methods and implementation types to which they are bound.

tions of messages. At the next higher level, messages and methods are grouped into *interface types* and *implementation types*. An interface type is simply a set of messages, while an implementation type consists of a set of methods and associated storage definitions. Variables of these types are called *interface references* and *implementation references* respectively. Implementation types serve as generators for *instances*, which are first-class values that can be assigned to implementation or interface references. As with messages and methods, interface types and implementation types serve as specifications and implementations for each other. We use the notation shown in Figure 5.1 to express these relationships graphically (the notion of conformance is defined in more detail below). At the highest level of the object model are *modules* which encapsulate sets of messages, methods, interface types, and implementation types. Modules are *unique* in the sense that only a sin-

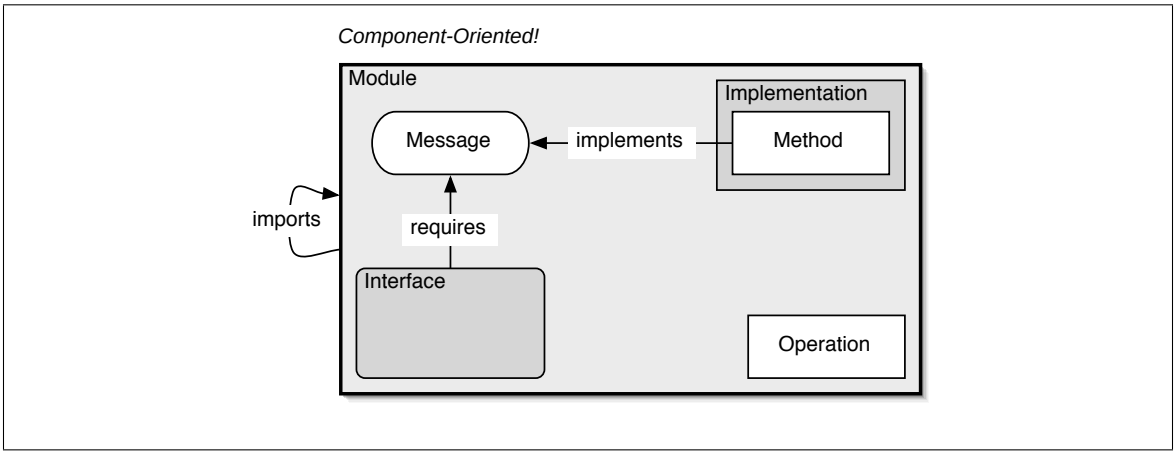


Figure 5.2: Lag*oo*na's model for component-oriented programming languages illustrated in the style of Figure 2.5 on page 19 and Figure 2.6 on page 25.

gle copy of a certain module can exist in a given system. Figure 5.2 summarizes the design framework graphically.

So far, this description of Lag*oo*na's object model reads almost like the textbook definition of any object-oriented programming language. What sets the Lag*oo*na framework apart are the following additional relations between the concepts introduced above. Although messages are "grouped into" interface types, they are not declared in the scope of a type but rather in the scope of a module. Since modules are unique, this implies that messages are unique as well. This is the concept of *stand-alone messages* introduced in Chapter 3. In contrast to messages, methods *are* declared in the scope of an implementation type. This asymmetry is intentional, since we want to support multiple implementations of identical specifications on the level of messages and methods as well as on the level of interface types and implementation types. To relate interface types and implementation types (including their instances), we need to define some notion of *conformance*:

1. An interface type B denoting a set of messages M_B conforms to an interface type A denoting a set of messages M_A if and only if M_B is a superset of M_A:

$$\text{IntIntConf}\frac{\Gamma \vdash A = M_A \qquad B = M_B \qquad M_A \subseteq M_B}{\Gamma \vdash A \leq B} \tag{5.1}$$

   In other words, we employ *structural conformance* or *structural subtyping* between interface types.

2. An implementation type C with a set of methods implementing a set of messages M_C conforms to an interface type B denoting a set of messages M_B if and only if M_C is a superset of M_B:

$$\text{IntImpConf}\frac{\Gamma \vdash B = M_B \qquad C = M_C \qquad M_B \subseteq M_C}{\Gamma \vdash B \leq C} \tag{5.2}$$

   We extend structural conformance to implementation types, and if (5.1) and (5.2) hold, $A \leq C$ will hold as well. Furthermore, this enables a form of *inclusion polymorphism* that we like to call *implementation polymorphism*.

3. An interface type never conforms to an implementation type. Of course, Lagoona allows interface types to be *cast* to implementation types, guarded by a dynamic check.

4. Two implementation types only conform if they are the same type. In other words, we employ *occurrence equivalence* between implementation types.

This completes the definition of conformance, but the fourth case raises the question of how implementation types can be reused or adapted.

At runtime, Lagoona's object model essentially reduces to a web of independent instances that communicate through messages. Assume we are sending a message m to a receiver r, which can be an interface or an implementation reference, whose type R denotes a message set M_R. We distinguish two *message send operators* with different semantics:

1. The first operator $\rightarrow$ is *strict* in the sense that the expression $m \rightarrow r$ is valid if and only if m is an element of M_R:

$$\text{StrictSend}\frac{\Gamma \vdash r : R \qquad R = M_R \qquad m \in M_R}{\Gamma \vdash m \rightarrow r} \tag{5.3}$$

   In other words, this operator statically ensures that the message m will be "handled" by the instance bound to r.

2. The second operator $\Rightarrow$ is *blind* in the sense that the expression $m \Rightarrow r$ is *always* valid.

$$\text{BlindSend}\frac{\Gamma \vdash m \in M \qquad r : R}{\Gamma \vdash m \Rightarrow r} \tag{5.4}$$

Of course, we have to guard the application of this operator by a dynamic check, similar to the one for casts mentioned above.[2]

The blind message send operator is necessary to support reuse and adaptation by intercepting and rerouting messages. Implementation types can define a *default method* which is triggered for messages that do not have an explicit method associated with them. Inside this default method, messages can be *resent* or *forwarded* to other instances. This is the concept of *generic message forwarding* introduced in Chapter 4. The actual message remains opaque during this process. Obviously, the strict message send operator alone would not be sufficient to support this.

Lagoona's object model can be viewed as another step towards eliminating the dominance of the class construct in object-oriented languages. Previous steps include the separation of interfaces and implementations [Sny86] and the separation of modules and types [Szy92], both of which are widely accepted by now. In the remainder of this section I explain each element of Lagoona's object model in more detail. I also discuss how these elements are mapped into the actual programming language using several concrete examples.

5.2.1 Modules

Lagoona's top-level language construct is the *module*, which serves a variety of purposes. Modules are compilation units and result in object files which in turn are the units of deployment [SGM02]. Modules live in a flat, global namespace and cannot be nested. However, we employ a "hierarchical" naming convention based on Internet domain names, similar to the one originally proposed for Java packages [GJSB00]. Modules are *sealed* in the sense of CARDELLI [Car89]; only explicitly exported declarations are visible to clients, and no new declarations can be added from the outside. Modules can *import* other modules and then refer to their exported declarations. These references are fully qualified, but to avoid "excessive" qualifications we allow the introduction of local *aliases* for imported modules.

[2] For sensible assignment semantics, it is also necessary to restrict $\Rightarrow$ to messages that do not return a result.

```
module com.lagoona.thesis.stacks {
  // pre obj != null; post top() == o;
  public message void push( any obj );
  // pre !empty();
  public message void pop();
  // pre !empty(); post return != null;
  public message any top();
  // "no elements?"
  public message boolean empty();
  public interface Stack { push, pop, top, empty }
}
```

Figure 5.3: The stack abstraction in Lag*oo*na. Messages are bound to modules, not types.

The module shown in Figure 5.3 exports all its declarations by marking them `public`. The module in Figure 5.5 on page 82 imports the first one under the alias `S` and uses this alias to qualify further references, for example to the message `push`. However, several declarations inside the second module are not marked `public` and are therefore hidden from its clients.

5.2.2 Messages

One feature that sets Lag*oo*na apart from established object-oriented programming languages is *stand-alone messages*. As shown in Figure 5.3, messages are bound to (declared in) modules instead of types. Since modules are unique within a given system, and since no two messages can have the same name within a given module, our approach makes messages unique as well. If messages were bound to types, the approach taken in most conventional object-oriented languages, we could not guarantee this property in general. Surprisingly, many of the applications described in Section 5.3 stem from this seemingly trivial difference.

We usually associate a semi-formal specification with each message, in terms of preconditions, postconditions, and invariants. The `push` message, for example, would be characterized with the precondition "obj $\neq$ null" and the postcondition

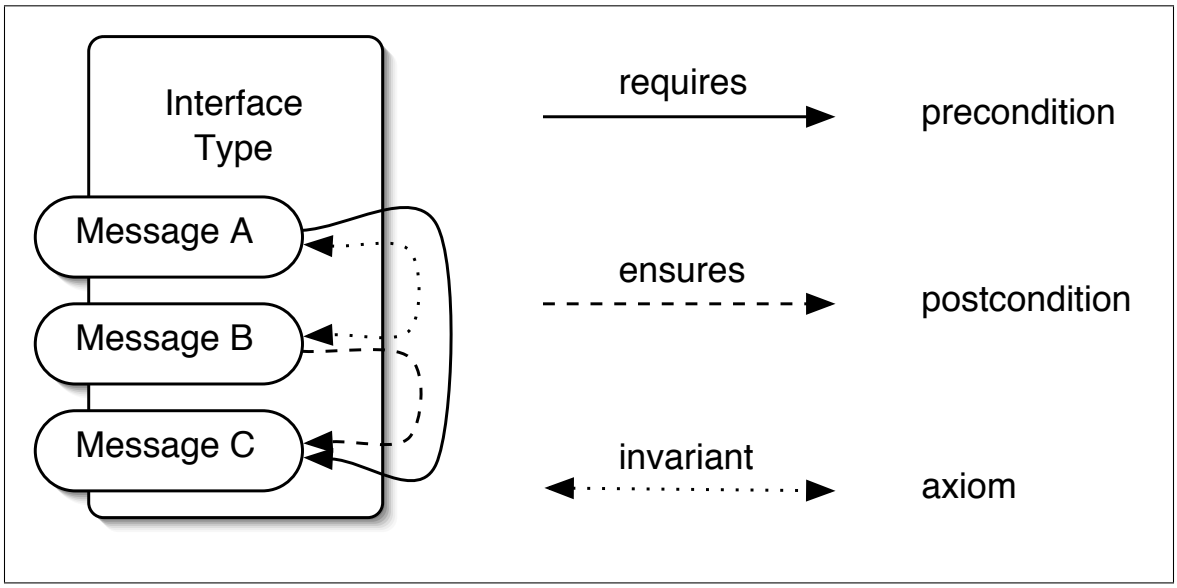


Figure 5.4: Notation for messages and their dependencies on other messages in terms of precondition, postconditions, and axioms.

"¬empty". We use the notation shown in Figure 5.4 to express these relationships between messages graphically. Finally, we assume that a message and it's specification are *immutable* once published, which is similar to the assumption made about interfaces in COM [Mic95] and related technologies.

5.2.3 Interface Types

Messages are the basis for *interface types* (`interface` in our concrete syntax) which represent references to objects that implement a certain set of messages. In Figure 5.3 on the preceding page, the interface type `Stack` is declared as supporting the messages `push`, `pop`, `top`, and `empty`. If we declare a variable `s` of type `Stack`, we can only assign objects that implement at least these four operations to `s`. As explained in Section 5.2, conformance to interface types is structural. The pervasive interface type `any` represents the empty message set and is the top element in the resulting type lattice. Note that the name we give to an interface type is only a convenient abbreviation; instead of using such a name, we could also declare isomorphic interface types repeatedly. Conceptually, interface types

```
module com.lagoona.thesis.simple_stacks {
  import S = com.lagoona.thesis.stacks;
  class Link {
    any object; Link next;
  }
  public class Stack {
    Link top;
    method void initialize() {
      this.top = null;
    }
    method void S.push( any obj ) {
      Link x = new Link(); x.object = obj;
      x.next = this.top; this.top = x;
    }
    method void S.pop() {
      this.top = this.top.next;
    }
    method any S.top() {
      return this.top.object;
    }
    method boolean S.empty() {
      return this.top == null;
    }
  }
}
```

Figure 5.5: An implementation of the stack abstraction. Methods implementing messages are bound to types.

in Lag*oo*na are used to decouple independent components, similar to the use of interfaces in both COM [Mic95] and to a certain extent Java [GJSB00].

5.2.4 Implementation Types

Implementation types (`class` in our concrete syntax) host methods and declarations for instance variables. Consider the implementation of the `Stack` abstraction shown in Figure 5.5 on the page before. Each method implements exactly one message imported from the module `S`. The message `initialize` (and also `finalize`) has a special meaning in Lag*oo*na: it is sent by the runtime system immediately after an instance has been created (or, in the case of `finalize`, right before it is garbage collected). The class `Link` is essentially used as a simple record type without any methods.

Figure 5.6 on the following page illustrates how message forwarding between instances is used to "extend" an existing implementation type. In this example, we want to extend the stack abstraction (and it's implementation) with an operation that determines the number of elements currently on the stack. First we introduce a new message `elements` which does exactly that. Next we declare a class `Stack` that has an interface reference to another stack and an instance variable for the actual counter. The method `elements` simply returns the counter value. The methods `S.push` and `S.pop` update the counter and forward their messages to the "basic" stack instance.

Although not directly related to the extension we want to produce, we also have to implement the messages `S.top` and `S.empty`. The reason is that both of these messages return a value and can therefore not be handled by the generic message forwarding mechanism implemented in the `default` method. However, implementing the `default` method as shown allows this extension to be composed with other, unrelated extensions.

```
module com.lagoona.thesis.counting_stacks {
  import S = com.lagoona.papers.thesis.stacks;
  public message int elements();
  public class Stack {
    S.Stack stack;
    int count;
    method void initialize( S.Stack stack ) {
      this.stack = stack; this.count = 0;
    }
    method int elements() {
      return this.count;
    }
    method void S.push( any obj ) {
      this.count += 1; S.push( obj ) -> this.stack;
    }
    method void S.pop() {
      this.count -= 1; S.pop() -> this.stack;
    }
    method any S.top() {
      return S.top() -> this.stack;
    }
    method boolean S.empty() {
      return this.count == 0;
    }
    method void default() {
      current => this.stack;
    }
  }
}
```

Figure 5.6: Adding counting to the stack abstraction and its implementation.

5.3 Applications

In this section, we illustrate how stand-alone messages and generic message forwarding address a number of recurring design and implementation problems in both object-oriented programming and component-oriented programming.

5.3.1 Structural Interface Conformance

Conformance of an implementation type A to an interface type B can either be *declared* explicitly as in Java [GJSB00], or it can be *inferred* based on a *structural* property as in the Lagoona design framework. Structural conformance has a number of advantages, especially for software evolution [LBR98]. More importantly, however, a certain degree of structural conformance is *required* for component-oriented programming [BW98].

Consider two interface types A and B that have been defined independently by vendors A and B. Vendors C and D define—again independently—combinations of A and B, for example C = A + B and D = B + A. While both C and D support *exactly* the same messages, they do *not* necessarily conform to each other. Most object-oriented languages rely on a declared form of conformance, i.e. types are equivalent *by name* (or *by occurrence*) instead of *by structure* (or *by extent*).

The usual objection to structural conformance is that it is "weaker" than declared conformance because it can result in "accidental" conformance relations that the programmer did not anticipate. The archetypal example of this problem is an interface type `Cowboy` that includes a message `draw` and an interface type `Shape` that also includes a `draw` message, presumably with different semantics. In a language that supports stand-alone messages, accidental conformance of this kind is *not* possible. The two `draw` messages would have to be defined in different modules and would therefore be distinguishable.

The use of structural conformance has been proposed before. In Modula-3 [CDG+91] structural conformance is used by default, but reference types can be *branded* to avoid accidental conformance. However, all brands in a composed sys-

```
...
import f = edu.uci.framework;
...
// does not modify "s"
void printTop( f.Stack s ) {
  if (!f.empty() -> s) {
    print( f.top() -> s );
  }
}
...
```

Figure 5.7: Example method to illustrate minimal typing.

tem (a "program" in Modula-3) must be unique, which can restrict independent extensibility by mutually unaware vendors. The *compound types* proposal for Java [BW98] uses declared conformance for individual interfaces and structural conformance for combined interfaces. Although backward compatible with Java, compound types add additional rules to an already complex language and do not address the problem of interface conflicts at all. Another proposal for Java [LBR98] requires that interfaces for which structural conformance should be used must *extend* an explicit marker interface `Structural`.

In contrast to these approaches, structural conformance in Lagoona does not require any additional language constructs to avoid accidental conformance. Furthermore, our design is more flexible since any arbitrary combination of messages can be "promoted" to an interface type which can subsequently be used with the "correct" conformance relationships.

5.3.2 Minimal Typing

An interesting application of structural conformance is that signatures of messages can be typed in a "minimal" way to express certain invariants. Consider a method that prints the `top` element of a `Stack` as shown in Figure 5.7. In this example, the fact that `printTop` does not modify the `Stack` is only stated as a comment. Clients are therefore unable to rely on this information with the same confidence

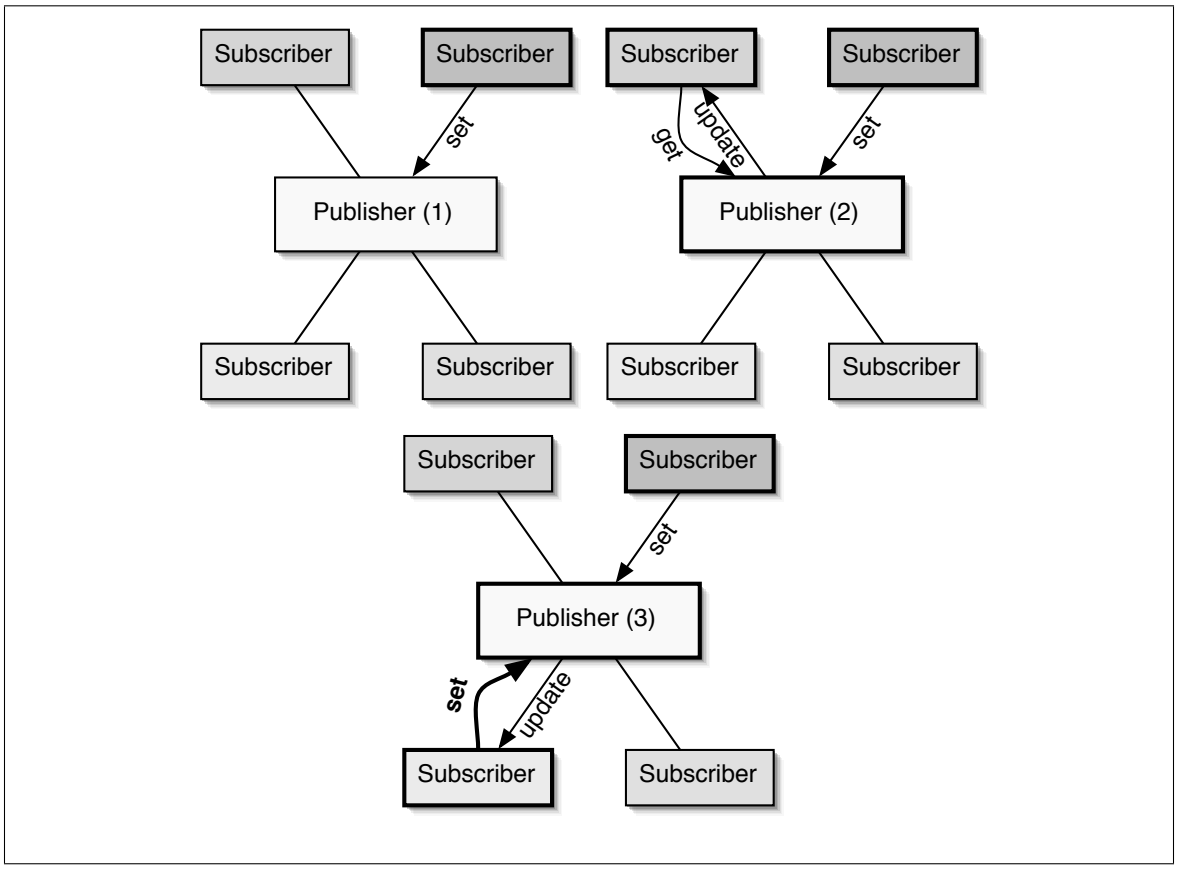


Figure 5.8: Publishers and subscribers as an example for the component reentrance problem.

that they rely on annotations that are expressed in the type system directly. In Lag*oo*na, we can express this fact succinctly and safely by using an anonymous interface type to define the signature of `printTop` as follows:

```
void printTop ( interface {f.empty,f.top} s )
```

Given this signature, only the `empty` and `top` messages could be sent to `s`, ensuring that `printTop` does indeed not modify the stack. Obviously this holds as long as `printTop` does not *cast* the parameter to another type that exposes more messages.

```
package org.bloat.pubsub;
public interface Publisher {
  public void attach( Subscriber me );
  public void detach( Subscriber me );
  public Object get();
  public void set( Object data );
}
public interface Subscriber {
  public void update( Publisher from );
}
```

Figure 5.9: Naive publishers and subscribers in Java.

5.3.3 Component Reentrance

When we use messages and interface types to specify the functionality of certain instances, we often make the assumption that each operation executes *atomically*. However, for certain design patterns that rely on "callbacks" between instances this is not the case, leading to the *component reentrance* problem [MSL99, SGM02].

Consider the *Observer* (or *Publish-Subscribe*) design pattern [GVJH95] for example, which is used to achieve loose coupling between objects by implicit invocation. A publisher encapsulates some kind of data that is of interest to subscribers. When this data changes, the publisher automatically notifies all its current subscribers. Figure 5.9 illustrates how this design pattern could be modeled in Java using two interfaces `Publisher` and `Subscriber`. Subscribers `attach` themselves to a publisher, and whenever `set` is invoked, the publisher in turn invokes `update` on all registered subscribers. Subscribers then use `get` to retrieve the current state of the publisher and update themselves accordingly.

While this sounds great, there are in fact several problems. For example, consider subscribers that send `attach` or `detach` to the publisher from *within* their `update` method. Since the publisher is currently traversing some kind of data structure to `update` all subscribers, the effect of these operations becomes highly dependent on the implementation of this traversal. Even worse, as shown in Figure 5.8 on the page before, subscribers might send `set` within their `update`

```
module com.lagoona.pubsub {
  public interface Publisher { attach, detach, get, set }
  public message void attach( Subscriber me );
  public message void detach( Subscriber me );
  public message any get();
  public message void set( any data );
  public interface Subscriber { update }
  public message void update( interface { get } from );
}
```

Figure 5.10: Smarter publishers and subscribers in Lagoona. As shown here, only `get` can be sent within `update`.

method, resulting in infinite recursion.

The component reentrance problem can be solved by implementing publishers very defensively, e.g. by cloning the data structure before traversal and by protecting the `set` method using some kind of flag. However, the problem really boils down to what messages can be sent to the publisher from within the `update` method. If we restrict this set of messages, we can *statically* ensure that the reentrance problem does not occur.

Figure 5.10 shows how we would model the design pattern in Lagoona. Instead of typing the `from` parameter of `update` with `Publisher`, we introduce an anonymous interface type that only supports the `get` message. While subscribers can still send other messages if they have another reference to the publisher, or if they cast the `from` parameter accordingly, our description of the design pattern is still more accurate and elegant. To achieve the same result in Java, we would have to introduce an artificial base type, e.g. `PublisherJustGet`, that we derive `Publisher` from.

5.3.4 Iterators

Certain programming languages, CLU [LAB+79] and Sather [MOSS96] for example, offer an *iterator* construct to traverse encapsulated data structures in a modular manner. In most object-oriented programming languages, iterators are "emulated"

```
import java.util.Enumeration;
...
class JavaIterator {
  ...
  public void action() {
    Enumeration e = container.elements();
    while ( e.hasMoreElements() ) {
       Object obj = e.nextElement();
       System.out.println( obj );
       ...
    }
  }
}
```

Figure 5.11: Using iterators in Java.

at the library level [GVJH95, SL94]. Using Lag*oo*na's mechanism for generic message forwarding, we can implement iterators that are as powerful as library approaches, but often as convenient to use as language approaches.

Figure 5.11 shows how iterators are commonly used in Java and similar languages that follow the library approach. The container class provides a method, in this case `elements`, that returns an iterator object. This object encapsulates the necessary operations to perform the actual traversal, and offers an interface to check for more elements or to advance to the next element. However, the iteration loop itself must be implemented manually each time an iteration is required.

In Lag*oo*na, we can offer a more elegant solution that avoids this tedious repetition. Since the `default` method enables us to specify a strategy for forwarding messages in the imperative core language, we are by no means limited to just a single receiver. Instead, we can implement a generic *broadcast* mechanism for messages. Figure 5.12 on page 92 shows how we can use this idea to implement iterators in Lag*oo*na. The container `Array` implements a message `forward`, which returns an iterator instance. The iterator contains a reference to the elements to be traversed and fully encapsulates the iteration strategy. For this example, we have limited ourselves to forward iteration, but a `backward` message could easily be

added, returning an iterator instance for backward iteration. The actual iteration is performed by simply sending a message to the iterator instance. The iterator itself does not implement any message but instead broadcasts all received messages to the elements in the container. The actual action to be performed on each element is located in a method of the container elements. Message parameters can be used to pass additional context information from the current control flow to this iterator method. This approach to iterators offers a much cleaner separation between the iteration code and the application code then traditional iterator schemes. All code related to the iteration is located in the module containing the container and its iterator functionality.

5.3.5 Design Guidelines

Stand-alone messages are also helpful as design guidelines during development. For example, consider designing an interface for bounded stacks based on the interface `edu.uci.framework.Stack` for unbounded stacks. The existing interface provides the messages `push`, `pop`, `top`, and `empty`. The only message not yet provided is `full` which indicates that no more elements can be `pushed`. This reasoning leads to the interface given in Figure 5.13 on page 93.

However, this interface does not capture the intended semantics accurately. Consider the precondition associated with the `push` message in Section 3.2. It states that `push` only fails if we pass **`null`** as a parameter, but for a bounded stack `push` should also fail if the stack is `full`. This insight leads to the interface shown in Figure 5.14 on page 93.

Focusing on messages and their semantics, for example using a graphical notation as shown in Figure 5.15 on page 94, thus helped us to uncover an inconsistency between the interfaces for bounded and unbounded stacks. While developers can not be forced to design semantically consistent interfaces, we believe that concentrating on messages facilitates this process.

Note how introducing a new `push` message enables us to express the semantic difference between bounded and unbounded stacks. The interfaces for bounded

```
module com.lagoona.thesis.iterator {
  ...
  class ArrayForwardIterator {
    any[] data;
    method void default() {
      int j = 0;
      while ( j < this.data.length ) {
        current => this.data[j++];
      }
    }
  }
  class Array {
    any[] data;
    ...
    method ArrayForwardIterator forward() {
      ArrayForwardIterator i =
        new ArrayForwardIterator();
      i.data = this.data;
      return i;
    }
  }
  class LagoonaIterator {
    Array array;
    ...
    method void action() {
      array.forward().print();
    }
  }
}
```

Figure 5.12: Implementing iterators in Lag*oo*na by leveraging generic message forwarding for broadcasting.

```
module edu.uci.framework.bounded {
  import f = edu.uci.framework;
  // "no more pushes?"
  public message boolean full();
  public interface Stack {
    full, f.push, f.pop, f.top, f.empty
  }
}
```

Figure 5.13: A semantically flawed interface for bounded stacks.

```
module edu.uci.framework.bounded {
  import f = edu.uci.framework;
  // pre !full() && o != null; post f.top() == o
  public message void push( Object o );
  // "no more pushes?"
  public message boolean full();
  public interface Stack {
    push, full, f.pop, f.top, f.empty
  }
}
```

Figure 5.14: An semantically sound interface for bounded stacks modeling behavioral subtyping.

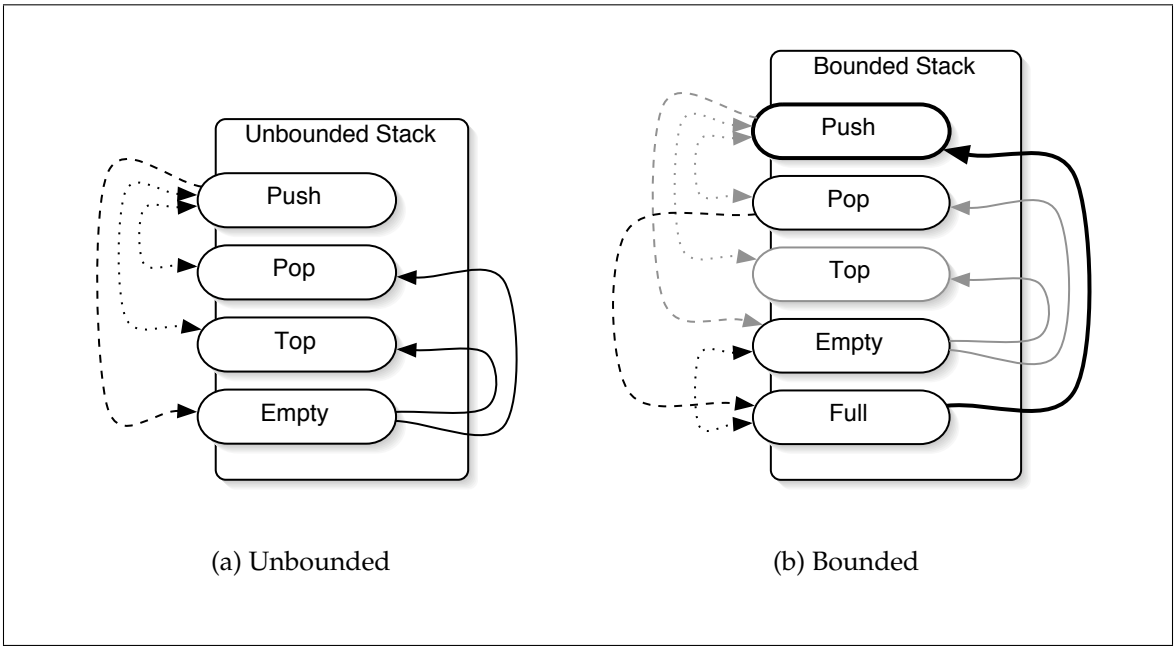


Figure 5.15: Bounded and unbounded stack specifications.

and unbounded stacks do not conform to each other, which is appropriate if we intend to model behavioral subtyping [LW94]. However, both interfaces *do* conform to the `interface {f.pop, f.top, f.empty}` and thanks to structural conformance we can avoid explicitly introducing this "virtual supertype."

5.4 Evaluation

To evaluate the Lag*oo*na design framework, I compare it to a number of existing proposals for component-oriented programming languages and related language mechanisms.

5.4.1 Multimethods

Stand-alone messages and generic message forwarding can be related to the concept of *multimethods* [BKK+86, Moo86]. Multimethods, also called generic functions, generalize "regular" methods in that they are dispatched on *multiple* receiver objects simultaneously instead of a single one.

In a language supporting multimethods, such as Cecil [Cha97], stand-alone messages can be "emulated" by introducing an additional dispatch parameter modeling the originating module. Also, generic message forwarding can be emulated by subclassing the receiver to be adapted and adding the appropriate, more specialized multimethods with new functionality.

Despite recent progress regarding type-safety and modularity of multimethods [MC99], the concept is not yet supported in mainstream languages. Stand-alone messages are conceptually simpler than multimethods because they only rely on the established notion of modules and add no additional concerns for separate compilation. Generic message forwarding is also simpler to understand, however the concept is not as safe as multimethods have recently been made. Overall, the biggest advantage of the Lag*oo*na design framework over multimethods might well be that it maintains the established object-oriented programming style and only "adapts it" as far as necessary.

5.4.2 Units and Mixins

Recent work on *units* and *mixins* [FF98a, FF98b, FKF98, Fla99] is related to Lag*oo*na design framework in a more interesting way. Units and mixins also aim at the combination of modular and object-oriented language constructs.

Units provide a module concept that is more flexible than ours: Instead of fixing the import relations of a set of modules once and for all, units allow the composition of modules through separate linking specifications. This has several important applications, for example for the flexible creation of extended objects.

Mixins provide a variation of inheritance (in the sense of subclassing) that allows derived classes to be parameterized by different base classes. However, Lag*oo*na's approach to forwarding and composition already subsumes mixins: while for mixins the base class relation is determined when units are linked, in Lag*oo*na we can actually defer this relation until objects are instantiated.

In summary, the units idea is very valuable, and we hope to explore the integration of a more flexible module system (with a distinct "units" flavor) into Lag*oo*na

	Message $\in$ **Type**	**Message** $\in$ **Module**
Method $\in$ **Type**	Object-Oriented	*Component-Oriented*
Method $\in$ **Module**	Useless?	Modular

Table 5.2: Explored language design space for messages.

in the future.

Chapter 6

Implementation

> *There is a widespread myth that a language designer can afford to ignore machine efficiency, because it can be regained when required by the use of a sophisticated optimizing compiler. This is false: there is nothing that the good engineer can afford to ignore.*
>
> — C. A. R. HOARE [Hoa73]

In this chapter, I discuss a number of implementation concerns for component-oriented programming languages, particularly for languages that follow the design framework developed above. I first discuss some general implications of component software for computer systems and language implementations (Section 6.1). Then I briefly describe two prototype implementations of Lagoona—the extensible interpreter PYLAG and the dynamically optimizing compiler LAVA—and review the design decisions made for each (Section 6.2). Finally I discuss efficient techniques for *message dispatch*, an area where languages following my design framework require more general solutions than those commonly adopted for object-oriented languages (Section 6.3). Since I am mainly concerned with language design and not language implementation in this dissertation, I follow HOARE's advice and focus on "non-pessimistic" solutions that achieve decent performance without sophisticated optimizations [Hoa73].

6.1 General Concerns

The implementation of component-oriented programming languages differs considerably from the implementation of traditional languages. It is not sufficient to simply implement a compiler and a basic runtime system, instead a complete *execution environment* has to be realized.

Apart from fulfilling traditional compilation tasks, this environment must at least provide for dynamic loading and dynamic linking of software components at runtime [Fra97a]. Safety and security concerns have to be addressed as well, for example by providing garbage collection to ensure memory safety [SGM02] and by verifying security properties of components acquired from potentially malicious sources [ADF+01].

6.1.1 Efficienct Execution

The efficient execution of software written in high-level languages is a primary concern for programming language implementation [Wir96, App02]. In bridging the gap between software and hardware, compilers rely on a variety of automated analyses to ensure source code is translated into native code that makes efficient use of machine resources [WM95, NNH99].

The concerns involved on both sides of this process are frequently at odds. While programming languages strive to offer sophisticated abstractions to aid programmers in expressing their designs accurately, computer systems perform most efficiently once all these abstractions have been elided from the program.

Figure 6.1 on the following page illustrates this mismatch between software concerns and hardware concerns with a simple example. The software perspective on the left side shows three abstractions A, B, and C that depend on each other, for example through some form of procedure call. Whether we consider these abstractions to be procedures, modules, classes, or components, the point is that each abstraction is isolated from the others as much as possible. The hardware perspective on the right side shows how the code generated for these abstractions

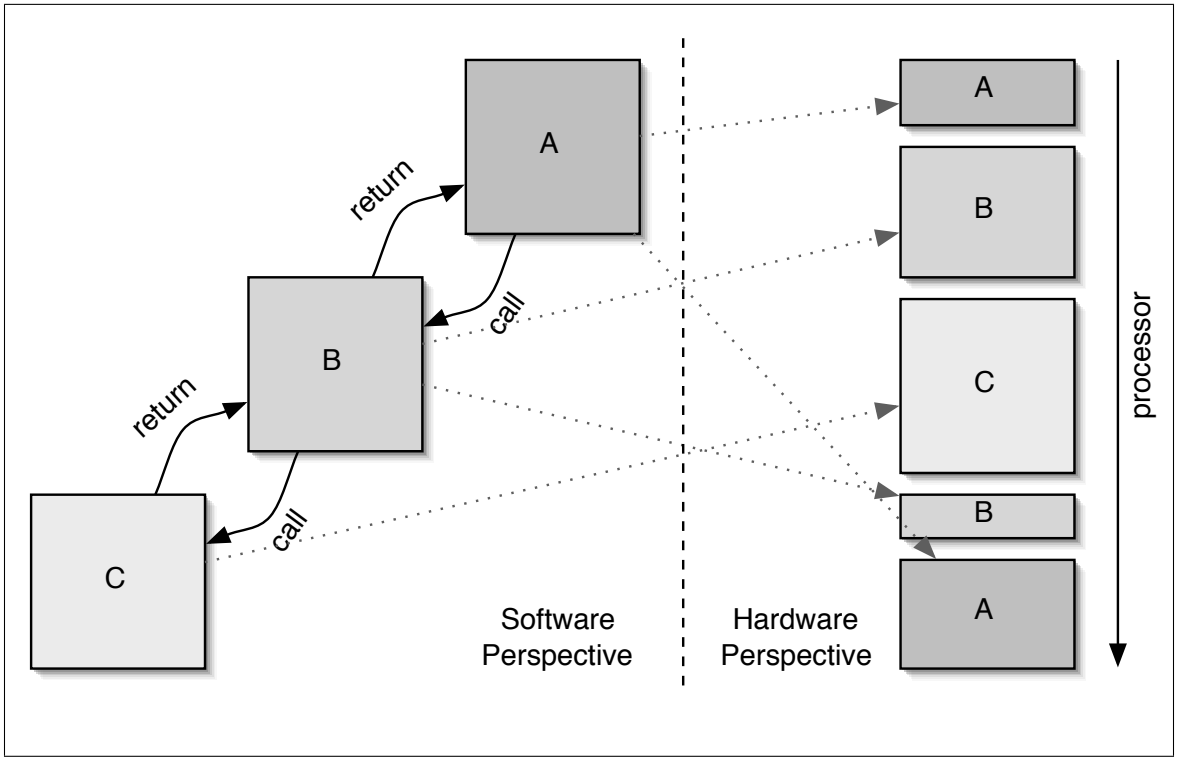


Figure 6.1: Mismatch between software (nice abstractions) and hardware (efficient execution). If the abstractions are components, even common optimizations such as *inlining* can not be performed at compile time.

should be laid out in memory for a pipelined processor architecture to achieve maximum performance [HP96, PH98]. Instead of the various branch instructions that a straightforward compiler would generate to cross abstraction boundaries, we would prefer not to have any branch instructions and to execute linear code instead.

In the case of component-oriented programming languages, the potential for traditional optimizations is inherently limited. The reason for this is simply that the analyses required achieve better results when they can examine a complete system as a whole instead of its parts in isolation. However, component software is by definition never "complete" in this sense. At the time frameworks or components are compiled, very little information about the configuration of the deployed systems they will be part of is available. This remains true even in the case of a software vendor who supplies a framework together with a number of components for it. The principle of distributed extensibility requires that third parties can isolate these components and either replace them or reuse them with other frameworks.

However, while frameworks and components must be *deployed* in a completely isolated form, nothing prevents us from "tearing down" these barriers once they have actually been *composed* into a running system. To enable optimization of component software, we therefore have to employ *dynamic compilation* techniques and defer many code generation tasks from compile-time to load-time. To avoid noticeable delays caused by time-consuming analyses and optimizations, we also have to rely on *dynamic* and *continuous optimization* that exploits idle time instead of interrupting the user's workflow. Note that *any* component-oriented programming language will have to utilize such techniques to achieve optimal performance, the concern is not limited to Lagoona.

6.1.2 Convenient Deployment

The notion of distributed extensibility (see Section 2.1.2) allows *any* party to extend the functionality of a system at *any* time, including users of the system. For this reason, the process of acquiring and integrating components can not assume a lot of

technical sophistication and must proceed with a minimum of intervention. Component software should therefore be deployed in "binary" form [SGM02]. However, the word "binary" does not necessarily imply "native code" in this context. Instead, it stands for a combination of the following requirements:

- Components are *internally* consistent (i.e. type-checked).
- Components contain *metadata* about the required environment.
- Components can be analyzed and executed with *reasonable* efficiency.

While it is possible to extend native code to fulfill these requirements [Nec98], there is an additional dimension to consider as well: If component vendors have to provide native code versions of *all* the components they offer for *several* different platforms, the resulting management overhead can become a serious impediment. The binary form for component software should therefore be a *portable* intermediate representation that fulfills the above qualities.

The choice of a particular intermediate representation affects both the security of the execution environment as well as the performance of deployed components. Interestingly, identical concerns arise in the area of *mobile code*, where technologies like Sun's Java Virtual Machine [LY99] and the Microsoft's .NET architecture [ECM01] currently dominate. However, there is increasing evidence that intermediate representations based on virtual instruction sets are far from optimal for addressing security and efficiency concerns [ADF+01]. Intermediate representations such as *slim binaries* [Fra94, FK97], which are based on suitably encoded *abstract syntax trees* instead, seem to offer significant advantages without compensating drawbacks [SHF00, ADFvR01].

6.2 Prototype Implementations

Language design is interesting and even fun, but it does not exist in a vacuum. Design choices made must be validated and a straightforward approach is implementing the language in the form of a prototype interpreter or compiler.

6.2.1 The PYLAG Interpreter

I developed the first Lagoona compiler in 1998 as an extension to the Oberon system [WG92], but abandoned it as it became obvious that it would have only very limited impact. Since then, I have concentrated on an extensible Lagoona interpreter instead, the latest incarnation of which is implemented in Python [vR01] and code-named PYLAG for obvious reasons.[1]

The goal for PYLAG is to serve as a platform in which new language features and various Lagoona dialects can be explored effectively, and to this end it supports multiple frontends. As illustrated in Figure 6.2 on the next page, all frontends translate Lagoona source code into a common intermediate representation, which is then executed by the interpreter.

Efficiency is not a priority for PYLAG, it indeed uses none of the more efficient message dispatch techniques outlined below (see Section 6.3). Instead, each message send triggers a traversal of the internal object and type graph, quite similar to early Smalltalk implementations [GR83, Kay96]. It does, however, enforce the type rules discussed in Chapter 5 strictly and can be used to experiment with new variations of the same.

The frontend for Oberon-based [RW92] concrete syntax is complete and follows the original Lagoona syntax closely [Fra97b]. A frontend for the Java-based [GJSB00] concrete syntax used throughout the dissertation is under development.

6.2.2 The LAVA Compiler

The LAVA project, headed by ANDREAS GAL, provides a second prototype implementation of Lagoona, exclusively for the Java-based syntax. Its architecture is illustrated in Figure 6.3 on page 104 and consists of a compiler and a dynamically optimizing runtime system, both written in Lagoona.[2]

[1] Due to chronic instabilities in the actual surface syntax, PYLAG has not yet been released publicly. I hope to remedy this situation in the near future.

[2] To ease bootstrapping, LAVA is currently hosted in Microsoft's .NET architecture [ECM01]. It is expected to become self-hosting in the near future.

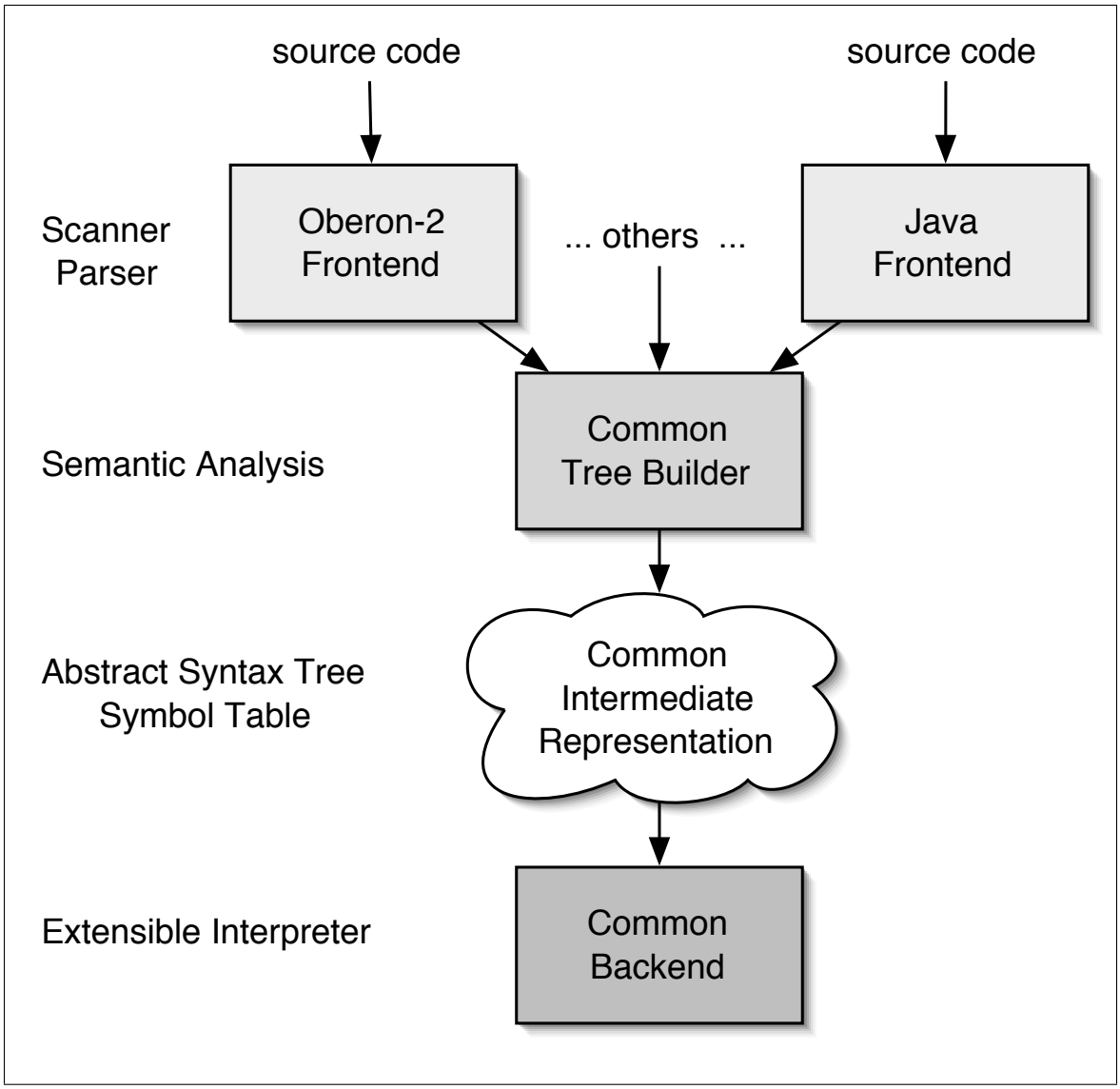


Figure 6.2: Architecture of the prototype Lagoona interpreter PYLAG consisting of multiple frontends (according to the style of concrete syntax supported) and common analysis and backend phases. Arrows indicate data flow.

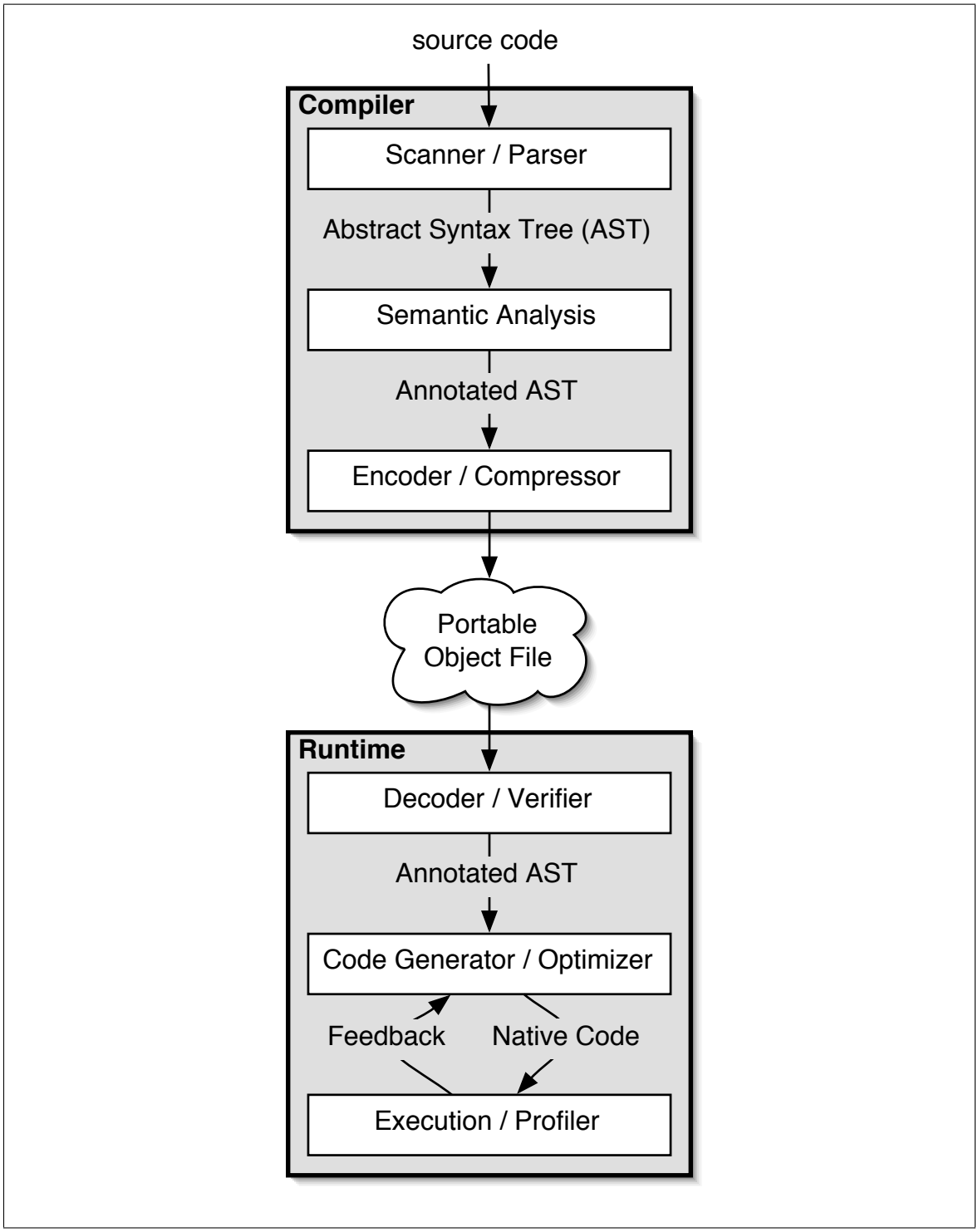


Figure 6.3: Architecture of the prototype Lag*oo*na compiler LAVA consisting of a compiler frontend and a dynamic code generation backend including a profiler. Arrows indicate data flow.

The goal for LAVA is to explore static and dynamic optimization techniques for Lagoona [GFF02, FGF02]. The compiler consists of a scanner and parser stage, followed by semantic analysis and static type-checking. The output of the compiler is an annotated abstract syntax tree, which serves as a portable intermediate representation.

Previous work established such a format as especially suitable for dynamic code generation and optimization [FK97, KF00, KF01]. In particular, it simplifies both the code verification step required by the runtime system as well as the generation of optimized native code.

6.3 Message Dispatch

The central concept Lagoona retains from object-oriented programming languages is *inclusion polymorphism* (also *subtype polymorphism*), the ability to dynamically use objects of a subtype in most contexts that statically expect one of its supertypes [CW85].[3] In language implementations, polymorphism of this kind leads to the problem of *message dispatch* which we can state as follows for Lagoona:

> **Message Dispatch:** Given a message *msg* to send and an interface reference *rcv* to a receiving object, locate the method *mth* in the implementation type *it* of *rcv* that should be invoked for *msg*.

Since the implementation type can change with each assignment to the interface reference, message dispatch must obviously be performed at runtime.

Message dispatch is commonly implemented using various runtime data structures [Dri99, App02]. For most established object-oriented languages, these data structures are constructed by the runtime system when an application is started, based on information supplied by the compiler. They are subsequently used by code the compiler generated for message sends and usually remain constant for the execution of the program.

[3] Note that the term *implementation polymorphism* would be more appropriate here since Lagoona restricts polymorphism to "many forms" of (concrete) implementation types that can be used where "one form" of (abstract) interface type is expected.

Message dispatch has been studied extensively for a variety of object-oriented programming languages [Dri93, VH94, DH95, ZCC97, LM98, CC99, SHR+00] and is closely related to the problem of *type inclusion* required for runtime type tests and type casts [VHK97]. However Lag*oo*na's object model differs from the "standard model" significantly enough to make many of these techniques difficult to apply.

In the following, I address the implementation of message dispatch and type inclusion for Lag*oo*na without concern for possible dynamic optimizations. The LAVA project (see Section 6.2.2) focuses on these advanced implementation issues with the goal of making Lag*oo*na competitive with other aggressively optimizing implementations of object-oriented programming languages [GFF02, FGF02].

However, even in light of more sophisticated approaches, conservative techniques still have a number of advantages. First and foremost, they are needed for environments such as embedded systems, in which dynamic optimization is still deemed too expensive (see Section 7.6). Second, they make execution times for basic operations such as message sends are much more predictable, which is important for real-time systems. Finally, sticking to a conservative approach simplifies the compiler and the runtime system considerably, which tends to influence its reliability—or at least our confidence in its reliability—positively.

6.3.1 Basic Dispatch Techniques

Since message dispatch is a pervasive operation in object-oriented programs, its efficiency is of primary concern. In early Smalltalk implementations [GR83, Kay96], each message send would trigger a traversal of the internal object and class graph, leading to comparatively low performance. However, the problem was soon addressed in various ways, either through the use of hashing selectors (message sends in Smalltalk terminology), caching previous results, or the use of tables that linearize the graphs involved for faster access [DS84, SUH86, Atk86].

The use of dispatch tables has been most widely adopted for object-oriented programming languages such as C++ [Str00], Java [GJSB00], or Lag*oo*na's ancestor Oberon [RW92]. While table-based techniques are not necessarily the most efficient

solution for *all* scenarios, they *do* have the advantage of predictable, constant time performance. Figure 6.4 on the next page illustrates the basic strategies for table-based dispatch.

The problem of finding the appropriate method to invoke given a message and an implementation type naturally leads to the idea of using a two-dimensional table as shown in Figure 6.4(a). Messages as well as implementation types are assigned unique identifiers, usually small integers, and the compiler emits code indexing this table to perform message dispatch. Depending on the amount of "dynamism" in the object model of the underlying language, various tradeoffs for *assigning* and *obtaining* these unique identifiers arise.

- **Assigning Unique Identifiers:**
  - In a closed system, where the compiler can perform global analysis, all identifiers can be assigned *statically* by the compiler.
  - In an open system, where new messages and implementation types can appear at runtime, all identifiers have to be assigned *dynamically* by the runtime system.
- **Obtaining Unique Identifiers:**
  - If messages are first-class citizens, each message reference must contain a *message identifier* at runtime.
  - If objects are first-class citizens, each object reference must contain a *type identifier* at runtime.

This model of dispatch is more general than usually discussed, but we will need it to better understand the tradeoffs made for Lag*oo*na in the following.

Several comments are in order at this point: First, Lag*oo*na obviously assumes open systems and we therefore have to postpone assignment of unique identifiers to runtime. However, the code emitted for a module also requires *locally* unique identifiers, which have to be mapped to globally unique ones at runtime.[4]

[4] Assuming *static* compilation; using *dynamic* compilation we can avoid this additional mapping.

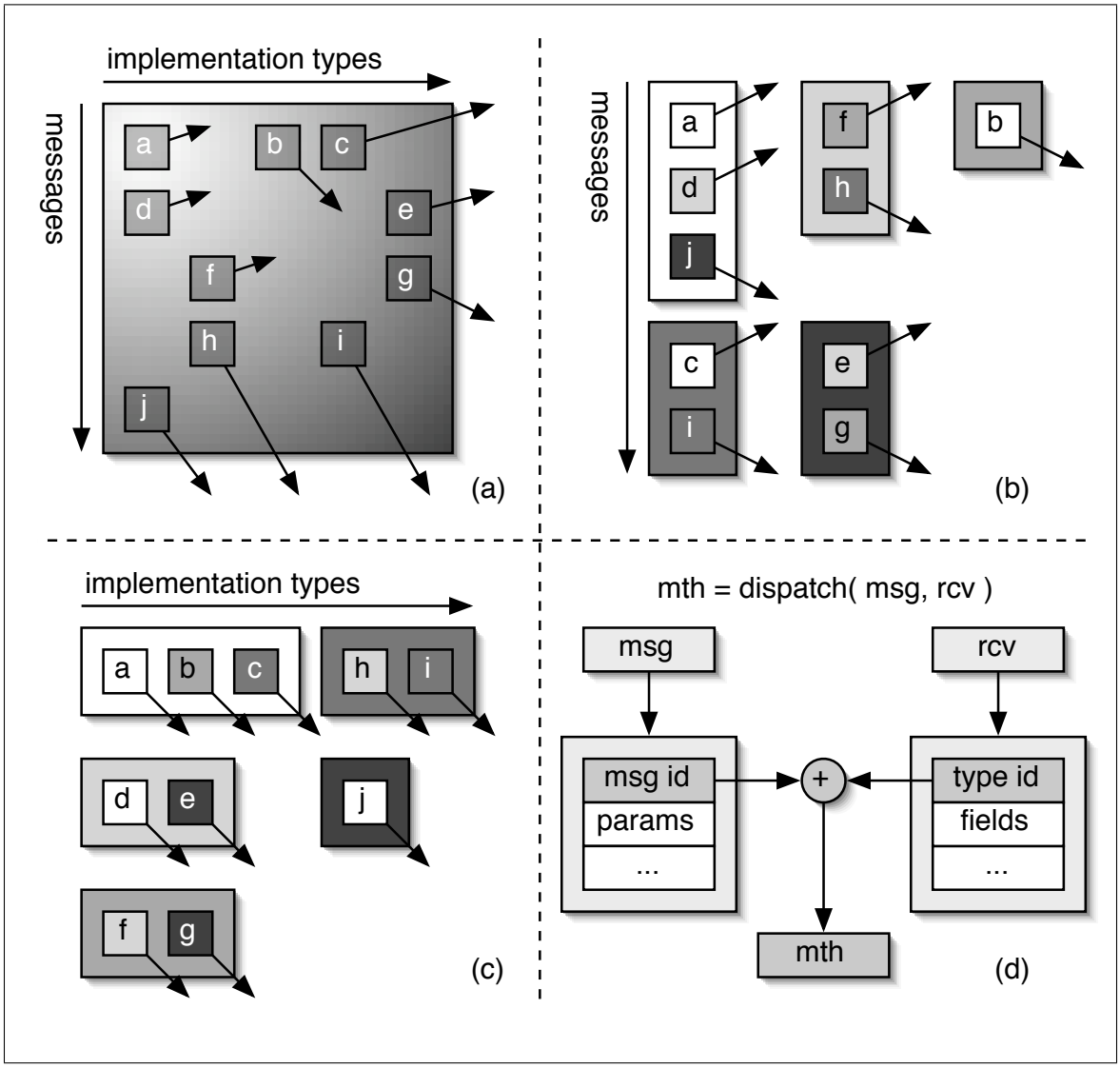


Figure 6.4: Basic data structures for message dispatch. A sparse table mapping (message, implementation type) pairs to methods (a). Slicing the table by implementation type (b). Slicing the table by message (c). The dispatch process in its most general form (d); "+" should be read as "index dispatch table," not as literal addition.

Second, Lagoona's messages are not first-class citizens in the usual sense of that term, i.e. they can not be assigned to variables (or passed to and returned from procedures). Nevertheless, a form of "message reference" is used as part of generic message forwarding, where the identifier `current` opaquely refers to the "currently active message" without information about its actual identity (see Chapter 4 and Chapter 5).[5] I'll return to this issue in Section 6.3.5 below.

Finally, the explicit mention of "first-class objects" might seem confusing. If objects were not first-class, we would not have to consider the problem of message dispatch at all, since the *exact* implementation type would be known at compile-time. However, Lagoona's rules for implementation types, namely that they can not be used polymorphically with *other* implementation types, enable us to avoid message dispatch at runtime in exactly this sense.

Figure 6.4(d) illustrates the most general case of message dispatch in this model. Message references as well as object references require a *tag* containing their globally unique identifier in addition to their actual data contents. The compiler emits code to dereference both pointers, obtain both tags, and index the dispatch table to invoke the appropriate method. As pointed out above, Lagoona's object model allow us to avoid either or even both of these separate tags in certain situations. These peculiar requirements are the primary reasons why established dispatch techniques can not be applied to Lagoona in a straightforward way.

Two-dimensional dispatch tables as shown in Figure 6.4(a) are usually sparse since most implementation types only support a comparatively small number of messages. The only exception to this general rule are special messages such as `initialize` and `finalize`. Various approaches for reducing the size of this dispatch table have been explored before, see for example [Dri99] for a survey. Approaches based on *compression* of the table often rely on global information and are therefore not an ideal fit in open systems. Furthermore they often can require a complete *rebuilding* of the compressed table when new messages or im-

[5] However, we have recently experimented with first-class messages for Lagoona to explore its application to parallel and distributed systems as well as to support more fine-grained routing of messages through reflective capabilities [WY88, Tem94].

plementation types are loaded. A more common technique to reduce the size of the dispatch table is to *slice* it, either by implementation type or by message. These two options are illustrated in Figure 6.4(b) and Figure 6.4(c) respectively, and the former is used regularly in object-oriented programming languages such as C++ [Str00] and Java [GJSB00].

The basic idea is to replace the "type tag" of each object with a pointer to an appropriate *portion* of the dispatch table directly (or alternatively, to replace the "message tag" of a message in a similar way). In established object-oriented languages this idea works well because inheritance induces a tree structure in which subclasses have *at least* the methods their superclasses have, possibly more. Since these languages do not separate subtyping from subclassing (and hence messages from methods), a unique offset can be assigned to each method in such a table. In Lag*oo*na, however, we can not assign identical offsets to identical messages when they are part of different implementation types. This is illustrated in Figure 6.4 on page 108 by the varying offsets that messages of the "same color" receive in different type-based dispatch tables (and a symmetric problem exists for message-based slices).[6] After this lengthy background on message dispatch, I will now turn to the specific techniques for languages following the Lag*oo*na design framework.

6.3.2 Building Dispatch Data Structures

Implementing message dispatch for Lag*oo*na requires keeping track of all messages and implementation types (including methods) currently loaded. As will become obvious below, we also need to keep track of interface types which do not appear explicitly in the model sketched above. Figure 6.5 on the next page illustrates the following discussion of the basic data structures.

When a module is loaded, each of the messages it declares is entered into a global message table and given a unqiue identifier. Note that it is not necessary to check for potential duplicates during this process since messages imported from

[6] Note that we can not leave "holes" in these tables, otherwise we would not save space compared to the full two-dimensional table.

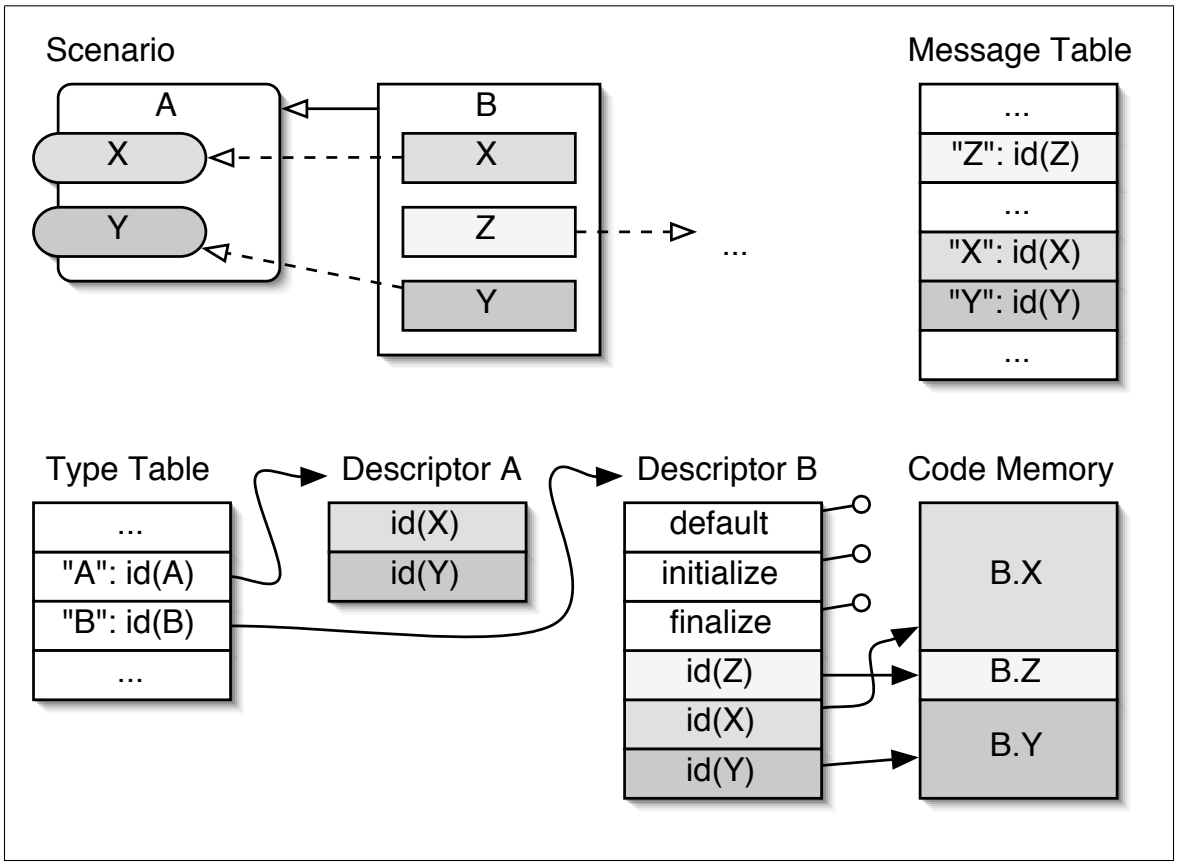


Figure 6.5: Layout of the descriptor tables generated at load-time (module names elided). Message Z was imported and thus received a "lower" identifier.

other modules can *not* be re-exported. For each implementation type, a descriptor containing the unique identifiers of each implemented messages as well as a pointer to the relevant method is allocated. In this descriptor, the first three slots are reserved for the `default`, `initialize`, and `finalize` methods, while the remaining slots are sorted by message identifier to obtain a canonical form of the dispatch table. As with messages, each implementation type is also entered into a global table and given a unique identifier.

In the case of interface types, however, the process of building the dispatch data structures is somewhat more involved. The actual descriptors themselves consist of the unique identifiers of all messages the type mandates, sorted as in the case of implementation types. However, since interface types utilize structural conformance, we need to take care not to create *duplicate* entries for types that are

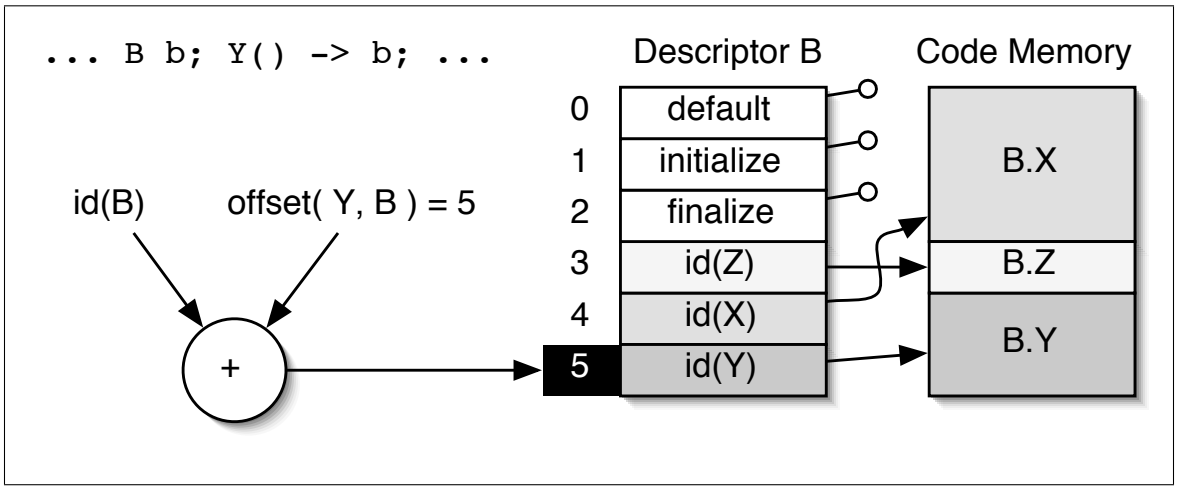


Figure 6.6: Resolving the message dispatch for implementation types at link-time.

introduced in separate declarations but which are structurally identical. We use a separate hash table data structure not shown in Figure 6.5 to detect duplicates in the following way. After building a descriptor for the interface type, we compute a hash value over the identifiers of *all* messages the type mandates. For this to work, it is important that descriptors are built in a canonical, sorted form. If the descriptor is not a duplicate, it is inserted into the hash table, otherwise the descriptor found in the hash table is used. We also insert interface types into the global type table for certain optimizations (see below).[7]

Note that the size of each descriptor table can be determined at compile-time, but the loader and linker are responsible for populating the descriptor tables with the appropriate identifiers and pointers.

6.3.3 Strict Message Sends

We have to distinguish two cases for the dispatch of strict message sends depending on whether the receiver is bound to an implementation reference or an interface reference. In both cases, however, we know that the message sent *will* be handled

[7]The names of explicitly declared interface types could in principle be fully elided without loss of generality, however we currently retain them in object files. In the future, a better choice might be to remove named interface types altogether since they can introduce unwanted compile-time dependencies.

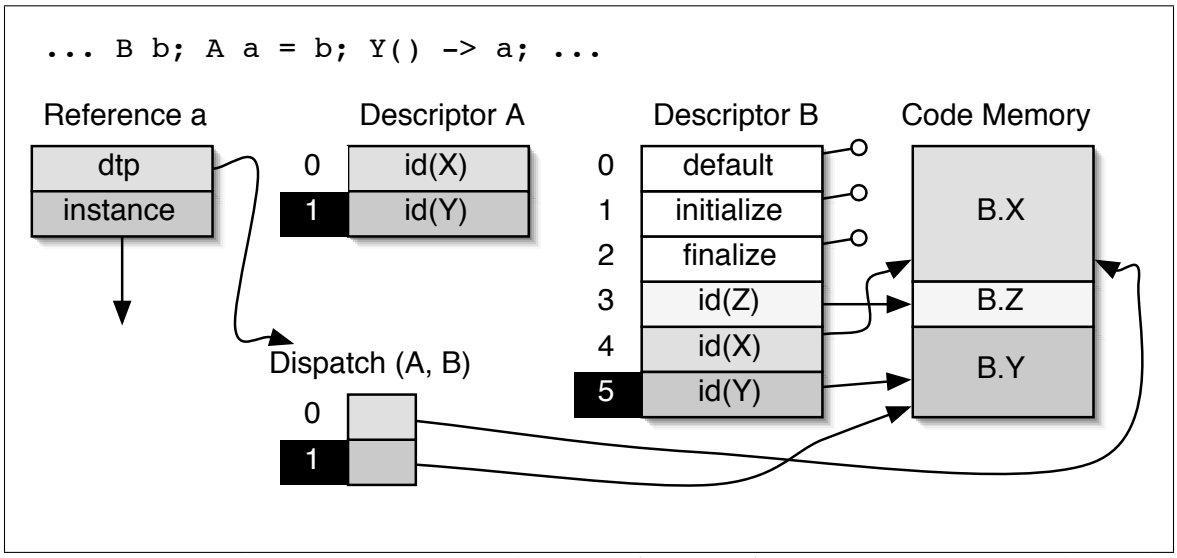


Figure 6.7: Resolving the message dispatch for interface types at runtime through customized dispatch tables.

by the receiver because of type checking (see Chapter 5).

If a message m is send to an instance through an implementation reference of type T, the address of the target method can be obtained already at link-time by accessing the descriptor table of T using the compile-time calculated offset (Figure 6.6 on the preceding page). This is possible since the reference can *never* point to an instance of another implementation type, a fact ensured by the type system. Dispatching a message this way therefore incurs no additional runtime overhead once loading and linking are completed.

To dispatch messages sent through an interface reference, a *dispatch table* has to be generated. This dispatch table maps the message offsets of a particular interface to the actual methods to be executed on arrival of that message in a particular implementation type. The set of methods to be matched to the messages has to be selected according to the actual type of the object which has been assigned to that interface reference (Figure 6.7). For every interface reference, the compiler allocates space for an instance pointer *and* a dispatch table pointer (dtp) used to send messages to the object hidden behind the interface. Thus, on the machine level two words are required to represent an interface reference.

Pre-generating all possible dispatch tables is a waste of space, as there are $n \times m$ possible combinations of n interface types and m implementation types. Instead, the mappings are created lazily at runtime whenever an instance is assigned to an interface reference, and held in a global dispatch table cache managed following an LRU scheme. [8]

6.3.4 Widening Interface References

In certain situations, it is desirable to explicitly widen the interface of an object reference. In the message set model this means that messages are added to the set of messages the object behind a particular reference is assumed to implement. However, in the general case widening can not be verified at compile-time for obvious reasons.

The verification of explicit casts is performed at runtime using the type descriptors. The message set of the object addressed by the reference is compared to the message set of the type to which the reference has been cast. If the conformity cannot be verified, the object must be incompatible to the interface and an exception is raised.

6.3.5 Blind Message Sends and Generic Forwarding

For blind message sends (see Section 5.2) we first attempt to dispatch as before. If the message can not be resolved successfully, we examine the `default` slot of the dispatch table and if it is filled, we invoke it with the current message id as an implicit parameter in a reserved register. The original parameters of the message are still on the call stack, but not accessible inside `default`. Message sends of the form `current => receiver` are treated as a special case by the compiler, but not by the runtime system. The message id is simply used directly without a separate lookup. Trough a chain of forwarding message sends, we can therefore avoid

[8] Note that this covers assignments that statically "look like" (interface type, interface type) pairs as well, since the second reference *must* refer to an instance and thus an implementation type.

duplicating parameters on the stack, which would be necessary if we implemented forwarding without a special language mechanism.

6.4 Summary

In this chapter, I outlined a number of general concerns for the implementation of component-oriented programming languages, reviewed two prototype implementations of Lagoona, and discussed an approach to message dispatch suitable for Lagoona's object model in detail.

Not all of the decisions described above were "set in stone" when work on Lagoona began, and since work on dynamic optimization for Lagoona is still ongoing [GFF02, FGF02], certain decisions might be revised again. Furthermore, while the "essence" of Lagoona has been fully implemented and is expected to remain stable, we are trying to improve Lagoona further. Concepts such as first-class messages or a more flexible module system will no doubt have an impact on the implementation.

In terms of "lessons learned" the LAVA compiler, written in Lagoona, was particularly important. First of all, its existence demonstrates the feasibility of the Lagoona design framework. More importantly, however, it yielded valuable insights on programming style. Lagoona places a particular emphasis on the clear separation of interface and implementation, both regarding types and regarding modules. In terms of software evolution, it proved to be valuable that this separation is enforced to a greater extent than in most established object-oriented languages. However, in terms of programming, we repeatedly found ourselves tempted to take various "shortcuts" that would violate this separation but which would be possible in languages like Java [GJSB00] or C++ [Str00]. Also, we repeatedly wanted to use inheritance in the sense of subclassing rather than forwarding as available in Lagoona. I believe that these experiences are part of learning the style of programming that component software requires while simultaneously "unlearning" what we had been doing for almost a decade prior. As was the

case with previous software development paradigms, such transitions are rarely straightforward, but almost always productive in the long run.

Working out the details of Lag*oo*na in terms of its implementation also proved to be harder than expected, for example in relation to interface types and their structural conformance rules. Our prior experience had been in languages where named-based conformance between types dominates. While implementing type checking for structural conformance was straightforward, the approach to storing this information in object files—in the sense of symbol files [Cre94, Wir96]—to support separate compilation was initially less obvious. We eventually developed the method described above, which relies on hashing to efficiently determine duplicate interface types, but initially we followed a much more complex approach based on automatically generating unique interface type names.

When the Lag*oo*na design framework is applied to an existing language and its implementation, these problems a bound to occur again since most existing compiler technology relies on name-based conformance.[9] More importantly, however, it may be difficult to achieve a straightforward integration of Lag*oo*na's concepts with the same elegance if the underlying language lacks a sensible module construct. This is true for many languages, including Java [GJSB00], C++ [Str00], and Eiffel [Mey92] that would otherwise be decent starting points. In the end, it may be the lack of modularity that these languages provide that might keep them from being used successfully for component-oriented programming.

[9] In this regard it is interesting to note that most formalizations of programming languages, even for those with name-based conformance, actually use structural conformance rules.

Chapter 7

Future Work

> *A fair conclusion might be that "why" is well understood, "what" is still subject to debate, and "how" is completely up in the air.*
>
> — NANCY G. LEVESON of Software Safety [Lev86]

As is the case with most research projects, the results I have presented in this dissertation "naturally" lead to further questions, to be addressed in the future. Some of these questions arise from various shortcomings of the completed project, while others become apparent along the way, but can not be investigated in detail due to external constraints. In this chapter, I briefly outline some of the areas—both for Lag*oo*na and for component-oriented programming languages in general—where future work seems most promising.

7.1 Static Typing and Message Forwarding

In Chapter 5, I discussed the design of Lag*oo*na's message send operators and showed that generic message forwarding, if allowed to be used as originally intended, leads to a loss of static typing. At compile-time, that is, when a component is created by a vendor, Lag*oo*na can not guarantee that a blind message send will be handled in the system as it is finally deployed. I also explained that we can not avoid this loss if extensibility in terms of messages is desired.

```
interface X { R, S, T }
interface Y { Q, R, S, T }

class A {
  ...
  method int Q() { ... }
  method void R() { ... }
  method int S( int x, int y ) { ... }
  method void T( string s ) { ... }
  ...
}

class B {
  A a;
  ...
  forward R, S -> this.a;
  ...
  method void T( string s ) {
    ... T( s ) -> this.a; ...
  }
  ...
}
```

Figure 7.1: A declarative form of forwarding to improve static typing in Lagoona. Forwarding relationships are made explicit instead of being "buried" inside the `default` method as arbitrary code.

If we accept this loss in extensibility and want to remedy the situation directly at the language level, the only possible alternative seems to be removing generic message forwarding in its current form. Instead of allowing arbitrary code in the `default` method, a more restricted, declarative form of forwarding could be used, making forwarding relationships more explicit. One possible form that such a mechanism could take is illustrated in Figure 7.1. Class `B` holds a reference to an `A` instance and declares that messages `R` and `S` will be forwarded to that instance unchanged. Message `T` is handled explicitly, presumably to "augment" its implementation in `A` in some way, but `Q` is neither forwarded automatically nor handled explicitly. Therefore `A` conforms to both `X` and `Y`, while `B` only conforms to `X`.

While the basic tradeoff between static typing and extensibility remains, the declarative approach has further advantages: It avoids the problem of "sudden feature acquisition" (see Chapter 5), and it also helps to make message dispatch more efficient since we do not have to rely on predictions about potential receivers anymore (see Chapter 6). The details of this declarative approach to message forwarding should be worked out to make a more informed decision as far as Lagoona is concerned.

7.2 Type Inference

The idea of *type inference* stems from the observation that types can often be determined automatically by the compiler, without the programmer declaring them explicitly [APS93, Age96, Sch95]. Trivial forms of type inference are used in almost all programming languages, for example when a compiler "infers" that the literal "`1`" has the type `int`. Similarly, structural conformance to interface types in Lagoona can be seen as a limited form of type inference: Unlike in "regular" object-oriented programming languages, conformance is never declared explicitly.

In functional programming languages like ML [MTHM97] and Haskell [PJ03], type inference has been generalized much further, to the point where it has become an "essential" aspect of the "programming experience." In these languages, types are inferred from the way identifiers are actually *used*, including types of functions. For example, a function `id` that simply returns its argument would be given a type of the form

$$\texttt{id} : \texttt{any} \to \texttt{any}$$

while a function `min` that returns the smaller of its two arguments would receive a type of the form

$$\texttt{min} : \texttt{ordered} \times \texttt{ordered} \to \texttt{ordered}$$

instead. These types are inferred from the *implementation* of their corresponding functions: `id` does not apply operation to its argument and thus "works" for `any`

type, while `min` needs to compare its arguments using a `less` operation, which is defined only for `ordered` types.

For Lag*oo*na, type inference in this style would enable *feedback* about minimal typing (Section 5.3.2). Consider a variable `x` of some interface type `X`. If inside a certain scope we only send two of the 27 messages supported by `X` through `x`, the compiler could issue a warning and suggest to use a smaller interface type instead. Note, however, that we can *not* propagate inferred types *across* module boundaries. The problem with this is that a change in the implementation could trigger a change in the type of a parameter, which could in turn break independently developed components. In Lag*oo*na, type inference would therefore take a different form than in established functional languages, besides having to deal with imperative features of course. Instead of inferring types purely "bottom-up," we need to infer types "top-down" as well. The problem we are left with is illustrated in Figure 7.2 on the next page, and it should prove interesting to investigate whether more efficient type inference algorithms can be found for this special case.

Besides using type inference to provide feedback for programmers, the idea might also be helpful to improve static typing in the presence of forwarding (see Chapter 5 and Section 7.1). Combined with basic data-flow analysis results, a type inference algorithm could conservatively approximate the possible sets of messages that will be handled through forwarding. Note that it is important to find an *efficient* type inference algorithm since this analysis can only be performed accurately at load-time or run-time in the system as it is finally deployed. The results of such an analysis would in turn help to perform more effective dynamic optimizations [Joh86, Atk86, BG93, APS93, ZCC97].

7.3 Dynamic Optimization

The dynamic optimizations performed by the LAVA compiler help Lag*oo*na's performance significantly, especially regarding message dispatch for strict message sends (see Chapter 5). However, they are currently less successful in the case of

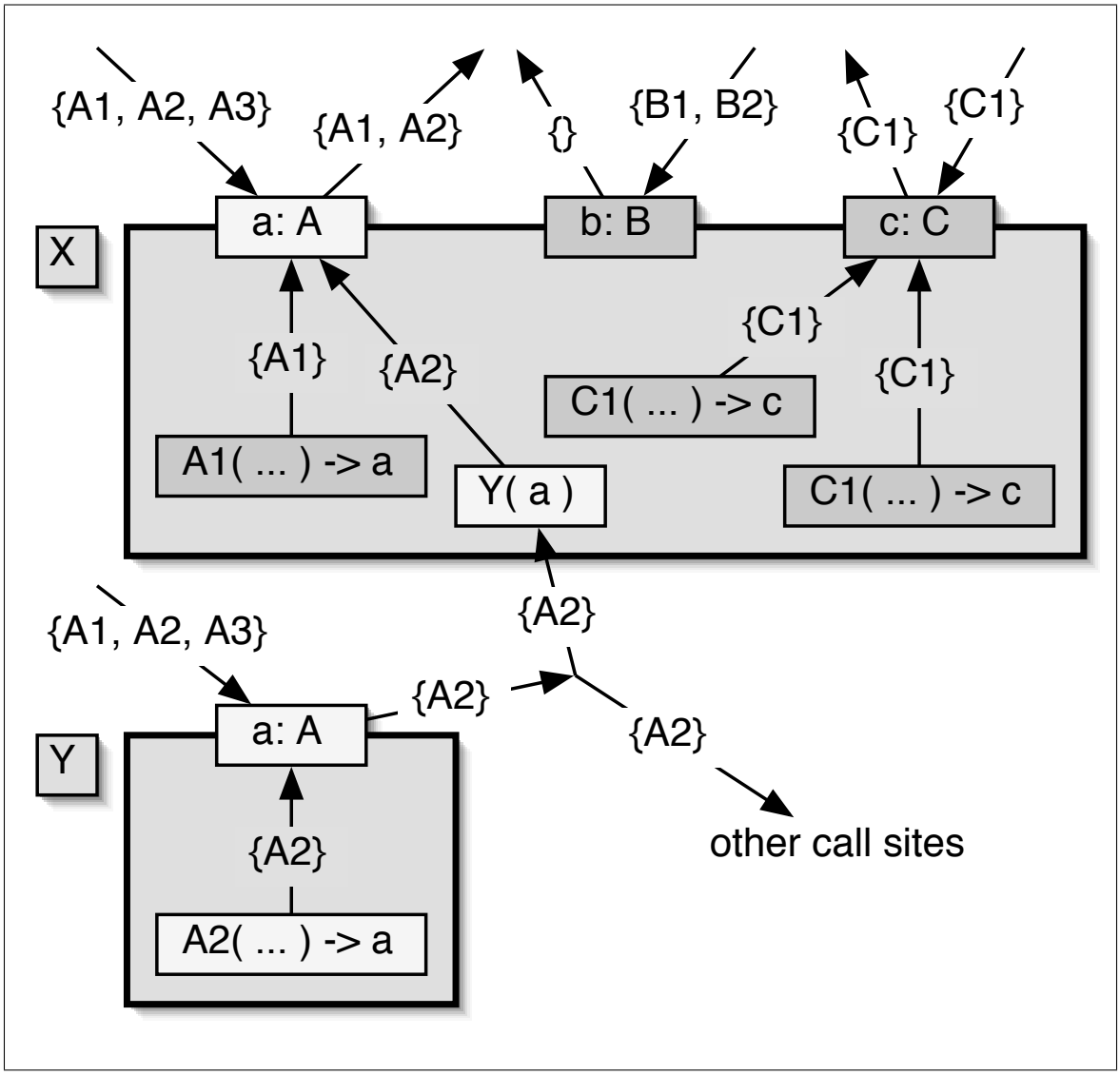


Figure 7.2: The "limited" type inference problem in Lagoona. Arrows *into* an abstraction represent the "maximum set" of messages possible (i.e. declared type), arrows *out of* an abstraction represent the "actual set" of messages used by the implementation (i.e. inferred type).

blind message sends that occur during generic message forwarding. There are two reasons for this, both of which should be addressed in the future.

First, as pointed out in Section 7.1 and Section 7.2 above, improvements in Lag*oo*na that would lead to tighter results for type-checking would help in optimizing these message sends further. Ideally, we would be able to predict the eventual target method invoked through a "chain" of forwarding relationships as soon as an instance is assigned to an interface reference. The question whether such changes to the language, which potentially restrict the extensibility of component software further, are acceptable remains to be explored.

Second, the current LAVA implementation in many cases fails to predict less frequently used message send operations correctly, leading to the dynamic compiler "wasting" significant amounts of time in optimizing code passages that are rarely used. In other words, we need to improve the pay-off prediction process that we apply to the feedback we obtain from the profiler. Said feedback is used to make decisions as to which parts of a system are optimized next. One way we plan to address this is by developing an improved suit of benchmarks that focus on the various patterns of call frequencies possible, and ideally we would like to measure the behavior of "real world" applications. However, porting such applications or even existing benchmark suits to Lag*oo*na is a slow process.

7.4 Aliasing and Representation Exposure

Composition, the mechanism at the core of component-oriented programming, is almost universally mapped to object references in programming languages (see Section 2.2). In contrast to object-oriented programming, where inheritance in the sense of subclassing dominates, composition is also the standard approach to reuse in component-oriented programming. However, whereas inheritance is a "static" mechanism resolved at compile-time, composition is a "dynamic" mechanism. In this context, it becomes important to address issues of *representation exposure* and *abstract aliasing* directly at the level of programming languages [DLN98].

Consider the implementation of a class `Stack` once again. If we want to avoid implementing the actual data structure used inside this class, we have to obtain a reference to a suitable data structure from elsewhere. An obvious choice is to specify this data structure using an interface type, and to require a conforming object reference in the initializer. However, another part of the system could retain this object reference and thus break the abstraction a `Stack` promises. For example, if an instance of some `List` class is passed, the internal state of the `Stack` could be changed in unexpected ways through a retained `List` reference.

Several language mechanisms addressing various aspects of this problem have been proposed in recent years [Alm97, Lei98, CPN98, VB99, BR00, MPH00]. For Lag*oo*na, I plan to either adopt an existing mechanism or (if none are suitable) design a customized mechanism in the near future. The static guarantees about possible dynamic aliasing relationships provided by such a mechanism should not only simplify reasoning about the correctness of component software, they should also prove helpful for certain code optimizations.

7.5 Versioning and Configuration Management

Software components retain their autonomous character even after they have been deployed as part of a software system. While this facilitates the addition, removal, and modification of components by third parties, it also creates new versioning and configuration management issues.

Traditionally, research in software configuration management has focused on assisting software vendors who work with source code and related artifacts, and who execute related development processes [Tic88, Tic92, CW98]. Versioning and configuration management support for software *consumers* and *third parties* is a relatively recent topic of interest, and would be especially beneficial for component-oriented programming.

Existing approaches to this kind of support often rely on meta data descriptions that are distinct from source code. For example, the Software Dock infrastructure

[HHW99], which addresses deployment-related activities such as release, install, update, and reconfigure, relies on *deployable software descriptions* (DSDs) written in a specialized declarative language. Other systems leverage existing research in software architectures [OMT98, KM98] and rely on architecture description languages (ADLs). The additional tools necessary to process these descriptions are often not integrated well with the underlying operating system or programming language, providing only a partial solution.

I believe that some—if not all—mechanisms to support deployment-side versioning and configuration management can be integrated into component-oriented programming languages and their runtime systems. In modular programming languages, symbol files are traditionally responsible for ensuring the version consistency of modules before they are loaded and linked [LS79, Cre94], and recent work suggests that versioning can indeed be lifted to the language level [Sew01]. Such an integration would simplify software maintenance since separate formalism such DSDs or ADLs become unneccesary. Furthermore, it would allow the compiler to leverage the additional information for optimizations.

7.6 Real-Time Programming and Embedded Systems

In order to become a "full-fledged" software development paradigm on par with structured, modular, and object-oriented programming, the ideas of component-oriented programming must be applicable universally. The domain of real-time and embedded systems seems particularly challenging in this regard.

On the one hand, available resources are extremely constrained: How could we ever hope to provide an execution environment supporting dynamic loading, compilation, linking, and possibly even optimization on a simple micro-controller? On the other hand, the flexibility afforded by component software would be especially beneficial in this domain. Embedded systems are often produced in great quantities, and, at this point in time, they are pervasive. The possibility of dynamically upgrading a system, in case of an urgent safety or security issue say, would

therefore be helpful to both producers and consumers of such systems.

As part of my dissertation research, I briefly investigated the suitability of Lag*oo*na for this domain [FFK99]. The only positive result, however, was an elegant approach to schedule "non-essential" computations in the face of hard deadlines. Given the (potential) benefits and (certain) challenges, I hope to investigate component-oriented programming for real-time embedded systems in more detail.

Chapter 8

Summary

> *Writing this sort of report is like building a big software system. When you've done one you think you know all the answers and when you start another you realize you don't even know all the questions.*
>
> — BRIAN RANDELL [BR70]

Although RANDELL's words of wisdom already make me worried about my next research publication, I am still glad to finally reach the last chapter of this one. In it, I summarize what has been achieved (Section 8.1), what remains to be improved (Section 8.2), and what can be learned from the Lagoona project (Section 8.3).

8.1 Achievements

In this dissertation, I developed a novel design framework for the organizational structure of component-oriented programming languages. The framework can be applied to type-safe core languages with arbitrary computational structure and is thus reusable. The Lagoona family of programming languages has been developed by applying the framework to the core of Oberon [RW92] and Java [GJSB00].

The framework is based on two novel language mechanisms, namely stand-alone messages and generic message forwarding. Stand-alone messages are bound to sealed modules instead of extensible types and therefore have globally unique

identities. Stand-alone messages allow languages implementing the framework to guarantee the following two properties:

> **Interface Combination:** Any combination of two or more interface types is itself a *valid* interface type preserving *all* constituent messages.
>
> **Interface Conformance:** Conformance between interface and implementation types is *structural* yet *safe* down to the level of constituent messages.

Both of these properties are required for programming languages that need to support the principle of distributed extensibility at the core of component-oriented programming. Both of these properties do not require any additional language mechanisms besides stand-alone messages.

Generic message forwarding is a compositional black-box code reuse mechanism for adapting and extending implementation types. Compared to inheritance or delegation, it does not suffer from the fragile base class problem. Compared to forwarding without explicit language support, it is more convenient to use and can be more efficient as well.

I have illustrated the utility of these mechanisms and the design framework in terms of minimal typing, retroactive supertyping, component reentrance, iteration abstraction, and component framework extensibility.

I have shown that the Lagoona framework occupies a previously unexplored point in the design space of programming languages and sheds new light on the *exact* combination of features from modular and object-oriented languages required for component-oriented programming. In particular, separating messages from methods can be viewed as another step towards the separation of concepts subsumed by classes in traditional object-oriented languages. Previous results in this direction include the separation of interface and implementation types (subtyping and subclassing) [Sny86] and the separation of modules from types [Szy92],

which both are by now widely accepted. I have also shown how Lag*oo*na's mechanisms can be implemented efficiently without undue overhead, sometimes even with definite performance advantages over established object-oriented languages. I have clarified the difference in expressive power between forwarding and recursive code reuse mechanisms such as inheritance and delegation that are unsuitable for component-oriented programming. I have also clarified the tradeoff between the level of extensibility required in a component framework and the level of type safety that can be guaranteed for it.

In summary, I hope that the design framework developed in this dissertation will enable future research on component-oriented programming languages to proceed with better focus and thus more productively.

8.2 Shortcomings

Although languages based on the Lag*oo*na design framework provide numerous advantages for component-oriented programming, there are several "remaining troublespots" as well.

Regarding stand-alone messages, the very fact that they *are* globally unique could turn out to be problematic since it might lead to an "explosion" of messages. Consider, for example, the problem faced by component vendors if there are 300 different messages defined for a certain operation, all with identical semantics, yet all unique by design of the mechanism. I have discussed this problem briefly in Section 3.4.7, Chapter 5, and Chapter 7, but possible mechanisms to avoid it seem less important than the question of whether it would actually arise. For obvious reasons, this can not be evaluated conceptually, a strategy that I have otherwise preferred in this dissertation. Instead, it will be necessary to collect experimental evidence and return to the issue in the future.

In the case of generic message forwarding, however, several conceptual problems regarding the safety and efficiency of the mechanism remain. As discussed in Chapter 4, generic message forwarding and the related concept of blind message

sends can not be statically checked. However, as pointed out there as well, this is a necessary consequence of the principle of distributed extensibility combined with the desire to allow extensibility even with regard to the messages exchanged through a component framework. Nevertheless, the problem of accidental feature acquisition seems quite troubling. Again, it is difficult to evaluate the actual problems caused by the use of generic message forwarding in this regard conceptually. The same applies regarding the efficiency of generic message forwarding. While it is obviously more expensive to forward messages along a chain of receivers than to dispatch within an inheritance hierarchy, it should be noted that forwarding affords much more flexibility. Furthermore, providing generic message forwarding on the language level is actually more efficient than implementing it "by hand" with regular message sends (see Chapter 6). I have outlined two possible approaches to these issues in Section 7.1 and Section 7.2, yet experimental evidence would surely be important here as well.

8.3 Conclusions

Time spent on the design of programming languages is frequently considered "time wasted" by those with a particularly "pragmatic" attitude. This is especially true in case of academic exercises such as this one, which can not reasonably be expected to "pay off" within a few years, and, in fact, may never do so.

If I had been discouraged by this several years ago, I certainly would not have learned as much about either programming languages or component software as I tried to share in this dissertation. This will remain true regardless of how well I was *actually able* to convey these insights, but I'll try to work on that…

From this perspective, the most important conclusion I can draw from the Lag*oo*na project is simply this: Programming language design is a viable and valuable approach to understanding new programming techniques and software development paradigms. Only when we try to "cast" the general ideas into concrete language mechanisms, and only when these mechanisms lead to coherent

programming languages, only then can we be sure to understand them.

On a less philosophical level, I believe the most surprising result of my work on component-oriented programming languages was finding a solution to the long-standing problem of "name clashes" in the form of stand-alone messages. In retrospect the solution seems quite trivial, but that hardly explains the number of trees that have been used to describe the problem. A classic article on programming in Simula contains the earliest mention of the problem I could find [DH72], and it continues to reappear regularly from then on [Sny86, Knu88, OH92, Mez97, BW00]. And of course it haunts and complicates most object-oriented programming languages. The second most surprising result is the wide array of applications stand-alone message open up in terms of safe structural conformance. I hope that at least these considerations will eventually "make it" into mainstream languages, following in the tradition of the proscription against `goto` [BJ66, Dij68], the `case` instruction [Hoa73], and the introduction of explicit `module` constructs [Par72, GMS77, LSAS77, Wir77].

Finally, I can not avoid to comment on the idea of component software itself. I believe the major result from McIlroy's original vision up to the much more evolved ideas of Szyperski is that they challenge and guide research. I do *not* believe, however, that we will ever see a true "free market of software components" in which competition rules and the best components win the day. The reason for this is not a flaw in the idea of component software, it is simply the fact that there are no "free markets" in the first place. Of course I am willing to be proven wrong on this judgement, and whatever the eventual outcome, component software sure provides research challenges for years to come...

> *It is the responsibility of intellectuals to speak the truth and to expose lies. This, at least, may seem enough of a truism to pass without comment. Not so, however. For the modern intellectual, it is not at all obvious.*
>
> — Noam Chomsky [Cho67]

Bibliography

[ADF+01] Wolfram Amme, Niall Dalton, Michael Franz, Peter H. Fröhlich, Vivek Haldar, Peter S. Housel, Jeffery von Ronne, Christian H. Stork, and Sergiy Zhenochin. Project TRANSPROSE: Reconciling mobile-code security with execution efficiency. In *Proceedings of the DARPA Information Survivability Conference and Exhibition*, pages II.196–II.210, Anaheim, CA, June 2001.

[ADFvR01] Wolfram Amme, Niall Dalton, Michael Franz, and Jeffery von Ronne. SafeTSA: A type safe and referentially secure mobile-code representation based on static single assignment form. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, Snowbird, UT, June 2001.

[ADH+98] H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, N. I. Adams IV, D. P. Friedman, E. Kohlbecker, Jr. G. L. Steele, D. H. Bartley, R. Halstead, D. Oxley, G. J. Sussman, G. Brooks, C. Hanson, K. M. Pitman, and M. Wand. Revised^5 report on the algorithmic language scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, August 1998.

[Age96] Ole Agesen. *Concrete Type Inference: Delivering Object-Oriented Applications*. PhD thesis, Department of Computer Science, Stanford University, 1996. Published by Sun Microsystem Laboratories (SMLI TR-96-52).

[Alm97] P. S. Almeida. Balloon types: Controlling sharing of state in data

types. In M. Aksit and S. Matsuoka, editors, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 19–32, June 1997.

[Ame87] Pierre America. Inheritance and subtyping in a parallel object-oriented language. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 276 of *Lecture Notes in Computer Science*, pages 234–242. Springer-Verlag, 1987.

[App02] Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 2nd edition, 2002.

[APS93] Ole Agesen, Jens Palsberg, and Michael I. Schwartzbach. Type inference for SELF: Analysis of objects with dynamic and multiple inheritance. In Oscar M. Nierstrasz, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 707 of *Lecture Notes in Computer Science*, pages 247–267. Springer-Verlag, 1993.

[Atk86] Robert G. Atkinson. Hurricane: An optimizing compiler for Smalltalk. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 151–158, Portland, OR, November 1986.

[BDMN73] M. G. Birtwistle, Ole-Johan Dahl, B. Myhrhaug, and Kristen Nygaard. *Simula Begin*. Petrocelli / Charter, New York, NY, 1973.

[BG93] Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In Andreas Paepcke, editor, *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 215–230, Washington D.C., September 1993.

[BGP00] Lazlo Böszörmenyi, Jürg Gutknecht, and Gerhard Pomberger, editors. *The School of Niklaus Wirth: The Art of Simplicity*. Morgan Kaufman, September 2000.

[BJ66] C. Boehm and G. Jacopini. Flow diagrams, turing machines and languages with only two formation rules. *Communications of the ACM*, 9(5):366–371, May 1966.

[BKK+86] Daniel G. Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. CommonLoops: Merging Lisp and object-oriented programming. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 17–29, Portland, OR, November 1986.

[BM94] Marc M. Brandis and Hanspeter Mössenböck. Single-pass generation of static single-assignment form for structured languages. *ACM Transactions on Programming Languages and Systems*, 16(6):1684–1698, November 1994.

[BR70] John N. Buxton and Brian Randell, editors. *Software Engineering Techniques: Report on a Conference Sponsored by the NATO Science Committee, Rome, Italy, 27th to 31th October, 1969*. Scientific Affairs Division, NATO, Brussels, Belgium, April 1970. Available at `http://www.cs.ncl.ac.uk/people/brian.randell/home.formal/NATO/`.

[BR00] Ciaran Bryce and Chrislain Razafimahefa. An approach to safe object sharing. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 367–381, Minneapolis, Minnesota, October 2000.

[Bro87] Frederick P. Brooks, Jr. No silver bullet: Essence and accidents in software engineering. *IEEE Computer*, 20(4):10–19, April 1987. Reprinted in [Bro95].

[Bro95] Frederick P. Brooks, Jr. *The Mythical Man-Month: Essays on Software Engineering (20th Anniversary Edition)*. Addison-Wesley, 1995.

[BW98] Martin Büchi and Wolfgang Weck. Compound types for Java. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 362–373, Vancouver, British Columbia, October 1998.

[BW00] Martin Büchi and Wolfgang Weck. Generic wrappers. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 201–225, Sophia Antipolis and Cannes, France, June 2000.

[Car89] Luca Cardelli. Typeful programming. SRC Research Report 45, Digital Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, May 24, 1989.

[CC99] Craig Chambers and Weimin Chen. Efficient multiple and predicate dispatching. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 238–255, Denver, CO, November 1999.

[CDG+91] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 language definition. In Greg Nelson, editor, *Systems Programming in Modula-3*, chapter 2, pages 11–66. Prentice-Hall, 1991.

[CE00] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[Cha97] Craig Chambers. The Cecil language: Specification and rationale. Technical report, Department of Computer Science and Engineering, University of Washington, Box 352350, Seattle, WA 98195-2350, USA, March 1997. Available at `http://www.cs.washington.edu/research/projects/cecil/`.

[Cho67] Noam Chomsky. The responsibility of intellectuals. *New York Review of Books*, September 23, 1967. Reprinted in [Cho87].

[Cho87] Noam Chomsky. *The Chomsky Reader*. Pantheon Books, New York, NY, 1987.

[CHP99] Karl Crary, Robert Harper, and Sidd Puri. What is a recursive module? In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 37–49, Atlanta, GA, May 1999.

[CPN98] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 48–64, Vancouver, British Columbia, October 1998.

[Cre94] Régis Crelier. *Separate Compilation and Module Extension*. PhD thesis, Swiss Federal Institute of Technology (ETH), Zürich, Switzerland, 1994.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[CW98] Reidar Conradi and Bernhard Westfechtel. Version models for software configuration management. *ACM Computing Surveys*, 30(2):232–282, June 1998.

[DDH72] Ole-Johan Dahl, Edsger Wybe Dijkstra, and Charles Antony Richard Hoare, editors. *Structured Programming*. Academic Press, London, England, 1972.

[DeM79] Tom DeMarco. *Structured Analysis and System Specification: Tools and Techniques*. Prentice-Hall, 1979.

[DH72] Ole-Johan Dahl and Charles Antony Richard Hoare. Hierarchical program structures. In Ole-Johan Dahl, Edsger Wybe Dijkstra, and Charles Antony Richard Hoare, editors, *Structured Programming*, pages 175–220. Academic Press, London, England, 1972.

[DH95] Karel Driesen and Urs Hölzle. Minimizing row displacement dispatch tables. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 141–155, Austin, TX, 1995.

[Dij68] Edsger Wybe Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, March 1968.

[DLN98] David L. Detlefs, K. Rustan M. Leino, and Greg Nelson. Wrestling with rep exposure. SRC Research Report 156, Digital Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, July 29, 1998.

[Dri93] Karel Driesen. Selector table indexing & sparse arrays. In Andreas Paepcke, editor, *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 259–270, Washington D.C., September 1993.

[Dri99] Karel Driesen. *Software and Hardware Techniques for Efficient Polymorphic Calls*. PhD thesis, Department of Computer Science, University of California, Santa Barbara, CA, June 1999. Also published as Technical Report TR-CS-99-24.

[DS84] L. Peter Deutsch and Allan M. Schiffmann. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, pages 297–302, Salt Lake City, UT, January 1984.

[ECM01] ECMA. The .NET Common Language Infrastructure. Technical Report TR/84, ECMA, Geneva, Switzerland, June 2001.

[FF98a] Robert Bruce Findler and Matthew Flatt. Modular object-oriented programming with units and mixins. In *Proceedings of the International Conference on Functional Programming (ICFP)*, pages 94–104, Baltimore, MD, 1998.

[FF98b] Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 236–246, Montreal, Canada, June 1998.

[FF00] Peter H. Fröhlich and Michael Franz. Stand-alone messages: A step towards component-oriented programming languages. In Jürg Gutknecht and Wolfgang Weck, editors, *Proceedings of the Joint Modular Languages Conference*, volume 1897 of *Lecture Notes in Computer Science*, pages 90–103, Zürich, Switzerland, September 2000. Springer-Verlag.

[FF01] Peter H. Fröhlich and Michael Franz. On certain basic properties of component-oriented programming languages. In David H. Lorenz and Vugranam C. Sreedhar, editors, *Proceedings of the Workshop on Language Mechanisms for Programming Software Components (at OOPSLA)*, pages 15–18, Tampa Bay, FL, October 15 2001. Technical Report NU-CCS-01-06, College of Computer Science, Northeastern University, Boston, MA 02115. Available at `http://www.ccs.neu.edu/home/lorenz/oopsla2001/`.

[FFK99] Michael Franz, Peter H. Fröhlich, and Thomas Kistler. Towards language support for component-oriented real-time programming. In *Proceedings of the International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS)*, Monterey, CA, November 1999.

[FGF02] Peter H. Fröhlich, Andreas Gal, and Michael Franz. On reconciling objects, components, and efficiency in programming languages. Technical Report 02-12, Department of Information and Computer Science, University of California, Irvine, CA 92697-3425, USA, March 2002. Revised May 2002.

[FK97] Michael Franz and Thomas Kistler. Slim binaries. *Communications of the ACM*, 40(12):87–94, December 1997.

[FKF98] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, pages 173–183, San Diego, CA, January 1998.

[Fla99] Matthew Flatt. *Programming Languages for Reusable Software Components*. PhD thesis, Department of Computer Science, Rice University, Houston, TX, June 1999.

[Fra94] Michael Franz. *Code Generation On-The-Fly: A Key to Portable Software*. PhD thesis, Swiss Federal Institute of Technology (ETH), Zürich, Switzerland, March 1994.

[Fra95] Michael Franz. Protocol extension: A technique for structuring large extensible software-systems. *Software: Concepts and Tools*, 16(2):14–26, July 1995.

[Fra96] Michael Franz. The programming language Lagoona: A fresh look at object-orientation. Technical Report 96-40, Department of Information and Computer Science, University of California, Irvine, CA 92697-3425, USA, September 1996.

[Fra97a] Michael Franz. Dynamic linking of software components. *IEEE Computer*, 30(3):74–81, March 1997.

[Fra97b] Michael Franz. The programming language Lagoona: A fresh look at object-orientation. *Software: Concepts and Tools*, 18(1):14–26, March 1997.

[Frö00] Peter H. Fröhlich. Component-oriented programming languages: Messages vs. methods, modules vs. types. In *Proceedings of the Workshop on Programming Languages and Computer Architecture*, Bad Honnef,

Germany, May 2000. Technical Report 2007, Institute for Computer Science and Applied Mathematics, Christian-Albrechts-University, Kiel, Germany.

[Frö02] Peter H. Fröhlich. Inheritance decomposed. In Andrew Black, Erik Ernst, Peter Grogono, and Markku Sakkinen, editors, *Proceedings of The Inheritance Workshop (at ECOOP)*, Malaga, Spain, June 14, 2002. Technical Report, Information Technology Research Institute (ITRI), University of Jyväskylä, Finland, June 2002.

[GFF02] Andreas Gal, Peter H. Fröhlich, and Michael Franz. An efficient execution model for dynamically reconfigurable component software. In Jan Bosch, Clemens Szyperski, and Wolfgang Weck, editors, *Proceedings of the International Workshop on Component-Oriented Programming (WCOP)*, Malaga, Spain, June 10, 2002. `http://www.research.microsoft.com/~cszypers/Events/WCOP2002/`.

[GJ97] Carlo Ghezzi and Mehdi Jazayeri. *Programming Language Concepts*. John Wiley & Sons, 3rd edition, 1997.

[GJM91] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice-Hall, 1991.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, 2nd edition, 2000.

[GMS77] Charles M. Geschke, James H. Jr. Morris, and Edwin H. Satterthwaite. Early experience with Mesa. *Communications of the ACM*, 20(8):540–553, August 1977.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[Gut77] John Guttag. Abstract data types and the development of data structures. *Communications of the ACM*, 20(6):396–404, June 1977.

[GVJH95] Erich Gamma, John Vlissides, Ralph Johnson, and Richard Helm. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[HHW99] Richard Hall, Dennis Heimbigner, and Alexander L. Wolf. A cooperative approach to support software deployment using the Software Dock. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 174–183, Los Angeles, CA, May 1999.

[HJ89] Charles Antony Richard Hoare and C. B. Jones, editors. *Essays in Computing Science*. Prentice-Hall, 1989.

[Hoa73] Charles Antony Richard Hoare. Hints on programming language design. Technical Report STAN-CS-73-403, Computer Science Department, School of Humanities and Sciences, Stanford University, December 1973. Available at `ftp://reports.stanford.edu/pub/cstr/reports/cs/tr/73/403/CS-TR-73-403.pdf`. Reprinted in [HJ89, pp. 193–216].

[HP96] John L. Hennesey and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman, 2nd edition, 1996.

[HW01] Daniel M. Hoffman and David M. Weiss, editors. *Software Fundamentals: Collected Papers by David L. Parnas*. Addison-Wesley, 2001.

[Int95] International Standards Organization. *(ISO/IEC 8652:1995): Information Technology — Programming Languages — Ada*. 1995.

[IP00] Atsushi Igarashi and Benjamin C. Pierce. On inner classes. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 129–153, Sophia Antipolis and Cannes, France, June 2000.

[Joh86] Ralph E. Johnson. Type-checking Smalltalk. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming,*

Systems, Languages, and Applications (OOPSLA), pages 315–321, Portland, OR, November 1986.

[JW91] Kathleen Jensen and Niklaus Wirth. *Pascal: User Manual and Report*. Springer-Verlag, 4th edition, 1991.

[Kay96] Alan C. Kay. The early history of Smalltalk. In Thomas J. Bergin and Richard G. Gibson, editors, *History of Programming Languages*, pages 511–597. Addison-Wesley / ACM Press, 1996.

[KF00] Thomas Kistler and Michael Franz. Automated data-member layout of heap objects to improve memory-hierarchy performance. *ACM Transactions on Programming Languages and Systems*, 22(3):490–505, May 2000.

[KF01] Thomas Kistler and Michael Franz. Continuous program optimization: Design and evaluation. *IEEE Transactions on Computers*, 50(6):549–566, June 2001.

[KLM+97] Gregor Kiczales, John Lamping, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, June 1997.

[KM98] Jeff Kramer and Jeff Magee. Analysing dynamic change in software architectures: A case study. In *International Conference on Configurable Distributed Systems (CDS)*, pages 91–100, Annapolis, MD, May 1998.

[Knu74] Donald E. Knuth. Structured programming with goto statements. *ACM Computing Surveys*, 6(4):261–301, December 1974. Reprinted in [Knu92].

[Knu88] Jørgen Lindskov Knudsen. Name collision in multiple classification hierarchies. In S. Gjessing and Kirsten Nygaard, editors, *Proceedings of*

the European Conference on Object-Oriented Programming (ECOOP), volume 322 of *Lecture Notes in Computer Science*, pages 93–109. Springer-Verlag, 1988.

[Knu92] Donald E. Knuth. *Literate Programming*. Number 27 in CSLI Lecture Notes. Center for the Study of Language and Information (CSLI), Stanford, CA, 1992.

[LAB+79] Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. CLU Reference Manual. Technical Report MIT/LCS/TR-225, MIT Laboratory for Computer Science, October 1979.

[LBR98] Konstantin Läufer, Gerald Baumgartner, and Vincent F. Russo. Safe structural conformance for Java. Technical Report OSU-CISRC-6/98-TR20, Department of Computer and Information Science, Ohio State University, Columbus, OH 43210-1277, June 1998.

[Lei98] K. Rustan M. Leino. Data groups: Specifying the modification of extended state. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 144–153, Vancouver, British Columbia, October 1998.

[Lev86] Nancy G. Leveson. Software safety: Why, what, and how. *ACM Computing Surveys*, 18(2):125–163, June 1986.

[LG86] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. MIT Press / McGraw-Hill, 1986.

[Lie86] Henry Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 214–223, Portland, OR, November 1986.

[LM98] Gary T. Leavens and Todd D. Millstein. Multiple dispatch as dispatch on tuples. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 374–387, Vancouver, British Columbia, October 1998.

[LS79] Hugh C. Lauer and Edwin H. Satterthwaite. The impact of Mesa on system design. In *Proceedings of the International Conference on Software Engineering (ICSE)*, September 1979.

[LS00] Gary T. Leavens and Murali Sitaraman, editors. *Foundations of Component-Based Systems*. Cambridge University Press, 2000.

[LSAS77] Barbara Liskov, Alan Snyder, Russel Atkinson, and Craig Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, August 1977.

[LvdH02] Chris Lüer and André van der Hoek. Composition environments for deployable software components. Technical Report 02-18, Department of Information and Computer Science, University of California, Irvine, CA 92697-3425, USA, August 2002.

[LW94] Barbara H. Liskov and Jeanette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.

[Mag93] Boris Magnusson. An overview of simula. In Jørgen Lindskov Knudsen, Mats Löfgren, Ole Lehrmann Madsen, and Boris Magnusson, editors, *Object-Oriented Environments: The Mjølner Approach*, chapter 5, pages 79–98. Prentice-Hall, 1993.

[MB97] Michael Mattsson and Jan Bosch. Framework composition: Problems, causes and solutions. In *Proceedings of the Conference on Technology of*

Object-Oriented Languages and Systems (TOOLS), pages 203–214, Santa Barbara, CA, July 1997.

[MC99] Todd Millstein and Craig Chambers. Modular statically typed multimethods. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1628 of *Lecture Notes in Computer Science*, pages 279–303. Springer-Verlag, June 1999.

[McI69] M. Douglas McIlroy. Mass produced software components. In Naur and Randell [NR69], pages 138–155.

[Mey92] Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 2nd edition, 1997.

[Mez97] Mira Mezini. Dynamic object evolution without name collisions. In M. Aksit and S. Matsuoka, editors, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 190–219, June 1997.

[Mic95] Microsoft Corporation. *The Component Object Model (Version 0.9)*, October 1995. Available at `http://www.microsoft.com/COM/resources/COM1598C.ZIP`.

[MMPN93] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the Beta Programming Language*. Addison-Wesley / ACM Press, 1993.

[Moo86] David A. Moon. Object-oriented programming with Flavors. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 1–8, Portland, OR, November 1986.

[MOSS96] Stephan Murer, Stephen Omohundro, David Stoutamire, and Clemens Szyperski. Iteration abstraction in Sather. *ACM Transactions on Programming Languages and Systems*, 18(1):1–15, January 1996.

[MPH00] Peter Müller and Arnd Poetsch-Heffter. Modular specification and verification techniques for object-oriented software components. In *[LS00]*, pages 137–160, 2000.

[MS98] Leonid Mikhajlov and Emil Sekerinski. A study of the fragile base class problem. In Eric Jul, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1445 of *Lecture Notes in Computer Science*, pages 355–382, Brussels, Belgium, July 1998. Springer-Verlag.

[MSL99] Leonid Mikhajlov, Emil Sekerinski, and Linas Laibinis. Developing components in presence of re-entrance. In *Proceedings of the World Congress on Formal Methods in the Development of Computing Systems (FM)*, volume 1709 of *Lecture Notes in Computer Science*, pages 1301–1320, Toulouse, France, September 1999. Springer-Verlag.

[MTG89] Hanspeter Mössenböck, Josef Templ, and Robert Griesemer. Object Oberon: An object-oriented extension of Oberon. Technical Report 109, Institute of Computer Systems, Swiss Federal Institute of Technology (ETH), Zürich, Switzerland, June 1989.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML*. MIT Press, revised edition, 1997.

[MW91] Hanspeter Mössenböck and Niklaus Wirth. The programming language Oberon-2. *Structured Programming*, 12(4):179–195, 1991.

[Nau63] Peter Naur. Revised report on the algorithmic language Algol 60. *Communications of the ACM*, 6(1):1–17, January 1963.

[Nec98] George C. Necula. *Compiling with Proofs*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, September 1998.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[NR69] Peter Naur and Brian Randell, editors. *Software Engineering: Report on a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October, 1968*. Scientific Affairs Division, NATO, Brussels, Belgium, January 1969. Available at `http://www.cs.ncl.ac.uk/people/brian.randell/home.formal/NATO/`.

[Obe97] Oberon microsystems. *Component Pascal Language Definition*, September 1997. Available at `http://www.oberon.ch/`.

[Obj99] Object Management Group. *The Common Object Request Broker: Architecture and Specification (Version 2.3.1)*, October 1999. Available at `http://www.omg.org/cgi-bin/doc?formal/99-10-07`.

[OH92] Harold Ossher and William Harrison. Combination of inheritance hierarchies. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 25–40, Vancouver, British Columbia, Canada, October 1992.

[OMT98] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 177–186, Kyoto, Japan, April 1998.

[Par72] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972. Reprinted in [HW01].

[PC86] David L. Parnas and Paul C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, 12(2):251–257, February 1986. Reprinted in [HW01].

[Per82] Alan J. Perlis. Epigrams in programming. *ACM SIGPLAN Notices*, 17(9):7–13, September 1982.

[PH98] David A. Patterson and John L. Hennesey. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufman, 2nd edition, 1998.

[PJ03] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003. Also available at `http://www.haskell.org/`.

[RR64] Brian Randell and L. J. Russell. *Algol 60 Implementation: The Translation and Use of Algol 60 Programs on a Computer*. Academic Press, London, England, 1964.

[RW92] Martin Reiser and Niklaus Wirth. *Programming in Oberon: Steps Beyond Pascal and Modula*. Addison-Wesley / ACM Press, 1992.

[Sch95] Michael I. Schwartzbach. Polymorphic type inference. Lecture Series LS-95-3, Basic Research in Computer Science (BRICS), Department of Computer Science, University of Aarhus, Ny Munkegade, building 540, DK-8000 Aarhus C, Denmark, June 1995.

[Sew01] Peter Sewell. Modules, abstract types, and distributed versioning. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, pages 236–247, London, England, January 2001.

[SGM02] Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley / ACM Press, 2nd edition, 2002.

[SHF00] Christian H. Stork, Vivek Haldar, and Michael Franz. Generic adaptive syntax-directed compression for mobile code. Technical Report 00-42, Department of Information and Computer Science, University of California, Irvine, CA 92697-3425, USA, November 2000. Revised April 2001.

[SHR+00] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallee-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for Java. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 264–281, Minneapolis, Minnesota, October 2000.

[SL94] Alexander A. Stepanov and Meng Lee. The Standard Template Library. Technical Report X3J16/94-0095, WG21/N0482, Silicon Graphics Inc., 1994.

[Sny86] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 38–45, Portland, OR, November 1986.

[Som02] Ian Sommerville. *Software Engineering*. Addison-Wesley, 6th edition, 2002.

[Ste87] Lynn Andrea Stein. Delegation is inheritance. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 138–146, Orlando, FL, October 1987.

[Str00] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, special edition, 2000.

[SUH86] A. Dain Samples, David Ungar, and Paul Hilfinger. SOAR: Smalltalk without bytecodes. In Norman Meyrowitz, editor, *Proceedings of the*

Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pages 107–118, Portland, OR, November 1986.

[Sun97] Sun Microsystems. *The JavaBeans Specification (Version 1.01)*, July 1997. Available at `http://www.javasoft.com/beans/docs/beans.101.pdf`.

[Szy92] Clemens Szyperski. Import is not inheritance—why we need both: Modules and classes. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 615 of *Lecture Notes in Computer Science*, pages 19–32, Utrecht, The Netherlands, June 1992. Springer-Verlag.

[Szy00] Clemens Szyperski. Modules and components: Rivals or partners? In Laszlo Böszörmeny, Jürg Gutknecht, and Gustav Pomberger, editors, *The School of Niklaus Wirth*. Morgan Kaufman, 2000.

[Tai96] Antero Taivalsaari. On the notion of inheritance. *ACM Computing Surveys*, 28(3):428–479, September 1996.

[Tem94] Josef Templ. *Metaprogramming in Oberon*. PhD thesis, Swiss Federal Institute of Technology (ETH), Zürich, Switzerland, 1994.

[Tho98] Mikkel Thorup. All structured programs have small tree width and good register allocation. *Information and Computation*, 142(2):159–181, May 1998.

[Tic88] Walter F. Tichy. Tools for software configuration management. In *Proceedings of the International Workshop on Software Version and Configuration Control*, pages 1–20, Grassau, Germany, January 1988.

[Tic92] Walter F. Tichy. Programming-in-the-large: Past, present, and future. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 362–367, Melbourne, Australia, 1992.

[US87] David Ungar and Randall B. Smith. SELF: The power of simplicity. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 227–242, Orlando, FL, December 1987.

[VB99] Jan Vitek and Boris Bokowski. Confined types. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 82–96, Denver, CO, November 1999.

[VH94] Jan Vitek and R. Nigel Horspool. Taming message passing: Efficient method look-up for dynamically typed languages. In Mario Tokoro and Remo Pareschi, editors, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 821 of *Lecture Notes in Computer Science*, pages 432–449, Bologna, Italy, July 1994. Springer-Verlag.

[VHK97] Jan Vitek, Nigel R. Horspool, and Andreas Krall. Efficient type inclusion tests. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 142–157, Atlanta, GA, October 1997.

[vR01] Guido van Rossum. *Python Reference Manual Release 2.2*, December 2001. Available at `http://www.python.org/`.

[Wec96] Wolfgang Weck. *On Document-Centered Mathematical Component Software*. PhD thesis, Swiss Federal Institute of Technology (ETH), Zürich, Switzerland, 1996.

[Weg87] Peter Wegner. Dimensions of object-based language design. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 168–182, Orlando, FL, October 1987.

[WG92] Niklaus Wirth and Jürg Gutknecht. *Project Oberon: The Design of an Operating System and Compiler*. Addison-Wesley / ACM Press, 1992.

[Wir77] Niklaus Wirth. Modula: A programming language for modular multiprocessing. *Software: Practice and Experience*, 7(1):3–35, June 1977.

[Wir89] Niklaus Wirth. *Programming in Modula-2*. Springer-Verlag, 4th edition, 1989.

[Wir96] Niklaus Wirth. *Compiler Construction*. Addison-Wesley, 1996.

[WM95] Reinhard Wilhelm and Dieter Maurer. *Compiler Design*. Addison-Wesley, 1995.

[WY88] Takuo Watanabe and Akinori Yonezawa. Reflection in an object-oriented concurrent language. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 306–315, San Diego, CA, September 1988.

[ZCC97] Olivier Zendra, Dominique Colnet, and Suzanne Collin. Efficient dynamic dispatch without virtual function tables: The SmallEiffel compiler. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 125–141, Atlanta, GA, October 1997.