

# **Traits**

## **Composing Classes from Behavioral Building Blocks**

Inauguraldissertation  
der Philosophisch-naturwissenschaftlichen Fakultät  
der Universität Bern

vorgelegt von

**Nathanael Schärli**

von Zell (LU)

Leiter der Arbeit:  
Prof. Dr. Stéphane Ducasse  
Prof. Dr. Oscar Nierstrasz  
Institut für Informatik und angewandte Mathematik



# **Traits**

## **Composing Classes from Behavioral Building Blocks**

Inauguraldissertation  
der Philosophisch-naturwissenschaftlichen Fakultät  
der Universität Bern

vorgelegt von

**Nathanael Schärli**

von Zell (LU)

Leiter der Arbeit:  
Prof. Dr. Stéphane Ducasse  
Prof. Dr. Oscar Nierstrasz  
Institut für Informatik und angewandte Mathematik

Von der Philosophisch-naturwissenschaftlichen Fakultät angenommen.

Bern, 03.02.2005

Der Dekan:  
Prof. P. Messerli



# Abstract

Inheritance is well-known and accepted as a fundamental mechanism for reuse in object-oriented languages. Unfortunately, the main variants — single inheritance, multiple inheritance, and mixin inheritance — all suffer from conceptual and practical problems related to software reuse and robustness with respect to changes. In a first part of this thesis, we identify and illustrate these problems.

To overcome these problems, we then present *traits*, a simple compositional model that extends single inheritance. A trait is essentially a (parameterized) set of methods; it serves as a behavioral building block for classes and is the primitive unit of code reuse. We develop a formal model of traits that establishes how traits can be composed to form other traits or classes, and we describe how we implemented traits in Squeak Smalltalk by bootstrapping a new language kernel.

We present our experimental validation in which we apply traits to refactor parts of the Smalltalk kernel and library, and we develop a programming methodology around the usage of traits and the trait browser, the tool that we implemented to take full advantage of the availability of traits in the Squeak programming environment.

# Acknowledgments

First and foremost, I would like to express my gratitude to my advisors Oscar Nierstrasz and Stéphane Ducasse. Oscar gave me the opportunity to work in his research group and provided me with tremendous professional support and excellent advice throughout my studies. Stéphan's passion and energy are truly contagious. Our countless discussions were invaluable in developing the concepts in this thesis. I also would like to thank Stéphan for introducing me to all of the great technology and people and for educating me in the "French way of life". Oscar and Stéphan, it was a great pleasure to work with you during all these years.

I would also like to thank the other members of my Ph.D. committee, Andrew Black and Alan Kay. Andrew invited me to work with his research group at OGI in Portland. Despite the geographic distance, he provided me with continuous support during my Ph.D. I thank him for the valuable discussions, enjoyable pair programming sessions, and for helping make my papers fit to read. Alan also gave me the opportunity to work with him and his group in California. I especially appreciate that he was willing to share his fantastic knowledge and experience with me as well as many interesting stories from the past. Andrew and Alan, thank you for accepting to be part of my Ph.D. committee; I feel very lucky and honored.

Very special thanks go to my family. I thank my parents, Rosmarie and Toni, and my siblings, Andrea, Matthias, and Petra, for their unending love and support and for the wonderful environment I grew up in. I have had a truly magical time with you and I cannot express how happy it makes me to be part of our family. Most importantly, I thank Rebecca — my love, my best friend and now also my wife — for being here and sharing her life, her love, and her future with me. There is nothing that could make me happier. Thanks also to Rebecca's family for welcoming me into their lives.

I would also like to thank my friends for all the adventures and the good times we have experienced together and for the support and advice you have given me. You know who you are, so I will not list you here. Just let me say that good friends are invaluable and that I feel extremely fortunate to know and be friends with every single one of you. You are the best.

Many thanks also to all the colleagues who have helped me and collaborated with me during my work on traits. I especially thank Gilad Bracha for inspiring me with his dissertation and for kindly sharing with me his knowledge about mixins and other aspects of programming language design. Thanks also to Andreas Raab, who got me thinking about traits in the first place, Adrian Lienhard, who did a lot of work for the stable implementation of traits in Squeak, Philip Quitslund, who developed traits for mini-Java and shared his office with me during my visits to Portland, Stevan Little, who implemented traits in Perl, Matthias Zenger and Martin Odersky, who discussed their implementation of traits in Scala with us, and Jean-Guy Schneider, who reviewed some of our papers on traits.

Next I would like to thank all the former and current members of the Software Composition Group. It was a pleasure to work with you: Franz Achermann, Gabriela Arevalo, Alexandre Bergel, Frank Buchli, Thomas Bühler, Calogero Butera, Juan Carlos Cruz, Marcus Denker, Markus Gälli, Tudor Girba, Orla Greevy, Marc-Philippe Horvath, Andreas Hosbach, Markus Kobel, Stefan Kneubühl, Adrian Kuhn, Michael Locher, Liang Peng, Laura Ponisio, Lukas Renggli, Tamar Richner, Matthias Rieger, Andreas Schlapbach, Therese Schmid, Mauricio Seeberger, Daniele Talerico, Sander Tichelaar, David Vogel, and Roel Wuyts.

I would also like to thank all the people that supported and hosted me during my travels and stays abroad. In particular, I thank the Squeak Group at Disney (now at Viewpoints Research and HP Labs) for making my

internships and visits possible. The time I got to spend with you in California was truly unforgettable. Thanks also to the team at OGI in Portland for making my visits so smooth and pleasant.

Finally, I would like to send my best to all the great people I have met and talked to at conferences, meetings, workshops, online and elsewhere.

*Nathanael Schärli*  
*February 2005*



# Contents

|                                                                |           |
|----------------------------------------------------------------|-----------|
| <b>Abstract</b>                                                | <b>i</b>  |
| <b>1 Introduction</b>                                          | <b>1</b>  |
| 1.1 Understanding the Problem . . . . .                        | 1         |
| 1.2 Our Proposal: Traits . . . . .                             | 2         |
| 1.3 Contributions . . . . .                                    | 3         |
| 1.4 Thesis Outline . . . . .                                   | 3         |
| <b>2 Problems with Inheritance</b>                             | <b>5</b>  |
| 2.1 Composition and Decomposition Problems . . . . .           | 5         |
| 2.1.1 Decomposition Problems . . . . .                         | 5         |
| 2.1.2 Composition problems . . . . .                           | 7         |
| 2.2 Occurrences of these Problems in Real Languages . . . . .  | 9         |
| 2.2.1 Strongtalk and Jam . . . . .                             | 9         |
| 2.2.2 C++ . . . . .                                            | 11        |
| 2.2.3 CLOS . . . . .                                           | 15        |
| <b>3 Traits</b>                                                | <b>16</b> |
| 3.1 Traits – Composable Units of Behavior . . . . .            | 16        |
| 3.1.1 Classes and Methods . . . . .                            | 17        |
| 3.1.2 Traits . . . . .                                         | 19        |
| 3.1.3 Composing Classes from Traits . . . . .                  | 20        |
| 3.1.4 Composite Traits . . . . .                               | 22        |
| 3.1.5 Conflict Resolution . . . . .                            | 23        |
| 3.1.6 Well-definedness . . . . .                               | 27        |
| 3.1.7 Refactoring, reachability and equivalence . . . . .      | 28        |
| 3.2 Evaluation . . . . .                                       | 31        |
| 3.2.1 Decomposition Problems . . . . .                         | 31        |
| 3.2.2 Composition Problems . . . . .                           | 31        |
| 3.3 Discussion . . . . .                                       | 32        |
| 3.3.1 Design Decisions . . . . .                               | 33        |
| 3.3.2 C++ Revisited . . . . .                                  | 35        |
| <b>4 Implementation</b>                                        | <b>39</b> |
| 4.1 Overview of the Smalltalk-80 Kernel Architecture . . . . . | 39        |
| 4.1.1 Organization of Classes and Metaclasses . . . . .        | 39        |
| 4.1.2 The Kernel Classes . . . . .                             | 40        |
| 4.2 Traits Design Rationale . . . . .                          | 42        |
| 4.3 The New Language Kernel . . . . .                          | 42        |

|          |                                                                 |           |
|----------|-----------------------------------------------------------------|-----------|
| 4.3.1    | The New Kernel Classes . . . . .                                | 42        |
| 4.3.2    | Decomposition into Traits . . . . .                             | 46        |
| 4.4      | Flattening Traits at Composition Time . . . . .                 | 46        |
| 4.5      | Conclusions . . . . .                                           | 47        |
| <b>5</b> | <b>Tools and Methodology</b>                                    | <b>48</b> |
| 5.1      | Traits and Tools: Analysis . . . . .                            | 48        |
| 5.2      | The Trait Browser . . . . .                                     | 49        |
| 5.2.1    | Overview . . . . .                                              | 49        |
| 5.2.2    | Virtual Categories . . . . .                                    | 50        |
| 5.2.3    | Using the Browser With Traits . . . . .                         | 52        |
| 5.2.4    | Implementation . . . . .                                        | 54        |
| 5.3      | Programming Methodology . . . . .                               | 59        |
| 5.3.1    | The Roles of Classes and Traits . . . . .                       | 60        |
| 5.3.2    | Uniform Protocols . . . . .                                     | 60        |
| 5.3.3    | Uncovering Hidden Structure . . . . .                           | 64        |
| 5.3.4    | Traits and Agile Methodologies . . . . .                        | 65        |
| 5.4      | Interaction between Language, Tools and Methodology . . . . .   | 66        |
| <b>6</b> | <b>Case Study: Refactoring the Smalltalk Collection Classes</b> | <b>67</b> |
| 6.1      | The Smalltalk Collection Classes . . . . .                      | 67        |
| 6.1.1    | The Varieties of Collection . . . . .                           | 69        |
| 6.1.2    | Streams . . . . .                                               | 71        |
| 6.2      | Analysis of the Collection Classes . . . . .                    | 71        |
| 6.2.1    | Unnecessary Inheritance . . . . .                               | 72        |
| 6.2.2    | Code Duplication and Inappropriate Hierarchies . . . . .        | 73        |
| 6.2.3    | Conceptual Shortcomings . . . . .                               | 74        |
| 6.3      | Refactoring Results . . . . .                                   | 75        |
| 6.3.1    | The New Collection Hierarchy . . . . .                          | 75        |
| 6.3.2    | The New Stream Hierarchy . . . . .                              | 79        |
| 6.3.3    | Measurements of the Refactored Classes . . . . .                | 79        |
| 6.3.4    | Assessment of the Refactored Classes . . . . .                  | 84        |
| 6.3.5    | Design Decisions . . . . .                                      | 85        |
| 6.4      | Discussion . . . . .                                            | 85        |
| 6.4.1    | Lessons Learned . . . . .                                       | 85        |
| 6.4.2    | Comparison to Other Approaches . . . . .                        | 86        |
| 6.5      | Conclusions . . . . .                                           | 88        |
| <b>7</b> | <b>Case Study: Trait-based Composition of Class Properties</b>  | <b>90</b> |
| 7.1      | Motivation and Overview . . . . .                               | 90        |
| 7.2      | Representing Class Properties As Traits . . . . .               | 91        |
| 7.2.1    | Singleton . . . . .                                             | 92        |
| 7.2.2    | Modeling the Boolean Hierarchy . . . . .                        | 93        |
| 7.3      | A Framework of Class Properties . . . . .                       | 95        |
| 7.3.1    | Class Properties . . . . .                                      | 96        |
| 7.3.2    | Advantages for the Programmer . . . . .                         | 98        |
| 7.4      | Evaluation . . . . .                                            | 99        |
| 7.4.1    | Criteria for Class Property Composition . . . . .               | 99        |
| 7.4.2    | Comparison to Other Approaches . . . . .                        | 101       |
| 7.4.3    | Discussion . . . . .                                            | 104       |

---

|          |                                                 |            |
|----------|-------------------------------------------------|------------|
| <b>8</b> | <b>State of the Art</b>                         | <b>106</b> |
| 8.1      | Traits in Other Languages . . . . .             | 106        |
| 8.1.1    | Traits in Perl . . . . .                        | 106        |
| 8.1.2    | Traits in VisualWorks Smalltalk . . . . .       | 108        |
| 8.1.3    | Traits in Scala . . . . .                       | 108        |
| 8.2      | Traits and Static Types . . . . .               | 110        |
| 8.2.1    | Relationship between Traits and Types . . . . . | 110        |
| 8.2.2    | Typing Trait Methods . . . . .                  | 110        |
| 8.2.3    | Conclusions . . . . .                           | 114        |
| 8.3      | Other Related Work . . . . .                    | 114        |
| <b>9</b> | <b>Conclusions</b>                              | <b>119</b> |
| 9.1      | Contributions . . . . .                         | 119        |
| 9.2      | Future Work . . . . .                           | 120        |

# List of Figures

|      |                                                                                                 |    |
|------|-------------------------------------------------------------------------------------------------|----|
| 2.1  | Incorrect attempt to factor out a synchronization wrapper . . . . .                             | 6  |
| 2.2  | Lack of composition control in the composite class MyRectangle . . . . .                        | 8  |
| 2.3  | Customizing the composition using two intermediate “glue mixins”. . . . .                       | 10 |
| 2.4  | The class SyncA in C++ . . . . .                                                                | 12 |
| 2.5  | The class SyncReadWrite implemented with two abstract methods . . . . .                         | 12 |
| 2.6  | Code duplication in the classes SyncA and SyncB . . . . .                                       | 13 |
| 2.7  | Synchronization expressed as a mixin . . . . .                                                  | 14 |
| 2.8  | The class MyDocument built from two mixins . . . . .                                            | 14 |
| 3.1  | An example hierarchy . . . . .                                                                  | 18 |
| 3.2  | The trait TReadStream with provided and required methods . . . . .                              | 21 |
| 3.3  | The class ReadStream composed from the trait TReadStream . . . . .                              | 22 |
| 3.4  | Smalltalk implementation of the class ReadStream and the trait TReadStream . . . . .            | 23 |
| 3.5  | TReadStream and TWriteStream as composite traits . . . . .                                      | 24 |
| 3.6  | Implementation of the composite traits TReadStream and TWriteStream . . . . .                   | 24 |
| 3.7  | Compositions with aliases and conflicts . . . . .                                               | 25 |
| 3.8  | Implementation of the traits TSynchronize and TSyncReadStream . . . . .                         | 26 |
| 3.9  | Code duplication in the Smalltalk Stream hierarchy . . . . .                                    | 28 |
| 3.10 | The Smalltalk Stream hierarchy refactored using traits . . . . .                                | 29 |
| 3.11 | Using C++ templates to simulate mixin composition . . . . .                                     | 35 |
| 3.12 | Using C++ templates and virtual base classes to simulate trait composition . . . . .            | 36 |
| 3.13 | Implementing MyDocument as the composition of two “C++ traits” . . . . .                        | 36 |
| 4.1  | Parallel class and metaclass hierarchies in Smalltalk . . . . .                                 | 40 |
| 4.2  | Class hierarchy of the Smalltalk kernel . . . . .                                               | 41 |
| 4.3  | Classes hierarchy of the new Smalltalk kernel supporting traits . . . . .                       | 43 |
| 4.4  | Decomposition of the new kernel classes into traits . . . . .                                   | 45 |
| 5.1  | The traditional Squeak (Smalltalk) system browser . . . . .                                     | 49 |
| 5.2  | The trait browser . . . . .                                                                     | 51 |
| 5.3  | The trait TCollEnumerationUI . . . . .                                                          | 52 |
| 5.4  | The composite trait TCollEnumerationI . . . . .                                                 | 53 |
| 5.5  | Pseudo-code for collecting the classes that self-send a given selector in a hierarchy . . . . . | 56 |
| 5.6  | Pseudo-code for finding methods that self-send a given selector . . . . .                       | 57 |
| 5.7  | The trait TRectangle . . . . .                                                                  | 61 |
| 5.8  | The new class RectangularMorph immediately after its creation . . . . .                         | 62 |
| 5.9  | The glue method corner in RectangularMorph . . . . .                                            | 63 |
| 5.10 | The method = in RectangularMorph. . . . .                                                       | 64 |

|      |                                                                                           |     |
|------|-------------------------------------------------------------------------------------------|-----|
| 6.1  | The collection classes in Squeak . . . . .                                                | 68  |
| 6.2  | Some collection classes categorized by functionality . . . . .                            | 69  |
| 6.3  | Some collection classes categorized by implementation technique . . . . .                 | 70  |
| 6.4  | Redefinition of at:put: and add: in the superclasses of WeakValueDictionary . . . . .     | 72  |
| 6.5  | Duplicated methods and methods implemented “too high” in the stream hierarchy . . . . .   | 73  |
| 6.6  | The refactored collection hierarchy . . . . .                                             | 76  |
| 6.7  | The hierarchy of traits corresponding to functional properties . . . . .                  | 77  |
| 6.8  | Common implementation traits . . . . .                                                    | 78  |
| 6.9  | Specific implementation traits . . . . .                                                  | 78  |
| 6.10 | The refactored Stream hierarchy. . . . .                                                  | 80  |
| 6.11 | Traits used to build the Stream classes. . . . .                                          | 80  |
| 6.12 | Usage histogram of the functional traits in the Collection classes . . . . .              | 81  |
| 6.13 | Usage histogram of the implementation traits in the Collection classes . . . . .          | 82  |
| 6.14 | Usage histogram of the traits in the Stream classes . . . . .                             | 82  |
| 7.1  | Application of class properties represented as traits . . . . .                           | 91  |
| 7.2  | Behavior and WebServer class built from the traits TInstantiator and TSingleton . . . . . | 92  |
| 7.3  | The metaclass WebServer class implemented using the property trait TSingleton . . . . .   | 93  |
| 7.4  | The refactored Boolean hierarchy . . . . .                                                | 94  |
| 7.5  | Implementation of the property TAbstract . . . . .                                        | 94  |
| 7.6  | A fine-grained architecture of class properties based on traits . . . . .                 | 95  |
| 7.7  | Implementation of the allocation, instantiation, and final class properties . . . . .     | 96  |
| 7.8  | Implementation of different instance creation class properties . . . . .                  | 97  |
| 7.9  | Composing the conflicting properties TDefault and TRememberInstances . . . . .            | 98  |
| 7.10 | Upward and downward compatibility . . . . .                                               | 99  |
| 7.11 | Property composition and property application . . . . .                                   | 100 |
| 7.12 | Smalltalk metaclasses organized in a parallel hierarchy . . . . .                         | 101 |
| 7.13 | By default, a CLOS class and its subclasses are instances of the same metaclass . . . . . | 102 |
| 7.14 | SOM guarantees upward but not downward compatibility. . . . .                             | 102 |
| 7.15 | Class property composition in NeoClasstalk. . . . .                                       | 103 |
| 8.1  | The trait TSyncRead in Perl . . . . .                                                     | 107 |
| 8.2  | The class SyncA defined as a subclass of A using the trait TSyncReadWrite . . . . .       | 107 |
| 8.3  | The trait TLinkable in the dynamically typed language Smalltalk . . . . .                 | 111 |
| 8.4  | The generic trait TLinkable used in the classes Link and Process . . . . .                | 112 |
| 8.5  | The trait TLinkable imlemented using ThisType . . . . .                                   | 113 |

# List of Tables

|     |                                                                                 |     |
|-----|---------------------------------------------------------------------------------|-----|
| 6.1 | Per hierarchy summary of the refactoring . . . . .                              | 83  |
| 6.2 | Summary of the refactoring (totals) . . . . .                                   | 83  |
| 7.1 | Comparison of the different approaches for class property composition . . . . . | 104 |

# Chapter 1

## Introduction

Inheritance is the fundamental reuse mechanism in object-oriented programming languages; it allows the programmer to derive new classes from existing ones [WEGN 88, TAIV 96]. Since single inheritance was first introduced in Simula 67, researchers have developed more sophisticated variants of inheritance, such as various forms of multiple inheritance and mixin inheritance, and millions of programmers have learned to build complex frameworks and applications using classes organized in inheritance hierarchies.

Despite this story of success, the position of inheritance-based reuse in current object-oriented programming languages is unsatisfying. While single inheritance is not expressive enough to factor out common features in a more complex scenario, neither multiple inheritance nor mixins have achieved wide acceptance [TAIV 96]. Summarizing Alan Snyder’s contribution to the inheritance panel discussion at OOPSLA ’87, Steve Cook wrote:

“Multiple inheritance is good, but there is no good way to do it.” [COOK 87]

Indeed, the trend seems to be away from multiple inheritance; the designers of recent languages such as Java and C# decided that the complexities and problems introduced by multiple inheritance outweighed its utility. For the programmer this means that the golden principle “say everything once and only once” remains an ideal that cannot be reached in reality: even with a perfect design, it is often not possible to avoid code duplication.

In this dissertation, we address this dilemma by introducing traits, a simple compositional model that allows one to build classes from multiple parts while avoiding most of the complexity and problems known from multiple inheritance and mixins. Before we give a brief overview of traits and present our thesis statement (Section 1.2), we summarize the problems associated with the different variants of inheritance (Section 1.1). We end this introductory chapter with a summary of the contributions (Section 1.3) and an outline of the rest of this dissertation (Section 1.4).

### 1.1 Understanding the Problem

Single inheritance is the classic and most accepted form of inheritance. While it is conceptually simple and well understood, it is inadequate for expressing classes that share features not inherited from their (unique) common parent. The shared features must either be forced into the common parent (where they do not belong), or they must be duplicated in the classes that should share them. To overcome this limitation, language designers have proposed various forms of multiple inheritance [MEYE 88, KEEN 89, STRO 86], as well as other mechanisms, such as mixins [MOON 86, BRAC 90, MENS 96, FLAT 98, ANCO 00], that allow classes to be composed incrementally from sets of features.

Despite the passage of nearly twenty years, neither multiple inheritance nor mixins have achieved wide acceptance [TAIV 96]. In case of multiple inheritance, the intuitive explanation for this situation has often been

that it is too complex and leads to code that is hard to understand. A more detailed assessment shows that multiple inheritance not only poses serious implementation problems [DIXO 89, SWEE 99], it also introduces serious conceptual problems.

Although multiple inheritance makes it possible to reuse any desired set of classes, a class is frequently not the most appropriate element to reuse. This is because classes play two competing roles. A class is first and foremost a *generator of instances*. Therefore, most of the recent object-oriented programming languages such as Java and C# make every class bundle together a *complete* set of basic features by requiring it to be a (direct or indirect) subclass of the class `Object`, which is the root of the inheritance hierarchy. A class doubles, however, as a *unit of reuse*. It should therefore bundle a *minimal* set of features which can sensibly be reused together.

Unfortunately these two roles conflict. Since classes must adopt a fixed position in the class hierarchy (i) it can be difficult or impossible to factor out wrapper methods as reusable classes, (ii) conflicting features inherited from different paths may cause semantic ambiguities (e.g., the “diamond problem” [SAKK 89, BRAC 90]), and (iii) overridden features may be difficult to access.

Moon’s Flavors [MOON 86] were an early attempt to address these problems: Flavors are small, incomplete implementations of classes, that can be “mixed in” at arbitrary places in the class hierarchy. More sophisticated notions of mixins were subsequently developed by Bracha and Cook [BRAC 90], Mens and van Limberghen [MENS 96], Flatt, Krishnamurthi and Felleisen [FLAT 98], and Ancona, Lagorio and Zucca [ANCO 00].

Mixins use the ordinary single inheritance operator to extend various parent classes with the same set of features. Although this inheritance operator is well-suited for deriving new classes from existing ones, it is not necessarily appropriate for composing reusable building blocks. Specifically, as mixin composition is implemented using inheritance, mixins are composed linearly. This gives rise to several problems. First, a suitable total ordering of features may be difficult to find, or may not even exist. Second, “glue code” that exploits or adapts the linear composition may be dispersed throughout the class hierarchy. Third, the resulting class hierarchies are often fragile with respect to change, so that conceptually simple changes may impact many parts of the hierarchy. It is for these reasons, we believe that mixins have never achieved wide success in mainstream object-oriented languages.

## 1.2 Our Proposal: Traits

Traits represent a simple solution to these various dilemmas. In a nutshell, traits are (parameterized) *sets of methods* that serve as a behavioral building blocks of classes. Traits can be composed in arbitrary order as trait composition is commutative and completely divorced from the inheritance hierarchy. Classes are organized in a single inheritance hierarchy, and can make use of traits to specify the incremental difference in behavior with respect to their superclasses. Traits have the following characteristics:

- Traits are simple software components that both *provide* and *require* methods (i.e., those that are used by, but not implemented in a trait).
- Classes (and composite traits) are composed from traits, resolving any conflicts, and possibly providing the required methods.
- Traits specify no state, so the only conflict that can arise when combining traits is a method conflict. Such a conflict can be resolved by exclusion or overriding.
- Traits can be inlined or “flattened”, so their presence or absence does not affect the semantics of a class.

**Thesis statement.** The design of traits was guided by the attempt to unify the advantages of multiple inheritance and mixin inheritance while avoiding their disadvantages. As a consequence, traits provide a better solution to the reuse problems in object-oriented languages than multiple inheritance and mixins.

Most difficulties experienced with multiple inheritance disappear with traits, because traits are purely behavioral and are (like mixins) divorced from the inheritance hierarchy, which makes it easy to factor out wrapper methods that can be applied to classes across the inheritance hierarchy and avoids most of the complexity related to conflicting features. At the same time, most difficulties experienced with mixins also disappear, because traits impose (like many multiple inheritance mechanisms) no linear composition order. As a consequence, compositions based on traits are more robust with respect to changes, and the composite entity has complete control over the composition and can resolve all conflicts explicitly, without use of linearization.

Unlike the semantics of single inheritance, multiple inheritance, and mixins, the semantics of trait composition adheres to the flattening property, which states that traits can be inlined (flattened) without changing the semantics of the class that uses them, and with no change to any of the methods in the traits. This allows the programmer to view and edit a class built from traits as if it were built using single inheritance alone. Especially if they are fine-grained, compositions based on traits are therefore easier to understand than corresponding hierarchies based on multiple inheritance or mixins. The flattening property, together with the fact that traits are purely behavioral, also allows for an efficient implementation while avoiding many of the difficulties known from multiple inheritance and mixin implementations.

## 1.3 Contributions

The main contributions of this thesis can be summarized as follows:

- A detailed illustration and analysis of the various composition and decomposition problems that are associated with multiple inheritance and mixins.
- The development of traits, a composition mechanism that was specifically designed to address these inheritance problems by allowing classes to be composed from behavioral building blocks.
- A formal model of traits.
- A description of the implementation of traits in the Smalltalk dialect Squeak [INGA 97], which we accomplished by bootstrapping a new language kernel using traits.
- The development of a programming methodology around the usage of traits and the trait browser, a tool that we implemented to take full advantage of the availability of traits in the Squeak programming environment.
- Two case studies, a refactoring of the Smalltalk collection classes and a trait-based approach to overcome metaclass composition problems, which demonstrate the practical usefulness and applicability of traits.
- A comparison between traits as we have developed them and adaptations of traits designed for other programming languages, and an overview of conceptual problems and possible solutions when applying traits to statically typed languages.

## 1.4 Thesis Outline

This dissertation is structured as follows: In Chapter 2, we identify and illustrate the composition and decomposition problems that are associated with different forms of inheritance. In Chapter 3, we then introduce traits by means of a formal model, diagrams, and running examples, and we evaluate the traits mechanism against the previously identified problems. In Chapter 4, we describe our implementation of traits in the Smalltalk dialect Squeak. In Chapters 6 and 7, we present two case studies that illustrate how traits and their theoretic properties pay off in realistic scenarios: a refactoring of the Smalltalk collection classes (Chapter 6) and a trait-based approach to overcome metaclass composition problems (Chapter 7). In Chapter 8, we give an overview of the trait

adaptations available in other programming languages, we analyze what it means to apply traits to statically typed languages, and we compare traits to other related work. In Chapter 9, we conclude by summarizing how the main results of our work support the statement of the thesis, and we look forward to future work related to traits.

## Chapter 2

# Problems with Inheritance

While inheritance is widely considered to be one of the key features of object-oriented programming, it is also saddled with many competing and contradictory definitions and interpretations [TAIV 96]. Over the years, researchers have developed various forms of inheritance, including single inheritance, multiple inheritance, and mixin inheritance. Each of these forms provides different answers to problems of *decomposition* — how one decomposes a software base into suitable units of reuse — and *composition*— how one composes these units to obtain a class hierarchy suitable for our application domain.

In this chapter, we analyze and illustrate how the key problems of decomposition and composition apply to the different forms of inheritance. While Section 2.1 is kept on a more conceptual and general level, Section 2.2 focuses on occurrences of these problems in concrete programming languages.

Note that we focus on conceptual issues related to reuse. Other problems with inheritance such as implementation difficulties [DIXO 89, SWEE 99] and conflicts between inheritance and subtyping [AMER 90, BRUC 95, CAST 95, COOK 90, MADS 90, LALO 91] are outside the scope of this thesis.

## 2.1 Composition and Decomposition Problems

In this section, we give an overview of the key problems of decomposition and composition that occur in the different forms of inheritance. For conciseness, this overview is kept on a conceptual and rather general level; details about the occurrence of these problems in concrete programming languages are deferred to Section 2.2.

### 2.1.1 Decomposition Problems

Object-oriented programming allows the programmer to model arbitrary domains as hierarchies of classes that are related by inheritance. However, the different forms of inheritance pose different restrictions and limitations on what kind of decompositions can be expressed. As a result, the way in which we decompose our domain concepts into classes is not necessarily the right way to decompose the implementations of these classes into sets of features [LALO 89, HARR 93, TARR 99]. We will now consider three decomposition problems that we have identified in this context.

#### Duplicated Features

Single inheritance is the simplest form of inheritance; it allows a class to inherit from (at most) one superclass [COOK 89]. Although well-accepted, single inheritance is not expressive enough to factor out all the common features shared by classes in a complex hierarchy. As a consequence, single inheritance does not provide the means to avoid code duplication.

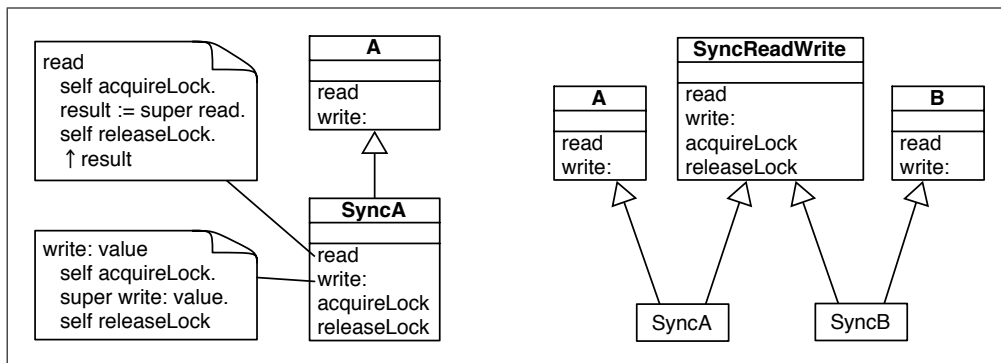


Figure 2.1: Incorrect attempt to factor out a synchronization wrapper

As an example, consider the Smalltalk stream classes `ReadStream`, `WriteStream`, and `ReadStream`. As suggested by their names, the class `ReadStream` contains features provided by both `ReadStream` and `WriteStream`. However, with single inheritance, `ReadStream` can inherit from only one of these classes. In Smalltalk, `ReadStream` inherits from `WriteStream` and then duplicates some `ReadStream` methods (cf. Figure 6.5).

Note that extension of single inheritance with interfaces, as promoted by Java and C#, addresses the issues of subtyping and conceptual modeling, but does not provide any help with the problem of duplicated code.

### Inappropriate Hierarchies

A common practice to avoid such code duplication is to implement certain methods “too high” in the hierarchy. The idea is that instead of duplicating a method, it is moved to a superclass until it is available in all the classes where it is actually required. In our example, this means that the programmer could implement all the reading methods in the class `PositionableStream`, which is the lowest common superclass of `ReadStream` and `WriteStream`. These methods would then be inherited by the class `ReadStream`, and hence, would not need to be duplicated.

The tactic succeeds, but the price is high: `PositionableStream` is polluted by many methods that have nothing to do with positioning, and `WriteStream` appears to implement many reading methods, although these methods will fail or result in inconsistent behavior if they are ever used. (See Section 6.2.2 for a more detailed discussion of the problems that occur in the Smalltalk stream hierarchy.)

Both multiple inheritance and mixins attempt to alleviate these problems by allowing a class to obtain features from multiple sources, but, as we shall see, each gives rise to other problems.

### Duplicated Wrappers

Multiple inheritance as provided by languages such as C++ and Eiffel enables a class to reuse features from multiple parent classes, but it does not allow one to write a reusable entity that extends methods implemented in as-yet unknown classes.

This limitation is illustrated in the UML class diagrams in Figure 2.1. Assume that class `A` contains methods `read` and `write`: that provide unsynchronized access to some data. (If not otherwise indicated, all the code in this thesis is written in Smalltalk. The traits model, however, is not specific to Smalltalk.) If it becomes necessary to synchronize the access to this data, we can create a class `SyncA` that inherits from `A` and wraps the methods `read` and `write`:. That is, `SyncA` defines new `read` and `write`: methods that call the inherited methods under control of a lock (cf. Figure 2.1 left).

Now suppose that class `A` is part of a framework that also contains another class `B` with `read` and `write`: methods, and that we want to use the same technique to create a synchronized version of `B`. Naturally, we

would like to factor out the synchronization code so that it can be reused in both SyncA and SyncB.

With multiple inheritance, sharing code among different classes means inheriting from a common superclass. This means that we should move the synchronization code into a class SyncReadWrite that will become the superclass of both SyncA and SyncB (cf. Figure 2.1 right). Unfortunately this does not work because super-sends are statically resolved in most forms of multiple inheritance such as those of C++ and Eiffel. Therefore, the super-sends in methods of SyncReadWrite would refer *statically* to methods of its superclass, and not to methods in A or B.

Workarounds are clumsy and just entail more duplicated code, for example, the super-sends in SyncReadWrite could be replaced by calls to abstract methods myRead and myWrite:, which are then implemented in both SyncA and SyncB to call, respectively, the read and write: methods of A and B. (See Section 2.2.2 for more details.)

Mixins solve this particular problem by late-binding super. A mixin is an abstract subclass specification that may be applied to various parent classes to extend them with the same set of features [MOON 86, BRAC 90, MENS 96, FLAT 98]. Instead of defining SyncReadWrite as a class, it is defined as a mixin. Then SyncA and SyncB will each apply the mixin to a different superclass, and obtain the desired wrapper behavior.

### 2.1.2 Composition problems

Although there is a clear progression in expressive power from single inheritance through multiple inheritance to mixins, this expressiveness does not come without a cost. Both multiple inheritance and mixins pose numerous problems when we consider how classes are composed from shared features.

#### Conflicting Features

One of the problems with multiple inheritance is the ambiguity that arises when conflicting features are inherited along different paths [DUGG 01]. A particularly problematic situation is the “diamond problem” [BRAC 90] (called “fork-join inheritance” by Sakkinen [SAKK 89]), which occurs when a class inherits from the same parent class via multiple paths. Very often, the root of a class hierarchy (or a sub-hierarchy) consists of a class that provides some common default behavior that may be overridden by subclasses (e.g., methods =, hash, and asString). However, this is precisely what causes the conflicts when several of these classes are reused.

The features that conflict may be methods or attributes. Whereas method conflicts can be resolved relatively easily (e.g., by overriding), conflicting attributes are more problematic. Even if the declarations are consistent, it is not clear whether conflicting attributes should be inherited once or multiply [MEYE 88, SAKK 92]; neither is it clear how these attributes should be initialized.

Single inheritance clearly does not suffer from this problem; nor do mixins that are based on single inheritance. With mixin composition, a mixin is applied to a class to generate a new subclass in a single inheritance hierarchy. Conflicts do not arise because the mixin features simply extend or override those of the class it is applied to. However, mixins must be applied one at a time in a particular order, which leads to other problems, as we shall shortly see.

#### Lack of Control and Dispersal of Glue Code

Mixin composition is linear: all the mixins used by a class must be inherited one at a time. Features defined in mixins appearing later in the order override *all* the identically named features of earlier mixins. Where conflicts should be resolved by selecting and combining features from different mixins, a suitable total order may not exist. So, while avoiding the problem of ambiguous conflicts, mixins put feature composition into a straitjacket from which it may be difficult to escape.

As a consequence, with mixins, the composite entity is not in full control of the way in which the mixins are composed: instead, the way in which the individual features override and extend one another is dictated by the total ordering imposed on the mixins. Obtaining the desired combination of features may necessitate the

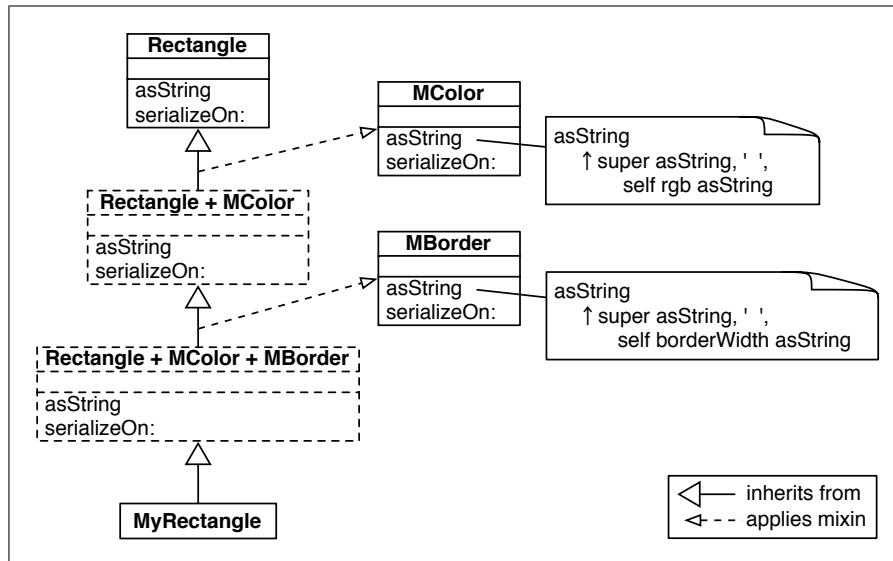


Figure 2.2: Lack of composition control in the composite class MyRectangle

introduction of glue code in new intermediate mixins, or even the modification of the used mixins. While modifying mixins is very problematic because it potentially breaks other classes using the same mixins, introducing intermediate mixins causes the glue code to be scattered throughout the inheritance hierarchy, which makes the composition hard to understand and adapt.

As an example, consider the class diagram shown in Figure 2.2, where a class `MyRectangle` uses two mixins `MColor` and `MBorder` that each provide method `asString` and `serializeOn`. (We introduce an *ad hoc* extension to UML to show where mixins are applied in the inheritance hierarchy, and we use class names such as `Rectangle + MColor` for the anonymous intermediate classes that result when mixins are applied.) The implementations of the method `asString` in the mixins first call the implementation inherited from the superclass (using the keyword `super`) and then extend the resulting string with a separation character followed by some specific information about their own state. Similarly (but not shown in Figure 2.2), the implementations of `serializeOn` in the mixins first call the superclass method and then append the mixin's own state to the argument stream.

Suppose now that for compatibility reasons, we need to serialize our class `MyRectangle` so that the `rgb` value appears before the `borderWidth`. Because mixin composition is linear, this means that the mixin `MColor` has to be applied before the mixin `MBorder`. However, because of the total ordering of *all* features provided by the mixins, this also means that it changes the order of the `asString` methods and that the color attributes are then printed before the border attributes, which may not be what we want.

The crux of the problem is that from within the composite entity `MyRectangle`, it is not possible to control separately how the different features are composed. This is because in `MyRectangle`, we can only access the *mixed* behavior of `Rectangle + MColor + MBorder`, but not the original behavior of `MColor` and `Rectangle`.

Thus, if we need to customize how the features are composed — be it because we need a different serialization order or a another separation character between the two strings — we need to modify the involved mixins, which is problematic as it potentially breaks all the other clients of these mixins. (See Section 2.2.1 for more details.)

Note that *composite mixins* [BRAC 92] do not provide any substantial advantage over regular mixins with respect to limited control over a composition. Like mixins, composite mixins only provide a linear form of composition where all the features of the involved mixins are totally ordered. This means that in the above example, the exact same problem occurs if we were to combine the two mixins `MColor` and `MBorder` to create a composite mixin `MColorAndBorder` and then use this composite mixin to define our new class `MyRectangle`.

The only difference is that now the problem manifests itself during the definition of the composite mixin rather than the definition of the class `MyRectangle`.

### Fragile Hierarchies

The total ordering of mixins can lead to a further problem, namely that the resulting inheritance hierarchies will often be fragile with respect to changes. Adding a new method to one of the mixins *implicitly* overrides all identically named methods of mixins that appear earlier in the superclass chain. It may furthermore be impossible to reestablish the original behavior of the composite without having to add or change several mixins in the chain. This is especially critical if one modifies a mixin that is used in many places across the class hierarchy.

As an illustration, suppose that in the previous example the mixin `MBorder` does not initially define a method `asString`. This means that the implementation of `asString` in `MyRectangle` will be the one specified by `MColor`. Suppose that, later, the method `asString` is added to the mixin `MBorder`. Because of the total ordering of mixins, this implicitly overrides the implementation provided by `MColor`. Worse, the original behavior of the composite class `MyRectangle` cannot be reestablished without changing more of the mixins involved in the composition. If we want to avoid the ripple effect caused by further changes to existing mixins, we have to introduce a new “glue mixin” between the mixins `MColor` and `MBorder`, which makes the method `asString` provided by `MColor` available under a new name such as `colorAsString`, and then add another glue method `asString` to the class `MyRectangle`.

With many forms of multiple inheritance, we also observe a fragility problem with respect to changes. Since identically named features can be inherited from different parent classes, a single keyword (*e.g.*, `super`) is not enough to access inherited methods unambiguously. For example, C++ [STRO 97] forces one to explicitly name the base class to access an overridden method; recent versions of Eiffel [MEYE 97] adopt an analogous technique<sup>1</sup>. Although explicitly naming the source of overridden features makes it possible to compose features from multiple parent classes, embedding explicit class references in the source code makes the code fragile with respect to changes in the class hierarchy.

## 2.2 Occurrences of these Problems in Real Languages

Since inheritance is the primary composition and decomposition mechanism in object-oriented programming languages, it is not surprising that occurrences of these problems can be found, in one flavor or another, in most of today’s object-oriented languages. In this section, we identify and illustrate such occurrences in some of the most interesting languages with regard to their inheritance features: Strongtalk, Jam, C++, and CLOS.

### 2.2.1 Strongtalk and Jam

Strongtalk [BAK 02] and Jam [ANCO 00] are respectively extensions of Smalltalk and Java with mixins. Both suffer from the limitations caused by the total ordering imposed by mixin composition. As an illustration, consider the situation shown in Figure 2.2, and suppose again that for compatibility reasons, we need to serialize our class `MyRectangle` so that the `rgb` value appears on the stream before the `borderWidth`, which means that the mixin `MColor` has to be applied before the mixin `MBorder`.

As pointed out before, this also means that the color attributes get printed before the border attributes and that it is not possible to change this within the composite entity `MyRectangle`. Instead, getting the reversed printing order without duplicating any code requires modifying the involved mixins.

<sup>1</sup>The ability to access an overridden method using the keyword `Precursor` followed by an optional superclass name was added to Eiffel in 1997 [MEYE 97]. In earlier versions of Eiffel, access to original methods required repeated inheritance of the same class [MEYE 92]

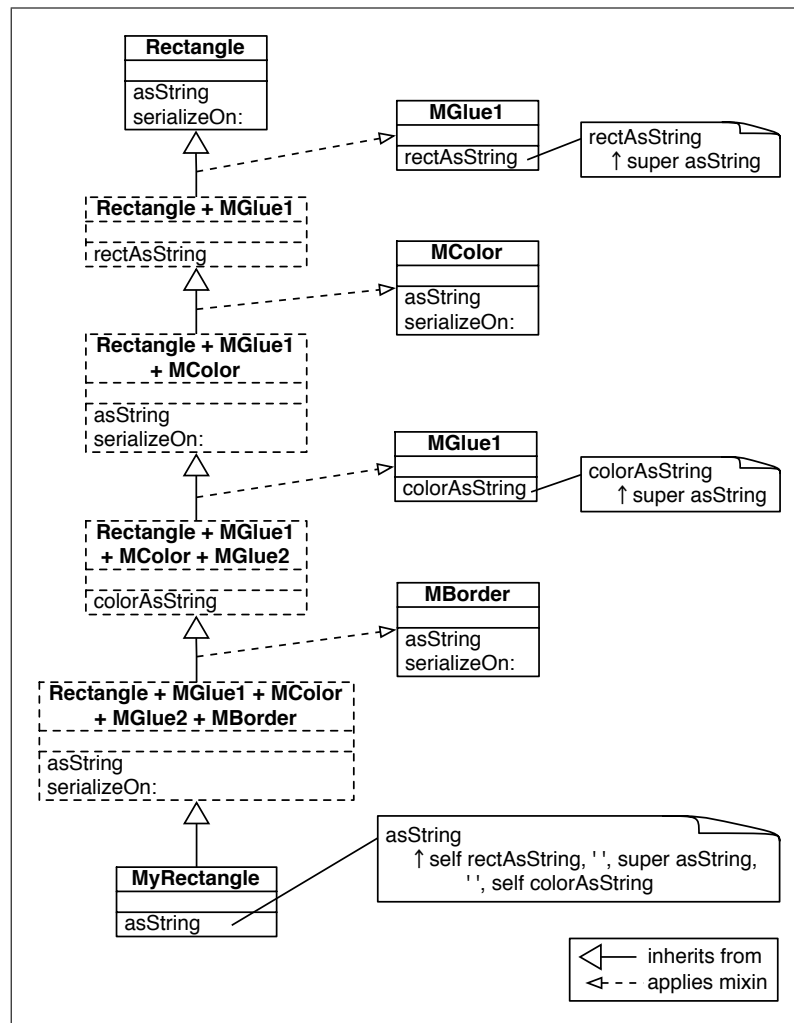


Figure 2.3: Customizing the composition using two intermediate “glue mixins”.

Modifying these mixins could be avoided if they were written so that the method `asString` included only the mixin-specific printing behavior and not the part referring to the superclass implementation. In the Smalltalk-based language Strongtalk, the corresponding code would look as follows:

```

MColor>>asString
  ↑ self color asString

MBorder>>asString
  ↑ self borderWidth asString
  
```

However, this does not solve the problem of inappropriate overrides caused by the linear application of mixins. In fact, the methods `asString` provided by `MColor` and `Rectangle` are overridden by the implementation of `MBorder`, and therefore, they are still not accessible from within the composite class `MyRectangle`. Hence, this class cannot specify the necessary glue code that combines the three `asString` methods in the desired order.

To fix this, we first need to interpose two additional “glue mixins” `MGlue1` and `MGlue2` into the hierarchy, as shown in Figure 2.3. `MGlue1` is inserted between the class `Rectangle` and the mixin `MColor`, and it implements the method

```
MGlue1>>rectAsString
  ↑ super asString
```

to make the method `Rectangle>>asString` available under the name `rectAsString`. Similarly, `MGlue2` is inserted between the mixins `MColor` and `MBorder`, and it implements the method

```
MGlue2>>colorAsString
  ↑ super asString
```

to make the method `MColor>>asString` available under the name `colorAsString`. Only now it is possible to write the necessary glue method `asString` in the combining class `MyRectangle` so that we obtain the desired printing order:

```
MyRectangle>>asString
  ↑ self rectAsString, ' ', super asString, ' ', self colorAsString.
```

The problem with this approach is that interposing the two additional glue mixins makes the inheritance hierarchy more complex and causes the glue code to be scattered over three different entities. This makes program comprehension much more difficult. Even a programmer who is familiar with the class `Rectangle` as well as the mixins `MColor` and `MBorder` will have to look at three different entities to understand the composition of the mixins. And because the occurrence of `super-sends` makes the code in the mixins `MGlue1` and `MGlue2` dependent on their position in the inheritance hierarchy, understanding the composite class `MyRectangle` requires analyzing the inter-dependencies of the complete hierarchy consisting of six entities.

Note that the even though the examples are presented in Strongtalk syntax, JAM suffers from exactly the same problems as it facilitates the same kind of mixin mechanism.

### 2.2.2 C++

The language C++ [STRO 97] is quite unique for our perspective: it features native support for multiple inheritance and also supports mixins by means of classes with a parameterized superclass (*i.e.*, templates).

#### Multiple inheritance

A distinctive feature of multiple inheritance in C++ is that the programmer has a certain amount of control over a diamond situation by declaring a base class as *virtual*, which means that the base class is shared and its attributes are inherited only once<sup>2</sup>. While this provides help for avoiding conflicts and ambiguities in a diamond situation, it does not help to solve the problem of duplicated wrappers.

As an illustration, let us reconsider the example shown in Figure 2.1, where a class `A` implements two methods for reading and writing and has a subclass `SyncA` that implements synchronized versions of these methods. Figure 2.4 shows the implementation of `SyncA` in C++.

With multiple inheritance, sharing code among different classes means (directly or indirectly) inheriting from a common superclass that contains the code to be shared. Therefore, if we want to share the synchronization code in `SyncA` to create another synchronized subclass `SyncB` of `B`, we need to factor this code into a new class `SyncReadWrite` and then make it the superclass of both `SyncA` and `SyncB` (cf. Figure 2.1 right).

Unfortunately, multiple inheritance alone is not expressive enough to do this. The problem is that the calls to the superclass versions of `read` and `write` are statically bound and can refer only to a *superclass* of `SyncReadWrite`. Therefore, the class `SyncReadWrite` cannot explicitly call the unsynchronized versions of the methods `read` and `write` provided by its *subclasses* `A` and `B`.

As a workaround, one would have to modify the methods `read` and `write` in `SyncReadWrite` so that the explicit calls to the superclass methods are replaced by calls to abstract methods `directRead` and `directWrite`

<sup>2</sup>In his description of C++ [STRO 97], Stroustrup uses the term “mixin” for classes overriding methods of a virtual base class. However, this definition differs from the meaning of the term “mixin” used in this thesis and in most of the research literature.

```
class SyncA : public A {  
    public:  
        virtual int read() {  
            acquireLock();  
            result = A::read();  
            releaseLock();  
            return result;  
        };  
        virtual void write(int n) {  
            acquireLock();  
            A::write(n);  
            releaseLock();  
        };  
  
    protected:  
        virtual void acquireLock() {  
            // acquire lock  
        };  
        virtual void releaseLock() {  
            // release lock  
        };  
};
```

Figure 2.4: The class SyncA in C++

```
class SyncReadWrite {  
    public:  
        virtual int read() {  
            acquireLock();  
            result = directRead();  
            releaseLock();  
            return result;  
        };  
        virtual void write(int n) {  
            acquireLock();  
            directWrite(n);  
            releaseLock();  
        };  
  
    protected:  
        virtual void acquireLock() {  
            // acquire lock  
        };  
        virtual void releaseLock() {  
            // release lock  
        };  
  
        virtual int directRead() = 0;  
        virtual void directWrite(int n) = 0;  
};
```

Figure 2.5: The class SyncReadWrite implemented with two abstract methods

|                                                                                                                                                                                                                                                                                                                                                         |                                                                                                                                                                                                                                                                                                                                                      |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> class SyncA : public A, SyncReadWrite { public:     virtual int read() {         return SyncReadWrite::read();     };     virtual void write(int n) {         SyncReadWrite::write(n);     };  protected:     virtual int directRead() {         return A::read();     };     virtual void directWrite(n) {         A::write(n);     }; }; </pre> | <pre> class SyncB : public A, SyncReadWrite { public:     virtual int read()         return SyncReadWrite::read();     };     virtual void write(int n) {         SyncReadWrite::write(n);     };  protected:     virtual int directRead() {         return B::read();     };     virtual int directWrite(n) {         B::write(n);     }; }; </pre> |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

Figure 2.6: Code duplication in the classes SyncA and SyncB

(Figure 2.5) that will then be implemented by the subclasses SyncA and SyncB (Figure 2.6). However, this solution is not optimal as it still requires duplication of four glue methods in each subclass. Furthermore, avoiding name clashes between the synchronized and unsynchronized versions of the read and write methods makes this approach rather clumsy, and one has to make sure that the unsynchronized methods `directRead` are not publicly available in SyncA and SyncB.

### Template-based mixins

Unlike the generics mechanisms of most other languages such as Java and C#, the C++ template mechanism allows the programmer to write classes with generic superclasses. As shown by VanHilst and Notkin [VANH 96a, VANH 96b] as well as Smaragdakis and Batory [SMAR 98, SMAR 00], this allows the programmer to express a mixin in C++ as a class with a generic superclass. Thus, in C++, the programmer can avoid the limitation of multiple inheritance with regard to wrappers by using mixins instead. In the previous example, this means that the synchronization code can be written as a generic class `MSyncReadWrite`. This generic class can then be used to create the classes SyncA and SyncB by applying it to the superclasses A and B, respectively. The corresponding code is shown in Figure 2.7.

Apart from the fact that C++ mixins are explicitly written as generic classes, this approach is identical to ordinary mixins as discussed earlier. Therefore, it is not surprising that it solves our problem without any code duplication. However, it also means that this approach suffers from the linearization problems pointed out before as soon as multiple mixins are composed.

As an example, assume that we want to combine the mixin `MSyncReadWrite` with another wrapper mixin `MLogOpenClose` to create a new class `MyDocument`, which differs from its superclass `Document` in that it synchronizes all the calls to the methods `read` and `write`, and logs all the calls to the methods `open` and `close`. Unfortunately, this again requires the programmer to choose a total order for the two mixins. In the code shown in Figure 2.8, we decided that the mixin `MSyncReadWrite` is applied last, which means that it overrides all the features of the other mixin `MLogOpenClose`. This is not a problem as long as the two mixins do not implement conflicting features. However, it makes the whole hierarchy fragile with respect to changes: if the mixin `MSyncReadWrite` is changed so that it also provides a method `reset`, this new method *implicitly* overrides the implementation provided by `MLogOpenClose` and hence breaks our class `MyDocument`.

As illustrated in Section 2.2.1, the programmer can fix such conflicts by modifying existing mixins, which is problematic if the changed mixins are used elsewhere, or by introducing new intermediate mixins, which leads to dispersal of glue code. In addition, C++ offers a third option for resolving such conflicts. This option

```

template <class Super>
class MSyncReadWrite : public Super {
    public:
        virtual int read() {
            acquireLock();
            result = Super::read();
            releaseLock();
            return result;
        };
        virtual void write(int n) {
            acquireLock();
            Super::Write(n);
            releaseLock();
        };

        protected:
        virtual void acquireLock() {
            // acquire lock
        };
        virtual void releaseLock() {
            // release lock
        };
};

class SyncA : public MSyncReadWrite<A> {};
class SyncB : public MSyncReadWrite<B> {};

```

Figure 2.7: Synchronization expressed as a mixin

```

template <class Super>
class MLogOpenClose : public Super {
    public:
        virtual void open() {
            Super::open();
            log("Opened");
        };
        virtual void close() {
            Super::close();
            log("Closed");
        };
        virtual void reset() {
            // reset logger
        };

        protected:
        virtual void log(char* s) {
            // write to log
        };
};

class MyDocument : public MSyncReadWrite<MLogOpenClose<Document>> {};

```

Figure 2.8: The class MyDocument built from two mixins

is based on the fact that C++ allows a class to explicitly access features of its indirect superclasses by using nested scope qualifiers.

In the previous example, this means that the class `MyDocument` can use the expression

```
MSyncReadWrite::MLogOpenClose::reset()
```

to refer to the method `reset` in `MLogOpenClose`, while it can use the expression

```
MSyncReadWrite::MLogOpenClose::Document::reset()
```

to refer to the method `open` in `Document`.

Using such nested scope qualifiers, the programmer can write the glue code directly in the composite class `MyDocument`. This avoids the dispersal of glue code caused by introducing intermediate mixins, but it introduces other problems. Besides the fact that nested scope qualifiers make the code hard to read, understand and maintain, they also make the code fragile with respect to changes in the hierarchy and can break encapsulation [SNYD 86]. This is because a class using such code does not only depend on the entirety of features inherited from its direct superclass, but can instead have explicit dependencies on the complete inheritance hierarchy (*e.g.*, the exact order of the applied mixins) and the implementation details of all its *indirect* superclasses (*e.g.*, whether a superclass implements or inherits a certain feature).

### 2.2.3 CLOS

Unlike C++ and Eiffel, the form of multiple inheritance featured by CLOS [LAWL 89, PAEP 93] imposes a linear order on the superclasses. This has the advantage that a single keyword `call-next-method` is enough to unambiguously call a superclass method. As a consequence, CLOS avoids the fragility that is caused by allowing the programmer to code explicit superclass references into the source code of arbitrary methods. Another advantage of the CLOS approach is that `super-sends` are dynamically resolved, which means that CLOS can express and apply generic wrappers without any code duplication.

On the downside, CLOS linearization often leads to unexpected behavior because it is not always clear how a complex multiple inheritance hierarchy should be linearized [DUCO 92]. Furthermore, the automatic linearization violates encapsulation because it may change the parent-child relationships among classes in the inheritance hierarchy [SNYD 86, SNYD 87].

## Chapter 3

# Traits

As a solution to the identified problems we propose the traits mechanism [SCHÄ 03], which is a simple extension of single inheritance. In this chapter, we first introduce traits by means of a formal model, graphical diagrams, and running examples (Section 3.1). Then, we evaluate the traits mechanism against the problems identified in the previous chapter (Section 3.2). Finally, we discuss the important decisions in the design of traits and analyze whether and how traits could be modeled in C++ (Section 3.3).

### 3.1 Traits – Composable Units of Behavior

The trait mechanism offers a simple solution to the problems outlined in Chapter 2. *Traits* are essentially sets of methods (*i.e.* mappings from method names to method bodies) that serve as the behavioral building block of classes and the primitive units of code reuse. Classes (and composite traits) are composed from a set of traits by specifying glue code that connects the traits together and accesses the necessary state.

With this approach, classes retain their primary role as generators of instances, while traits are purely units of reuse. As with mixins, classes are organized in a single inheritance hierarchy, thus avoiding the key problems of multiple inheritance, but the incremental extensions that classes introduce to their superclasses are specified using one or more traits.

Traits bear a superficial resemblance to mixins, with several important differences. Several traits can be applied to a class in a single operation, whereas mixins must be applied one at a time. Trait composition is unordered, thus avoiding problems due to linearization of mixins. Traits contain methods, but no state, so state conflicts are avoided, but method conflicts may exist. A class is specified by composing a superclass with a set of traits and some *glue methods*. Glue methods are defined in the class and they connect the traits together; *i.e.*, they implement required trait methods (possibly by accessing state), they adapt provided trait methods, and they resolve method conflicts.

Trait composition respects the following three rules:

- *Methods defined in the class take precedence over trait methods.* This allows the glue methods defined in a class to override methods with the same name provided by the used traits.
- *Flattening property.* A non-overridden method in a trait has the same semantics as if it were implemented directly in the class using the trait.
- *Composition order is irrelevant.* All the traits have the same precedence, and hence conflicting trait methods must be explicitly disambiguated.

A *conflict* arises if we combine two or more traits that provide identically named methods that do not originate from the same trait. Conflicts are resolved by implementing a glue method at the level of the class

that overrides the conflicting methods, or by *excluding* a method from all but one trait. In addition traits allow *method aliasing*; this makes it possible for the programmer to introduce an additional name for a method provided by a trait. The new name is used to obtain access to a method that would otherwise be unreachable because it has been overridden.

We shall first introduce our formal model by summarizing those aspects of classes that we need to capture. We will then proceed to define traits, and show how traits are used to build classes and composite traits. We will introduce a running example in Section 3.1.3 to illustrate various aspects of the formal model.

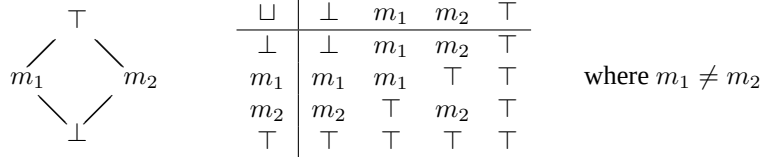
### 3.1.1 Classes and Methods

A key feature of traits is that, although classes may be built using traits, the way in which this is done does not affect the semantics of classes. In effect, traits can be inlined, or flattened. For this reason, we start by describing a model of classes without traits.

The primitive elements of our model are the following disjoint sets:

- $\mathcal{N}$ , a countable set of method *names*, and
- $\mathcal{B}$ , a countable set of method *bodies*,
- $\mathcal{A}$ , a countable set of *attribute* names (*i.e.*, instance variables).

To express conflicts, we extend the set of method bodies  $\mathcal{B}$  to a flat lattice  $\mathcal{B}^*$ , with new elements  $\perp$  and  $\top$  such that  $\perp \sqsubseteq m \sqsubseteq \top$ , for all  $m \in \mathcal{B}$ , and in which all other elements are incomparable. We will use  $\perp$  to represent undefined and  $\top$  to represent a method conflict. Thus, the *least upper bound* or *join* operator  $\sqcup$  for  $\mathcal{B}^*$  is as shown:



**Definition 1** A *method* is a partial function mapping a single method name to a particular method body. We use the notation

$$a \mapsto m$$

for the method that maps the name  $a \in \mathcal{N}$  to the method body  $m \in \mathcal{B}$ .

**Definition 2** A *method dictionary*,  $d \in \mathcal{D}$  is a total function,  $d : \mathcal{N} \rightarrow \mathcal{B}^*$  that maps only a finite subset of method names to bodies and has no conflicts, *i.e.*, where  $d^{-1}(\mathcal{B})$  is finite, and  $d^{-1}(\top) = \emptyset$ .

Note that a method dictionary represents a finite set of methods. For this reason we will always specify them extensionally, listing only the mappings to elements in  $\mathcal{B}$ . For example,

$$d = \{a \mapsto m_1, b \mapsto m_2\}$$

defines a method dictionary  $d$  that maps method name  $a$  to body  $m_1$  and  $b$  to  $m_2$ , and all other method names to  $\perp$ .

**Definition 3** A *class*,  $c \in \mathcal{C}$ , is either the empty class, *nil*, or a sequence  $\langle \alpha, d \rangle \cdot c'$ , with attributes  $\alpha \subset \mathcal{A}$ , method dictionary  $d \in \mathcal{D}$ , and superclass  $c' \in \mathcal{C}$ .

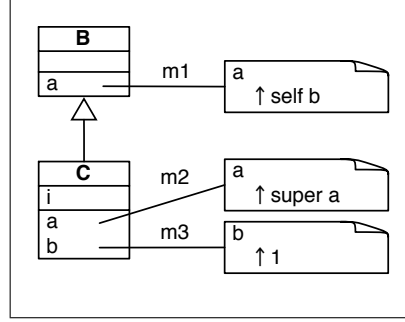


Figure 3.1: An example hierarchy

For the purpose of explaining traits, we do not need to detail the behavior of methods, however we do need to keep track of self and super calls. We therefore model:

- $selfSends : \mathcal{B} \rightarrow 2^{\mathcal{N}}$ , the set of method names used in self-sends, and
- $superSends : \mathcal{B} \rightarrow 2^{\mathcal{N}}$ , the set of method names used in super-sends.

Note that it is considered poor style for a method to perform a super-send to a different method [RIEL 96]. We would therefore expect that for a given method  $l \mapsto m$ ,  $superSends(m) = \emptyset$  or  $\{l\}$ . Since programming languages do not enforce this practice, however, we allow for the more general case.

We extend  $selfSends$  and  $superSends$  to sets of methods in the obvious way:

- $selfSends : 2^{\mathcal{B}} \rightarrow 2^{\mathcal{N}}$ ,  $selfSends(\mu) \stackrel{\text{def}}{=} \bigcup_{m \in \mu} selfSends(m)$
- $superSends : 2^{\mathcal{B}} \rightarrow 2^{\mathcal{N}}$ ,  $superSends(\mu) \stackrel{\text{def}}{=} \bigcup_{m \in \mu} superSends(m)$

(We will later further extend these functions to traits and classes in a similar way, simply taking the union of all self-sends or super-sends of the methods belonging to the trait or class.)

**Example.** Consider a class  $c$  defined as follows:

$$c = \langle \{i\}, \{a \mapsto m_2, b \mapsto m_3\} \rangle \cdot \langle \emptyset, \{a \mapsto m_1\} \rangle \cdot \text{nil}$$

Class  $c$  has attribute  $i$  and methods  $a \mapsto m_2$  and  $b \mapsto m_3$ . Its superclass is  $\langle \emptyset, \{a \mapsto m_1\} \rangle \cdot \text{nil}$ , whose superclass in turn is  $\text{nil}$ . (For conciseness, we will in future omit the trailing  $\cdot \text{nil}$  from all examples.) Note that  $a \mapsto m_2$  in  $c$  overrides  $a \mapsto m_1$  in its superclass (cf. Figure 3.1). Suppose now that:

$$\begin{array}{ll} selfSends(m_1) &= \{b\} & superSends(m_1) &= \emptyset \\ selfSends(m_2) &= \emptyset & superSends(m_2) &= \{a\} \\ selfSends(m_3) &= \emptyset & superSends(m_3) &= \emptyset \end{array}$$

Since the superclass of  $c$  requires, but does not implement  $b$ , we see that method  $b$  is abstract. Furthermore,  $a \mapsto m_2$  not only overrides  $a \mapsto m_1$ , but it extends it by means of a super call. As a consequence, we see that knowledge of the superclass chain is essential to capturing the semantics of classes, since the meaning of  $a \mapsto m_2$  depends on  $a \mapsto m_1$ .

Shortly we will formalize what it means for a class to be abstract or concrete, and when two classes constructed in different ways are equivalent.

### 3.1.2 Traits

We model traits as an extension of method dictionaries where some methods may conflict. Conflicts may arise when traits are composed; the conflicts can be resolved when the composed trait is used in another class or trait. A trait both provides methods (*i.e.*, the methods implemented in the trait) and requires methods (*i.e.*, those that are invoked by self-sends and super-sends, but are not provided).

**Definition 4** A trait,  $t \in \mathcal{T}$ , is a function,  $t : \mathcal{N} \rightarrow \mathcal{B}^*$ , where  $t^{-1}(\mathcal{B})$  is finite.

**Example.** A trait, like a method dictionary, represents a finite set of methods. For example,

$$t_1 = \{a \mapsto m_1, b \mapsto m_2, c \mapsto \top\}$$

defines a trait  $t$  that maps method name  $a$  to body  $m_1$ ,  $b$  to  $m_2$ , and for which method name  $c$  has a conflict. (Assume  $m_1$  and  $m_2$  to be the same method bodies we saw in the previous example.)

Since traits are just finite mappings, two traits are equal when these mappings are equal, that is, when equal names map to equal method bodies<sup>1</sup>.

By convention, *selfSends* and *superSends* of  $\top$  and  $\perp$  are all  $\emptyset$ . We extend *selfSends* and *superSends* to traits in the obvious way:

- $\text{selfSends} : \mathcal{T} \rightarrow 2^{\mathcal{N}}$ ,  $\text{selfSends}(t) \stackrel{\text{def}}{=} \bigcup_{l \in \mathcal{N}} \text{selfSends}(l)$
- $\text{superSends} : \mathcal{T} \rightarrow 2^{\mathcal{N}}$ ,  $\text{superSends}(t) \stackrel{\text{def}}{=} \bigcup_{l \in \mathcal{N}} \text{superSends}(l)$

In the example,  $\text{selfSends}(t_1) = \{b\}$  and  $\text{superSends}(t_1) = \{a\}$ .

**Definition 5** The *conflicts* :  $\mathcal{T} \rightarrow 2^{\mathcal{N}}$ , of a trait  $t$  are defined by:

$$\text{conflicts}(t) \stackrel{\text{def}}{=} \{l \mid t(l) = \top\}$$

In the example,  $\text{conflicts}(t_1) = \{c\}$ .

Note that every method dictionary is, by definition, a trait, but traits with conflicts are not method dictionaries. In fact, a method dictionary  $d \in \mathcal{D}$  is just a *conflict-free* trait, that is, a trait  $d$  such that  $\text{conflicts}(d) = \emptyset$ . We therefore consider that  $\mathcal{D} \subset \mathcal{T}$ .

The provided names of a trait are the names of the methods that it provides. A trait may also require a set of methods that parameterize the provided behavior.

**Definition 6** The *provided* method names,  $\text{provided} : \mathcal{T} \rightarrow 2^{\mathcal{N}}$ , of a trait  $t$  are:

$$\text{provided}(t) \stackrel{\text{def}}{=} t^{-1}(\mathcal{B})$$

*i.e.*, the set of all names that  $t$  does not map to  $\perp$  or  $\top$ . In the example,  $\text{provided}(t_1) = \{a, b\}$ .

**Definition 7** The *required* names,  $\text{required} : \mathcal{T} \rightarrow 2^{\mathcal{N}}$ , of a trait  $t$  are:

$$\text{required}(t) \stackrel{\text{def}}{=} \text{selfSends}(t) \setminus \text{provided}(t)$$

In the example,  $\text{required}(t_1) = \emptyset$ , since  $\{b\} \setminus \{a, b\} = \emptyset$ . In contrast, if we have a trait  $t' = \{a \mapsto m_1\}$ , then  $\text{required}(t') = \{b\}$  since  $b$  is sent to self in  $m_1$  but not provided.

Notice that the required names of a trait do not consider super-sends. This is because traits, like mixins, do not bind super. When we compose a class from traits, however, we must take super-sends into account to determine whether the class is well-founded (cf. Definition 19).

Since traits contain only methods, they cannot specify any state, nor access state directly. Trait methods can access state indirectly, using required methods that are ultimately provided by accessors (getter and setter methods) of the class using the trait.

<sup>1</sup>Our formalization does not define equality of method bodies. This is because equality of method bodies may be established in a variety of ways, *e.g.*, by the location of the source code, or by the syntactic equality or equivalence of source code.

### 3.1.3 Composing Classes from Traits

Trait composition does not subsume single inheritance; trait composition and inheritance are complementary. Whereas inheritance is used to derive one class from another, traits are used to achieve structure and reusability *within* a class definition. We summarize this relationship with the equation

$$\text{Class} = \text{Superclass} + \text{State} + \text{Traits} + \text{Glue methods}$$

This means that a class is derived from a superclass by adding the necessary attributes (state variables), using a set of traits, and implementing the glue methods that connect the traits together and serve as accessors for the attributes. For a class to be *complete*, all the requirements of the traits must be satisfied, *i.e.*, methods with the appropriate names must be provided. These methods can be implemented in the class itself, in a direct or indirect superclass, or in another trait that is used by the class.

Whereas an ordinary class has the form  $\langle \alpha, d \rangle \cdot c'$ , a class composed from traits has the form

$$\langle \alpha, d \triangleright t \rangle \cdot c'$$

where  $t$  is a trait, and  $d$  is a method dictionary that may extend and override  $t$ . In general,  $t$  can be a *composition clause*, an expression that specifies the composition of several traits, and possibly aliases or excludes selected methods. The glue we refer to consists precisely of the overriding, alias and exclusion operations.

**Definition 8** The *sum* of two traits is formed by taking the union of the non-conflicting methods and disabling the conflicting methods. For traits  $t_1$  and  $t_2$ , we define their sum  $(t_1 + t_2) : \mathcal{N} \rightarrow \mathcal{B}^*$  as follows:

$$(t_1 + t_2)(l) \stackrel{\text{def}}{=} t_1(l) \sqcup t_2(l)$$

For example:

$$\{a \mapsto m_1, b \mapsto m_2, c \mapsto m_3\} + \{a \mapsto m_1, b \mapsto m_4\} = \{a \mapsto m_1, b \mapsto \top, c \mapsto m_3\}$$

**Proposition 1** Trait sum is associative and commutative.

*Proof* Immediate from the definition, since the *join* operator  $\sqcup$  is associative and commutative.

Because trait sum is commutative, conflicts must be resolved explicitly (cf. Section 3.1.5). Note that equal methods do not conflict, so in the previous example there is no conflict for  $a$ .

**Definition 9** A method dictionary  $d$  may *override* some of the methods in a trait  $t$ . We define  $d \triangleright t : \mathcal{N} \rightarrow \mathcal{B}^*$  as follows:

$$(d \triangleright t)(l) \stackrel{\text{def}}{=} \begin{cases} t(l) & \text{if } d(l) = \perp \\ d(l) & \text{otherwise} \end{cases}$$

Overriding is the key mechanism for resolving conflicts. Note that  $d \triangleright t$  is, in general a trait, not a method dictionary. However, if  $t$  contains conflicts, we can always choose  $d$  so that  $d \triangleright t$  will be conflict-free. For example:

$$\{b \mapsto m_2\} \triangleright \{a \mapsto m_1, b \mapsto \top, c \mapsto m_3\} = \{a \mapsto m_1, b \mapsto m_2, c \mapsto m_3\}$$

In class definitions of the form  $\langle \alpha, d \triangleright t \rangle \cdot c'$ ,  $d$  will typically be used to resolve conflicts in  $t$ , and to provide any missing methods required by  $t$ . For the moment, we will assume that a class formed in this way is *well-defined*, and defer a discussion of what this means to Section 3.1.6.

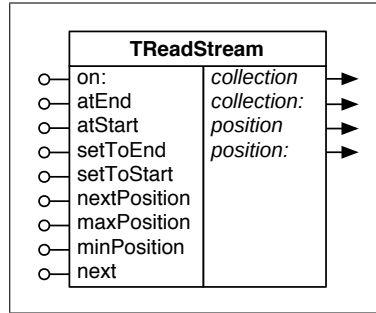


Figure 3.2: The trait TReadStream with provided and required methods

**The flattening property.** An important property follows from the way that classes are constructed from traits. If  $c = \langle \alpha, d \triangleright t \rangle \cdot c'$  is well-defined (cf. Section 3.1.6), and, in particular, if  $d' = d \triangleright t$  is conflict-free, i.e.,  $d'$  is a method dictionary, then  $c$  can clearly be *flattened* to an equivalent definition  $c = \langle \alpha, d' \rangle \cdot c'$  that does not make use of traits. In other words, in any class defined using traits, the *traits can be inlined* to give an equivalent class definition that does not use traits.

As a consequence, traits and classes have the following properties:

- *Methods defined in the class take precedence over trait methods.* This follows from the fact that the methods in the class' method dictionary  $d$  override those provided by  $t$  in  $d \triangleright t$ .
- *Trait methods take precedence over superclass methods.* This follows from the flattening property. Since,  $\langle \alpha, d \triangleright t \rangle \cdot c'$  can be *flattened* to  $\langle \alpha, d' \rangle \cdot c'$ , trait methods behave as if they were implemented in the class itself.
- The keyword `super` is late bound in traits; it simply causes the method lookup to be started in the superclass of the class that *uses* the trait.

Note that the flattening property expresses only that the use of traits can be flattened within a class specification. It does *not* state that inheritance between classes can be flattened. In general inheritance hierarchies may be *refactored*, but not flattened. Later, in Section 3.1.7, we will consider when classes that have been refactored using traits are equivalent.

### Running Example

Suppose that we want to implement a library that provides streams which may be readable, writeable, both readable and writeable, or synchronized. In the examples, trait names start with the letter T, and class names do not. We *italicize* required methods and **embolden** glue methods. Because traits have been implemented in the Smalltalk dialect Squeak [INGA 97], we present the code in Smalltalk.

In our example, stream classes are built from elementary traits such as TReadStream, TWriteStream and TSynchronize. We introduce a minor extension to UML to graphically present traits, as seen in Figure 3.2. The left column lists the provided methods of TReadStream and the right column lists the required methods. The code implementing this trait is shown below. Required methods (shown in *italics*) are flagged by the use of the method body self requirement.

As illustrated in Figure 3.3 and the corresponding Smalltalk code in Figure 3.4, we create the class ReadStream by using the trait TReadStream, which is parameterized by the required methods *collection*, *collection:*, *position*, and *position:*. To be complete, the class ReadStream has to fulfill these requirements by providing corresponding glue methods. In the example, the methods *collection*, *position*, and *position:* are implemented as accessors to two instance variables *collection* and *position*, while the method *collection:* ensures that when

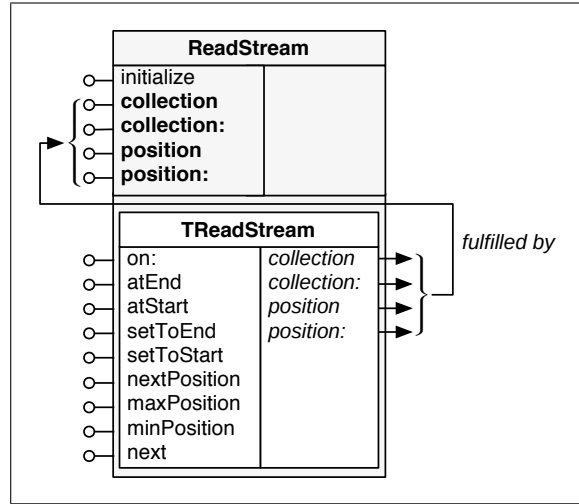


Figure 3.3: The class `ReadStream` composed from the trait `TReadStream`

a new collection is set, the stream is correctly positioned at the start of the collection. The class `ReadStream` also implements a method `initialize` for initializing the two instance variables.

### 3.1.4 Composite Traits

In the same way that classes are composed from traits, traits can be composed from other traits. Unlike classes, most traits are not complete, which means that they do not define all the methods that are required by their subtraits. Unsatisfied requirements of subtraits simply become required methods of the composite trait. Here too, the composition order is not important, and methods defined in a composite trait take precedence over the methods of its subtraits.

**Definition 10** A *composite trait* is a trait expression of the form  $d \triangleright t$ , where  $d \in \mathcal{D}$  and  $t$  is a composition clause (trait expression) using only trait sum (+), alias ( $\rightarrow$ ) and exclusion ( $-$ ) operators. (A formal definition of aliasing and exclusion follows in Section 3.1.5.)

Even in case of multiple levels of composition, the flattening property remains valid. The semantics of a method does not depend on whether it is defined in a trait or in entities that use this trait (cf. Section 3.1.3).

**Example.** Since the traits `TReadStream` and `TWriteStream` contain several identical methods, we now factor out the duplicated behavior into a new trait `TPositionableStream`, which provides all the functionality to manipulate a position over a collection. As illustrated in Figure 3.5, the traits `TReadStream` and `TWriteStream` can then be expressed in terms of the trait `TPositionableStream`.

The implementation of `TPositionableStream` is identical to the implementation of `TReadStream` (cf. Figure 3.4) minus the methods `next` and `on:`. Figure 3.6 shows the implementation of the traits `TReadStream` and `TWriteStream`, which both use the trait `TPositionableStream`. The trait `TReadStream` overrides the methods `on:` and `next`, which position the stream to the beginning of the given collection and read the next element. Similarly, the trait `TWriteStream` overrides the methods `on:` and `nextPut:`, which position the stream to the end of the given collection and append an element.

Note that the unfulfilled requirements of `TPositionableStream` are propagated to the traits `TReadStream` and `TWriteStream`, respectively. This means that the traits `TReadStream` and `TWriteStream` are also parameterized by the required methods `collection`, `collection:`, `position`, and `position:`.

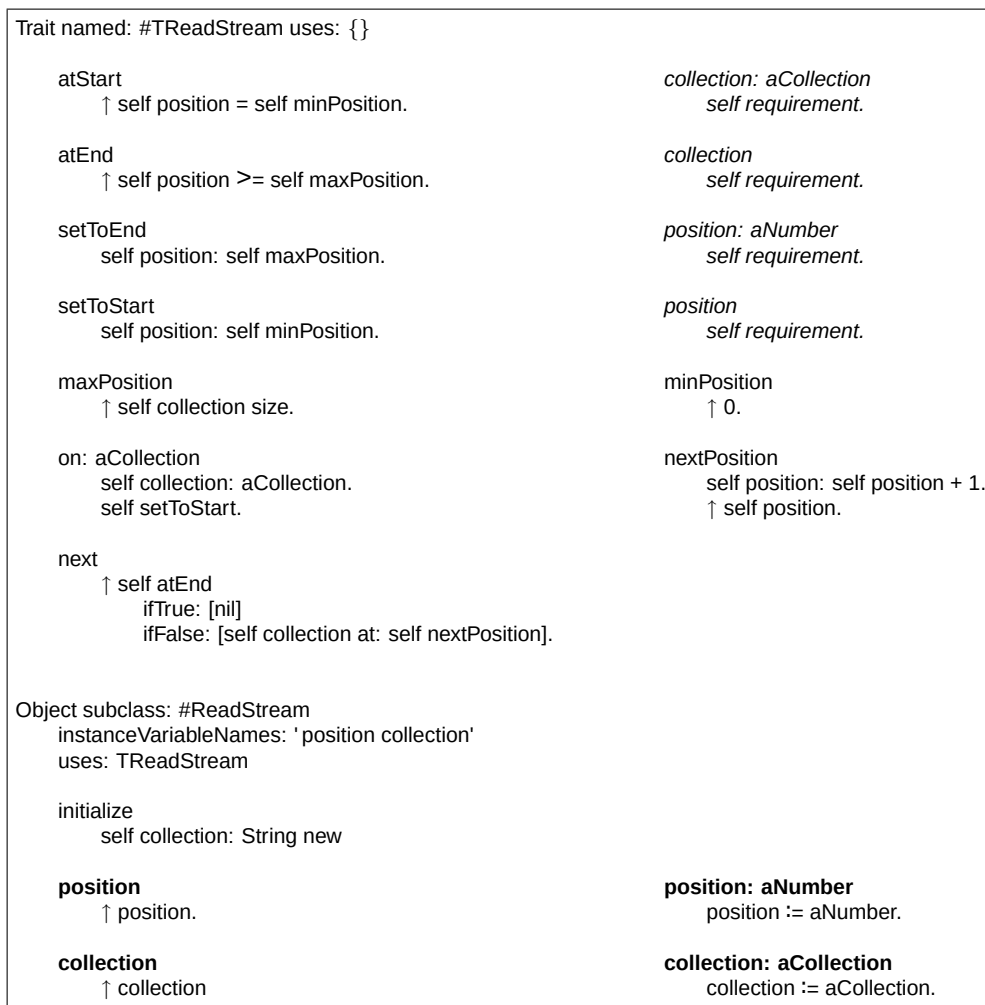


Figure 3.4: Smalltalk implementation of the class ReadStream and the trait TReadStream

### 3.1.5 Conflict Resolution

A conflict arises if and only if we compose two traits that provide identically named methods with different bodies<sup>2</sup>. In particular, this means that if the *same* method is obtained more than once via different paths, there is no conflict (cf. Definition 8). This issue is further discussed in Section 3.3.

Method conflicts must be explicitly resolved by defining an overriding method in the class or in the composite trait, or by means of exclusion in the composition clause. This guarantees that the conflict can be resolved only on the level of the composite, but not by another subtrait that happens to provide a method with the same name.

Oftentimes, conflicts are resolved by overriding the conflict with a new method that somehow combines the conflicting alternatives (*e.g.*, it invokes one conflicting method after the other). To do this without code duplication, traits feature a mechanism called *aliasing*. Aliases allow the programmer to make a trait method

<sup>2</sup>In the Squeak implementation bodies are considered to be different if they originate from different traits — other strategies could also be adopted, such as comparing the source code, or the byte-code.

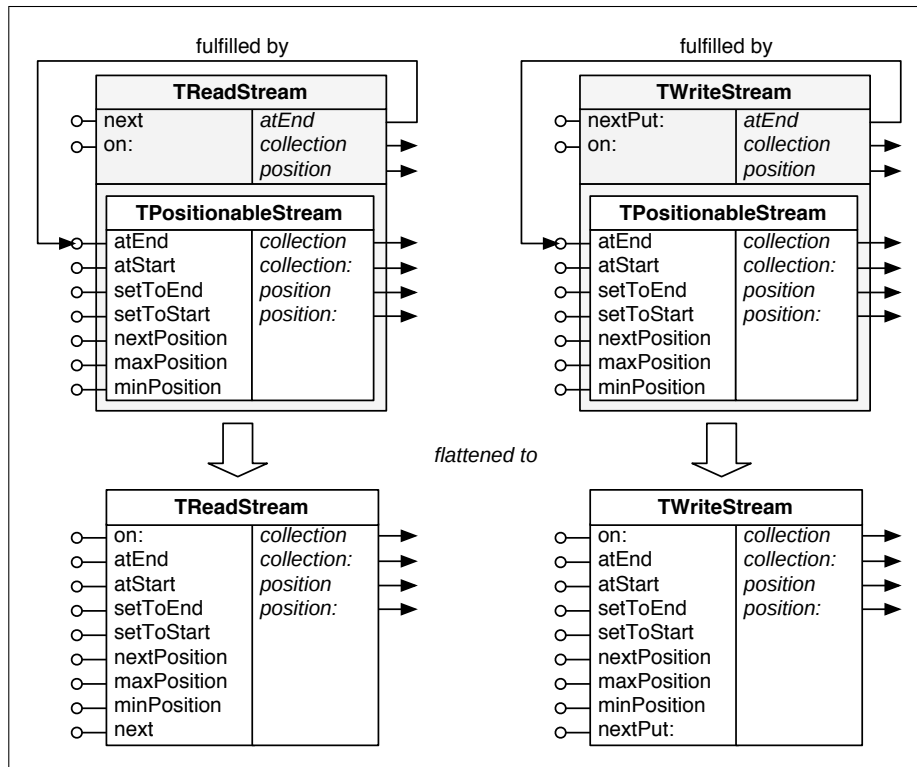


Figure 3.5: TReadStream and TWriteStream as composite traits

```

Trait named: #TReadStream uses: TPositionableStream

on: aCollection
    self collection: aCollection.
    self setToStart.

next
    ↑ self atEnd
    ifTrue: [nil]
    ifFalse: [self collection at: self nextPosition].

Trait named: #TWriteStream uses: TPositionableStream

on: aCollection
    self collection: aCollection.
    self setToEnd.

nextPut: anElement
    ↑ self atEnd
    ifTrue: [self error: 'no space']
    ifFalse: [self collection at: self nextPosition put: anElement].

```

Figure 3.6: Implementation of the composite traits TReadStream and TWriteStream

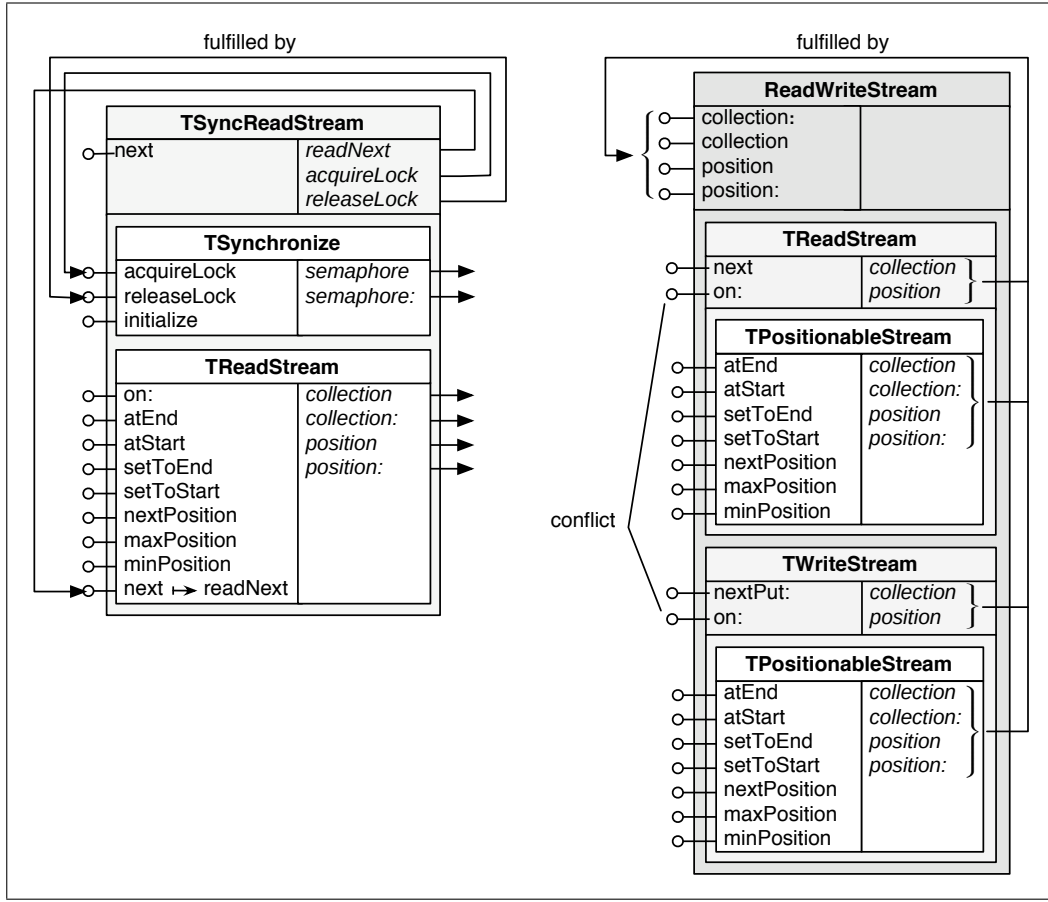


Figure 3.7: Compositions with aliases and conflicts

available under an additional name, which can then be used from any method in the composite entity.

**Definition 11** *Aliasing* introduces an additional name for an existing method:

$$t[a \rightarrow b](l) \stackrel{\text{def}}{=} \begin{cases} t(l) & \text{if } l \neq a \\ t(b) & \text{if } l = a \text{ and } t(a) = \perp \\ \top & \text{otherwise} \end{cases}$$

For example:

$$\{a \mapsto m_1, b \mapsto m_2\}[c \rightarrow b] = \{a \mapsto m_1, b \mapsto m_2, c \mapsto m_2\}$$

Note that  $\{a \mapsto m_1, b \mapsto m_2\}[a \rightarrow b] = \{a \mapsto \top, b \mapsto m_2\}$ , which shows that an attempt to alias an already bound method name will introduce a conflict. Aliases are discussed further in Section 3.3.

In addition to overriding and aliasing, traits also support *exclusion*, which allows one to avoid a conflict before it occurs. The composition clause allows a programmer to exclude methods from a trait when it is composed. This suppresses these methods and hence avoids conflicts with implementations provided by other traits. Note that excluding a method does not necessarily turn the excluded method into a requirement. It causes a requirement only if the name of the excluded method is self-sent by other methods of the trait.

**Definition 12** *Exclusion* creates a trait in which a particular method is not defined:

$$(t - a)(l) \stackrel{\text{def}}{=} \begin{cases} \perp & \text{if } a = l \\ t(l) & \text{otherwise} \end{cases}$$

For example:

$$\{a \mapsto m_1, b \mapsto \top\} - b = \{a \mapsto m_1\}$$

**Example.** As a concrete example for the use of aliases, consider the trait `TSyncReadStream` in Figure 3.7 (left). This trait represents a synchronized read stream, and it is constructed as the composition of the traits `TSynchronize` and `TReadStream`. To ensure that element access is properly synchronized, the `TSyncReadStream` redefines the `next` method provided by the trait `TReadStream`. Since this redefinition needs to invoke the original `next` method provided by the trait `TReadStream`, we create an alias that makes the original method `TReadStream>>next` available under the new name `readNext`.

```

Trait named: #TSynchronize uses: {}

  acquireLock                                semaphore
    self semaphore wait.                      self requirement.

  initialize                                  semaphore: aSemaphore
    self semaphore: Semaphore new.             self requirement.
    self releaseLock.

  releaseLock
    self semaphore signal.

Trait named: #TSyncReadStream uses: TSynchronize + (TReadStream @ {#readNext -> #next})

  next
    | read |
    self acquireLock.
    read := self readNext.
    self releaseLock.
    ↑ read.

```

Figure 3.8: Implementation of the traits `TSynchronize` and `TSyncReadStream`

Figure 3.8 shows the actual implementation of the traits `TSyncReadStream` and `TSynchronize`. (The implementation of `TReadStream` is shown in Figure 3.6.) The trait `TSynchronize` is implemented in a very straight forward way; it provides the methods `acquireLock`, `releaseLock`, and `initialize`, while it requires the methods `semaphore:` and `semaphore`.

The implementation of trait `TSyncReadStream` is more interesting. In the composition clause of `TSyncReadStream`, we first use the operator `@` to create the alias `readNext` for the method `next` provided by `TReadStream`, and then we use the operator `+` to compose the aliased trait with the trait `TSynchronize`. The method `next` is then overridden in `TSyncReadStream` so that it acquires a lock, calls the original method via the alias, and then releases the lock. Note that `TSyncReadStream` does not satisfy the requirements of the traits `TReadStream` and `TSynchronize`, which means that they are propagated to `TSyncReadStream`.

As an example of a conflict, consider the class `ReadWriteStream` shown in Figure 3.7 (right). This class is built from the two traits `TReadStream` and `TWriteStream`, which each provide their own version of the method `on:.` This results in a conflict that may be resolved by excluding one of the conflicting methods or by overriding it in the composite class.

In our example, we avoid the conflict by excluding the method `TReadStream>>on:`, which means that the method `TWriteStream>>on:` will be included in the composite. The corresponding composition clause uses the exclusion operator `-` and looks as follows:

```
Stream subclass: #ReadStream
  uses: (TReadStream - {#on:}) + TWriteStream
```

As shown in Figure 3.5, the traits `TReadStream` and `TWriteStream` are both composed from the trait `TPositionableStream`. Thus, all methods originating from `TPositionableStream` are *identical* in both traits and do not pose a conflict (cf. Definition 8).

### 3.1.6 Well-definedness

We have deferred a discussion of when a class built from traits is well-defined. We will now make that notion precise. In particular, we will define what it means for a class to be valid and well-founded.

**Definition 13** The *dictionary* of a class  $c$ ,  $\text{dict}(c)$ , is the  $\triangleright$  composition of the (flattened) method dictionaries in its inheritance chain:

$$\text{dict}(c) \stackrel{\text{def}}{=} \begin{cases} \{\} & \text{if } c = \text{nil} \\ d \triangleright \text{dict}(c') & \text{if } c = \langle \alpha, d \rangle \cdot c' \end{cases}$$

Note that the *dictionary* of a class  $c$  is not necessarily a (conflict-free) *method dictionary*: if  $c$  is composed from traits, i.e.,  $c = \langle \alpha, d \triangleright t \rangle \cdot c'$ , then  $d \triangleright t$  is not necessarily a (conflict-free) method dictionary in  $\mathcal{D}$ , and  $\text{dict}(c)$  might contain conflicts. We therefore need the following definition to tell us when a class composed from traits is *valid*:

**Definition 14** A class  $c$  is *valid* if  $\text{conflicts}(\text{dict}(c)) = \emptyset$ .

**Definition 15** The *method lookup*,  $c \gg a$ , of a method name  $a$  in a class  $c$  is:

$$c \gg a \stackrel{\text{def}}{=} \text{dict}(c)(a)$$

**Definition 16** The *provided names*,  $\text{provided} : \mathcal{C} \rightarrow 2^{\mathcal{N}}$ , of a class  $c$  are:

$$\text{provided}(c) \stackrel{\text{def}}{=} \{l \in \mathcal{N} \mid c \gg l \in \mathcal{B}\}$$

i.e., the set of all names that  $\text{dict}(c)$  does not map to  $\perp$  or  $\top$ .

**Definition 17** The *increment*,  $\text{delta}(c)$ , of a class  $c$ , is:

$$\text{delta}(c) \stackrel{\text{def}}{=} \begin{cases} \{\} & \text{if } c = \text{nil} \\ d & \text{if } c = \langle \alpha, d \rangle \cdot c' \end{cases}$$

**Definition 18** The *superclass*,  $\text{super}(c)$ , of  $c$  is:

$$\text{super}(c) \stackrel{\text{def}}{=} \begin{cases} \text{nil} & \text{if } c = \text{nil} \\ c' & \text{if } c = \langle \alpha, d \rangle \cdot c' \end{cases}$$

**Definition 19** A class  $c$  is *well-founded* if and only if all super-sends in its inheritance chain are bound, i.e., if  $\text{superSends}(\text{delta}(c)) \subseteq \text{provided}(\text{super}(c))$  and  $\text{super}(c)$  is well-founded. `nil` is well-founded by convention.

For a particular programming language, a class that is not well-founded may generate run-time errors, or compile-time errors, depending on the philosophy of its designers.

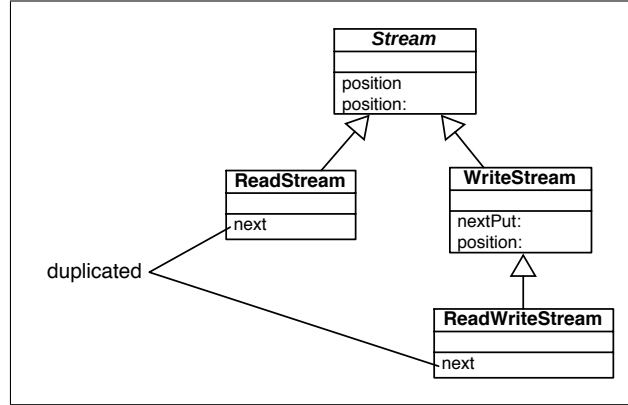


Figure 3.9: Code duplication in the Smalltalk Stream hierarchy

**Definition 20** A class  $c$  is *well-defined* if  $c$  is valid and well-founded.

Now we can state the *flattening property* more precisely:

**Proposition 2** *Flattening property.* If  $c = \langle \alpha, d \triangleright t \rangle \cdot c'$  is well-defined and  $d' = d \triangleright t$ , then  $c = \langle \alpha, d' \rangle \cdot c'$  is an equivalent, flattened specification of  $c$ .

*Proof* Follows trivially from Definitions 13 and 14, since  $c$  is well-defined, it is valid, and hence  $d' \triangleright \text{dict}(c')$  is conflict-free, so  $\langle \alpha, d' \rangle \cdot c'$  is an equivalent, valid class specification.

### 3.1.7 Refactoring, reachability and equivalence

The flattening property helps us to ensure that the semantics of a class does not change when it is rewritten as a composition of traits. However it is inadequate for reasoning about the equivalence of classes when an entire class hierarchy is refactored. The reason for this is that flattening helps us reason about trait composition, but not about inheritance. In particular, flattening says nothing about which methods are reachable through the inheritance chain of the class by means of self-sends and super-sends.

Consider, for example, the class hierarchy in Figure 3.9. We can represent this formally as follows (eliding the trailing nil):

$$\begin{aligned}
 \text{Stream} &= \langle \alpha, \{ \text{position} \mapsto m_{\text{position}}, \text{position:} \mapsto m_{\text{position:}} \} \rangle \\
 \text{ReadStream} &= \langle \beta, \{ \text{next} \mapsto m_{\text{next}} \} \rangle \cdot \text{Stream} \\
 \text{WriteStream} &= \langle \gamma, \{ \text{nextPut:} \mapsto m_{\text{nextPut:}}, \text{position:} \mapsto m_{\text{position:}'} \} \rangle \cdot \text{Stream} \\
 \text{ReadWriteStream} &= \langle \beta, \{ \text{next} \mapsto m_{\text{next}} \} \rangle \cdot \text{WriteStream} \\
 \text{selfSends}(m_{\text{next}}) &= \{ \text{position} \} \\
 \text{selfSends}(m_{\text{nextPut:}}) &= \{ \text{position}, \text{position:} \} \\
 \text{selfSends}(m) &= \{ \} && \text{for all other values of } m \\
 \text{superSends}(m_{\text{position:}'}) &= \{ \text{position:} \} \\
 \text{superSends}(m) &= \{ \} && \text{for all other values of } m
 \end{aligned}$$

As illustrated in Figure 3.9, one problem with this hierarchy is that the method  $\text{next} \mapsto m_{\text{next}}$  is duplicated: it is implemented in both the class `ReadStream` and `ReadWriteStream`. We avoid this duplication by refactoring this hierarchy with traits as shown in Figure 3.10. In the refactored hierarchy, the classes `ReadStream'`, `WriteStream'`, and `ReadWriteStream'` are direct subclasses of the class `Stream'`, while the actual read and write behavior is factored out into two traits `TReadStream` and `TWriteStream`, respectively.

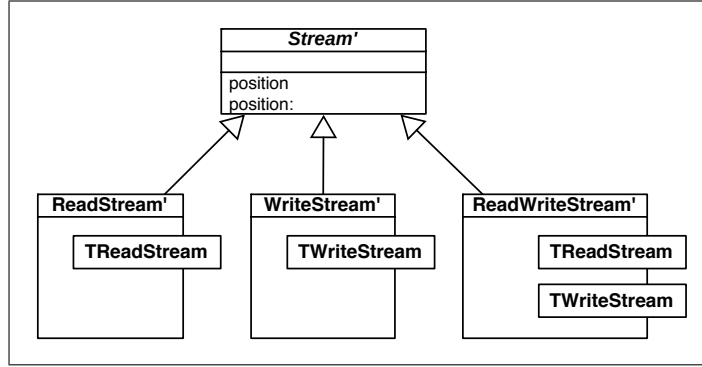


Figure 3.10: The Smalltalk Stream hierarchy refactored using traits

Formally, the new hierarchy is expressed as:

$$\begin{aligned}
 \text{TReadStream} &= \{ \text{next} \mapsto m_{\text{next}} \} \\
 \text{TWriteStream} &= \{ \text{nextPut:} \mapsto m_{\text{nextPut:}}, \text{position:} \mapsto m_{\text{position:}} \} \\
 \text{ReadStream}' &= \langle \beta, \text{TReadStream} \rangle \cdot \text{Stream} \\
 \text{WriteStream}' &= \langle \gamma, \text{TWriteStream} \rangle \cdot \text{Stream} \\
 \text{ReadWriteStream}' &= \langle \beta \cup \gamma, \text{TReadStream} + \text{TWriteStream} \rangle \cdot \text{Stream}
 \end{aligned}$$

But how do we know that we have really preserved the semantics of these classes? In general, we must consider not only the mapping from provided method names to method bodies, but also methods that may be reached by super-sends from those method bodies, methods that may be reached by super-sends from those methods, and so on. We therefore introduce the notation  $c \uparrow a_1 a_2 \dots a_n = \langle m, c' \rangle$  to mean that it is possible for the method body bound to  $a_1$  in class  $c$  to perform a super-send to  $a_2$ , and for the method body bound to  $a_2$  in this context to perform a super-send, and so on, until eventually  $a_n$  can be super-sent. If this occurs, then  $a_n$  will be bound to method body  $m$  obtained from  $c'$ .

**Definition 21**  $c \uparrow \bar{a}$ , where  $c \in \mathcal{C}$  and  $\bar{a} \in \mathcal{N}^+$ , is defined recursively, as follows:

$$\begin{aligned}
 \text{nil} \uparrow a &\stackrel{\text{def}}{=} \langle \perp, \text{nil} \rangle \\
 c \uparrow a &\stackrel{\text{def}}{=} \begin{cases} \langle m, c \rangle & \text{if } m = \text{delta}(c)(a) \in \mathcal{B} \\ \text{super}(c) \uparrow a & \text{otherwise} \end{cases} \\
 c \uparrow \bar{a}b &\stackrel{\text{def}}{=} \begin{cases} \text{super}(c') \uparrow b & \text{if } c \uparrow \bar{a} = \langle m, c' \rangle \text{ and } b \in \text{superSends}(m) \\ \langle \perp, \text{nil} \rangle & \text{otherwise} \end{cases}
 \end{aligned}$$

For example,

$$\begin{aligned}
 \text{ReadWriteStream}' \uparrow \text{position:} &= \langle m_{\text{position:}'}, \text{WriteStream}' \rangle \\
 \text{ReadWriteStream}' \uparrow \text{position: position:} &= \langle m_{\text{position:}}, \text{Stream} \rangle
 \end{aligned}$$

For convenience, we also introduce the notation  $c \downarrow \bar{a}$ , which returns just the method body reachable by  $\bar{a}$  without its class.

**Definition 22** A method body  $m \in \mathcal{B}$  is *reachable* from class  $c$ , if  $\exists \bar{a} \in \mathcal{N}^+$  such that  $m = c \downarrow \bar{a}$ , where

$$c \downarrow \bar{a} \stackrel{\text{def}}{=} m, \text{ where } \exists c' \text{ such that } c \uparrow \bar{a} = \langle m, c' \rangle$$

For example:

$$\begin{aligned} \text{ReadStream} \mid \text{position:} &= m_{\text{position:}}' \\ \text{ReadStream} \mid \text{position: position:} &= m_{\text{position:}} \end{aligned}$$

As expected, method bodies reachable without any super-sends correspond precisely to method lookups:

**Proposition 3** For any class  $c$  and any single message  $a$ ,  $c \mid a = c \gg a$

*Proof* By induction on the depth of the inheritance hierarchy.

For the base case,  $c = \text{nil}$ , we trivially have  $\forall a, \text{nil} \mid a = \perp = \text{nil} \gg a$ .

Now consider  $c = \langle \alpha, d \rangle \cdot c'$ :

(a) Suppose  $d(a) = m \in \mathcal{B}$ . Then  $c \mid a = m = c \gg a$ .

(b) Suppose  $d(a) = \perp$ . Then, by definition,  $c \mid a = c' \mid a$ . Similarly  $c \gg a = c' \gg a$ . But by induction, we have  $c' \mid a = c' \gg a$ , hence  $c \mid a = c \gg a$

**Definition 23** The reachability set of a class  $c$ , is:

$$\text{reachable}(c) \stackrel{\text{def}}{=} \{ \langle \bar{a}, c \mid \bar{a} \rangle \mid \bar{a} \in \mathcal{N}^+, c \mid \bar{a} \neq \perp \}$$

This precisely expresses which method bodies are reachable by means of self-sends and super-sends through the public methods of a class. For example:

$$\begin{aligned} \text{reachable}(\text{ReadStream}) &= \{ \langle \text{position}, m_{\text{position}} \rangle, \langle \text{position:}, m_{\text{position:}}' \rangle, \\ &\quad \langle \text{next}, m_{\text{next}} \rangle \} \\ \text{reachable}(\text{ReadWriteStream}) &= \{ \langle \text{position}, m_{\text{position}} \rangle, \langle \text{position:}, m_{\text{position:}}' \rangle, \\ &\quad \langle \text{position:position:}, m_{\text{position:}} \rangle, \\ &\quad \langle \text{next}, m_{\text{next}} \rangle, \langle \text{nextPut:}, m_{\text{nextPut:}} \rangle \} \end{aligned}$$

Two classes are equivalent if exactly the same method bodies are reachable by the same super-send chains.

**Definition 24** A class  $c$  is *equivalent* to a class  $c'$ ,  $c \equiv c'$ , iff:

$$\text{reachable}(c) = \text{reachable}(c')$$

(Note that  $\equiv$  is trivially reflexive, symmetric and transitive, so is, in fact, an equivalence.)

**Proposition 4**  $c \equiv c' \Rightarrow \text{provided}(c) = \text{provided}(c')$

*Proof*  $a \in \text{provided}(c) \Rightarrow c \gg a \in \mathcal{B} \Rightarrow c \mid a \in \mathcal{B} \Rightarrow \langle a, c \mid a \rangle \in \text{reachable}(c) \Rightarrow \langle a, c \mid a \rangle \in \text{reachable}(c') \Rightarrow \dots \Rightarrow a \in \text{provided}(c')$  In the example, it is now straightforward to show that  $\text{ReadStream} \equiv \text{ReadStream}'$ , and so on.

Finally, we would like to know which classes are abstract and which are concrete. To determine this, we must establish the set of all self-sends in the reachable method bodies, and check if these methods are actually provided. Those that are missing are required.

**Definition 25** The set of self-sends,  $\text{selfSends}(c)$ , of a class  $c$  is:

$$\text{selfSends}(c) \stackrel{\text{def}}{=} \bigcup \{ \text{selfSends}(m) \mid \exists \bar{a}, \langle \bar{a}, m \rangle \in \text{reachable}(c) \}$$

For example,  $\text{selfSends}(\text{ReadWriteStream}) = \{ \text{position}, \text{position:} \}$ .

**Definition 26** The set of *required* names,  $\text{required}(c)$ , of a class  $c$  is:

$$\text{required}(c) \stackrel{\text{def}}{=} \text{selfSends}(c) \setminus \text{provided}(c)$$

**Definition 27** A class,  $c \in \mathcal{C}$ , is *concrete* if  $\text{required}(c) = \emptyset$ . A class that is not concrete is *abstract*.

In particular, `ReadStream` is concrete since

$$\text{selfSends}(\text{ReadStream}) \subset \text{provided}(\text{ReadStream})$$

so  $\text{required}(\text{ReadStream}) = \emptyset$ .

Note that a class that is built-up from traits need not be concrete to be well-defined, so it is also possible to compose abstract classes from traits.

## 3.2 Evaluation

In Section 2.1 we identified a set of problems that are associated with various forms of inheritance. The design of traits was significantly influenced by the desire to solve these problems. In the following, we present a point by point evaluation of the results.

### 3.2.1 Decomposition Problems

**Duplicated features** Duplicated code can easily be factored out into unique traits, which may then be used to compose arbitrary classes, independent of their position in the class hierarchy [BLAC 03].

**Inappropriate hierarchies** Trait composition allows behavior to be reused in a way that is complementary to single inheritance: with trait composition being the primary mechanism for (fine-grained) code reuse, the inheritance hierarchy is freed to capture conformance and conceptual relationships between classes. This means that the programmer can avoid inappropriate inheritance hierarchies by moving reusable methods into traits and apply them only to the classes where they are appropriate and actually needed.

This is illustrated in Figures 3.9 and 3.10, which show a part of the Smalltalk stream hierarchy constructed using single inheritance and traits, respectively. The traditional hierarchy without traits (Figure 3.9) does not correctly model the conceptual relationship between the stream classes: the class `ReadStream` is related to `WriteStream` but not to `ReadStream`. Furthermore, this hierarchy involves code duplication. Both of these problems are avoided in the hierarchy based on traits (Figure 3.10). This hierarchy maximizes code reuse and is conceptually consistent.

#### Duplicated wrappers

Generic wrappers, such as the synchronization wrappers discussed in Section 2.1.1, can be expressed easily with traits. In fact, the solution shown at the right side of Figure 2.1 would work if `SyncReadWrite` were a trait, since `super` in a trait refers to the superclass of the class that will actually use that trait. If `SyncA` is defined to be a subclass of `A` and `SyncB` a subclass of `B`, and both subclasses use the trait `SyncReadWrite`, the `super-send` in the trait's `read` and `write` methods will be bound to the respective superclasses `A` and `B` *when the classes `SyncA` and `SyncB` are composed*. Other kinds of generic wrappers can be defined in much the same way.

### 3.2.2 Composition Problems

**Conflicting features** Traits avoid state conflicts entirely, because traits cannot define state. Method conflicts may be resolved either by explicitly excluding one of the conflicting methods from the composition, or by overriding the conflict in the composite entity. Because traits are not required to be composed from a root trait containing basic features, the programmer can largely avoid conflicts by using traits that are lean and focus on small sets of collaborating features.

**Lack of control and dispersal of glue code** One of the most significant differences between traits and mixins is that trait sum is associative and commutative, so the ordering of the composition is irrelevant. As a consequence, the composite entity is always in full control of the composition: for each conflicting feature, the composite entity can *independently* choose which trait should take precedence or how the available implementations should be composed. This avoids the need for intermediate “glue components” that are spread over the inheritance hierarchy. Instead, the glue code is always located in the composing entity, reflecting the idea that the composing entity is in complete control of plugging together the components that implement its aspects. This property nicely separates the glue code from the code that implements the different aspects, and it makes a class easy to understand, even if it is composed from many different parts.

As an illustration reconsider the example discussed in Section 2.2.1, where we want to create a new class `MyRectangle` based on the class `Rectangle` and two components adding color and a border. With traits, this is done by putting the color and border behavior into two traits `TColor` and `TBorder` and then defining the new class `MyRectangle` as a subclass of `Rectangle` that uses the two traits `TColor` and `TBorder`. Because the features of these traits are unordered and fully accessible from within the composite class `MyRectangle`, all the glue code that is necessary to resolve conflicts and obtain the intended behavior can be defined within the class `MyRectangle`.

Note that although trait composition is unordered, it can be productively combined with inheritance to obtain a large variety of different partially ordered compositions. The basic idea is that if we want a class `C` to use two traits `T1` and `T2` in that order, we first introduce a superclass `C'` that uses `T1`, and then we define `C` to inherit from `C'` and use `T2`. This has the consequence that the methods in `T2` override the methods in `T1`. This strategy proved itself in practice when we refactored the Smalltalk collection hierarchy (see Chapter 6).

**Fragile hierarchies** Any hierarchical approach to composing software is bound to be fragile with respect to certain kinds of change: if a feature that is used by many clients changes, the change will clearly affect all the clients. The important questions are: how severely will the change affect the features and the correctness of direct and indirect clients? Do we need to change implementations, or only glue code? Will there be a ripple-effect throughout the hierarchy due to apparently innocuous changes? Are there changes that implicitly change the behavior of direct or indirect clients in unexpected ways?

Adding methods provided by a trait may well affect clients by introducing a new conflict. However, due to the design of trait composition (*i.e.*, commutativity, explicit conflict resolution), such changes cannot lead to implicit and unexpected changes in the behavior of direct or indirect clients. Furthermore, a direct client can generally resolve a conflict without changing or introducing any other traits, and no ripple effect will occur (*cf.* Section 3.3). For example, if a new method is added, a direct client can always reestablish its original behavior by excluding the newly added method. Neither additional traits nor additional methods or changes to existing methods are needed. In contrast, adding a new method to a mixin may require introducing new glue mixins as well as glue methods in the composite entity in order to reestablish the original behavior (*cf.* Section 2.1.2).

Traits also avoid the fragility problem we identified in multiple inheritance languages such as C++ and Eiffel, where methods get cluttered with navigational glue code because one must explicitly name the class that provides a method to resolve an ambiguity. With traits, conflicting features are accessed by aliases, which are defined in the composition clause and can be called like regular methods. This approach leads to hierarchies that are more robust and easier to understand, because it avoids class references in the source code.

### 3.3 Discussion

In this section we discuss some decisions that significantly influenced the design of traits and the operations on them, and we evaluate whether and how it is possible to express trait composition in C++ by combining templates and multiple inheritance (with virtual base classes).

### 3.3.1 Design Decisions

Traits were designed with other models of classes and inheritance in mind: we tried to combine their advantages, while avoiding their disadvantages. Here, we discuss the most important design decisions.

#### Untangling Reusability and Classes

Although they are inspired by mixins, traits are a new concept because they are composed using a set of distinct composition operators rather than single inheritance and because they cannot define state. Like mixins, they are finer-grained units of reuse than classes and are not tied to a specific place in the inheritance hierarchy. We believe that these two properties improve code reuse and enable better conceptual modeling. Fine-grained reuse is important because the gulf that lies between entire classes and individual methods is too wide.

Traits allow classes to be built by composing reusable behaviors rather than by implementing a large and unstructured set of methods. But unlike mixins, trait composition is unordered and thus agrees with the unordered characteristics of a class' methods.

#### Single Inheritance and the Flattening Property

Instead of replacing single inheritance, we decided to keep this familiar concept and extend it with the concept of trait composition. These two concepts are similar but complementary and work together nicely.

Single inheritance allows one to reuse all the features (*i.e.*, methods and attributes) that are available in a class. If a class can inherit from only a single superclass, inheriting state does not cause complications and a simple keyword (*e.g.*, `super`) is enough to access overridden methods. This form of access to inherited features is very convenient, but it also assigns semantics to the place of a method in the inheritance hierarchy. Therefore, it is generally not possible to understand a class hierarchy without knowing in which class a certain method is implemented.

Traits operate at a finer granularity than inheritance; they are typically used to modularize the behavior *within* a class. As such, traits are designed to capture behavior but not state. In addition, trait composition attributes no semantic significance to the place where a method is defined, with the result that traits enjoy the flattening property.

In combination with single inheritance, traits and the flattening property provide a smooth migration path for single inheritance languages. Given appropriate tool support, a system based on traits not only allows one to write and execute traditional single inheritance code, but even if there are hundreds of deeply composed traits, the user can still *view and edit* the classes in the same way as if the system were implemented without using traits at all (see Chapter 5).

#### Aliasing

Many multiple inheritance implementations provide access to overridden features by requiring the programmer to explicitly name the defining superclass in the source code. C++ provides the scope operator (`::`) [STRO 97], whereas Eiffel provides the keyword `Precursor` [MEYE 97]. With traits, we chose method aliasing in preference to placing named trait references in method bodies to avoid the following problems:

- Named trait references contradict the flattening property, because they prevent the creation of a semantically consistent flattened view without adapting these references in the method bodies.
- Named trait references cause aspects of the trait structure to be hard-coded in the methods that use them. This means that changing the trait structure, or simply moving methods from one trait to another, potentially invalidates many methods.
- Named trait references would require an extension of the syntax of the underlying single inheritance language.

Method aliasing avoids all of these problems. It works with the flattening property because the flattening process can simply introduce a new name for the aliased method body.

Although there are some similarities between aliasing and method renaming as provided by Eiffel, there are also essential differences. Defining an alias *y* for a method *x* in the trait *T* just establishes an alternative name *y* without affecting the original one. In particular, all references to the original name *x* in the used trait *T* remain unchanged (*i.e.*, they still refer to the original name *x*). In contrast, when a method *x* is renamed to *y* in an Eiffel class *C*, the original method name *x* becomes undefined, and all the references to *x* in the class *C* are changed so that they conceptually refer to the new method name *y*.

While renaming violates the flattening property, it has the advantage that it completely frees the old name *x* as if it were never used in *C* (see the discussion of unintended name clashes below). Furthermore, renaming works well with recursive methods, where aliasing is not really adequate.

### Unintended Name Clashes

With traits, as with any other name-based approach to composing software features, unintentional naming conflicts may arise. For example, consider a Java class that should implement two interfaces, where each of these interfaces specifies a method with precisely the same name (and signature), but with different semantics. The same problem also appears in many mixin approaches such as Strongtalk [BAK 02] and JAM [ANCO 00]: if two mixins provide or require two semantically different methods that happen to have the same name, they cannot easily be composed.

At present, traits offer no real solution to this problem — when two traits are composed, it may be that each requires a semantically different method that happens to have the same name. Unlike Eiffel’s renaming, aliases alleviate the problem only to a small extent. In our view, a complete solution requires good refactoring tools, or preferably a flexible encapsulation mechanism [SCHÄ 04a, SCHÄ 04b] (see Section 9.2).

### Conflict Resolution Strategies

Although traits are based on single inheritance, a form of diamond problem may arise when features from the same trait are obtained multiple times via different paths. For example, consider trait `ReadStream` (Figure 3.7) that uses two traits `TWriteStream` and `TReadStream`, which in turn both use the trait `TPositionableStream` (Figure 3.5).

Since traits contain no state, the most nefarious diamond problem does not arise. Nevertheless, in our example, a method `atEnd` provided by `TPositionableStream` will be obtained by `ReadStream` twice. The key language design question is: should this be considered a conflict?

As established in Definition 8, there is no conflict if the *same* method is obtained more than once via different paths. This “same-operation exception”, as it is called by Snyder [SNYD 86], has the advantage of having a simple, intuitive semantics, but it can lead to surprises if the underlying traits are changed. Suppose that trait `TReadStream` is re-implemented so that it no longer uses `TPositionableStream` but still supports the same behavior (*e.g.*, the method `TPositionableStream`  $\gg$  `atEnd` is copied to the trait `TReadStream`). This causes a conflict because trait `ReadStream` now obtains two *different* methods `atEnd`. Thus, what may have appeared to be a strictly internal change to trait `TReadStream` becomes visible to one of its clients.

Although it may seem that this situation will lead to fragile hierarchies, we argue that it does not. When `TReadStream` re-implements `atEnd`, it is changing what it provides to its clients in a way that is less severe, but just as significant, as when it adds or removes methods. Any of these changes may introduce naming conflicts. However, the resulting conflict is a purely *local* matter, that is, it can be corrected by the *direct clients* of `TReadStream` alone. `ReadStream` can easily resolve the resulting conflict by suppressing one `atEnd` or the other.

Let us examine two alternatives to our current rule. One alternative is for `ReadStream` to “automatically” obtain either one `atEnd` or the other, as happens with linearly-ordered mixins. The problem with this

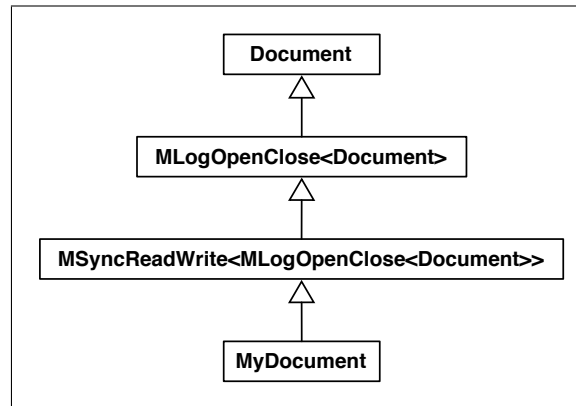


Figure 3.11: Using C++ templates to simulate mixin composition

is that the change to `TReadStream` would give the programmer no feedback, even though the semantics of `ReadWriteStream` might have changed.

The alternative suggested by Snyder is to abandon the “same-operation exception”, and announce a conflict even if the same method is obtained multiple times [SNYD 86]. In our example, this means that there would already be a conflict in the original scenario, and that the programmer would have to *arbitrarily* decide which of the two `atEnd` methods should be available in `ReadWriteStream`. We argue that this is more dangerous, because a later change to the `atEnd` provided by either `TWriteStream` or `TReadStream` will not be signalled as having a possible consequence on `ReadWriteStream`. With the current approach, the conflict is signalled at precisely the point in time when it arises, which is when the programmer is able to make an informed resolution.

### 3.3.2 C++ Revisited

In the discussion above, we pointed out that traits resulted from the attempt to design a composition mechanism that combines the beneficial properties of both multiple inheritance and mixins. C++ is the only language we are aware of that allows the programmer to express both of these composition mechanisms: it has native support for multiple inheritance, and it also allows one to express mixins by using templates (see Section 2.2.2). This poses the interesting question whether it is possible to express a form of composition similar to traits in C++ by combining multiple inheritance with templates.

#### Traits in C++

It turns out that this is indeed possible. The trick is that instead of expressing the reusable entities as generic classes and composing them into a linear inheritance hierarchy by template instantiation as suggested by Van-Hilst and Notkin [VANH 96a, VANH 96b] as well as Smaragdakis and Batory [SMAR 98, SMAR 00], we express them as classes with a *virtual* generic base class and then compose them into a *parallel* hierarchy using multiple inheritance.

The conceptual difference between these two approaches is illustrated in Figures 3.11 and 3.12. Figure 3.11 shows how the class `MyDocument` is derived from the class `Document` by a nested instantiation of the templates `MLogOpenClose` and `MSyncReadWrite`, which leads to a linear hierarchy (cf. Section 2.2.2). In contrast, Figure 3.12 shows how `MyDocument` is built from two templates `TLogOpenClose` and `TSyncReadWrite`, which are both applied to the class `Document` and are then composed using multiple inheritance.

The implementation of the template-based traits `TSyncReadWrite` and `TLogOpenClose` is shown in Figure 3.13. This figure does not show the method bodies because they are identical to the ones in the mixins `MSyncReadWrite` (Figure 2.7) and `MLogOpenClose` (Figure 2.8) discussed in Section 2.2.2. In fact, the only

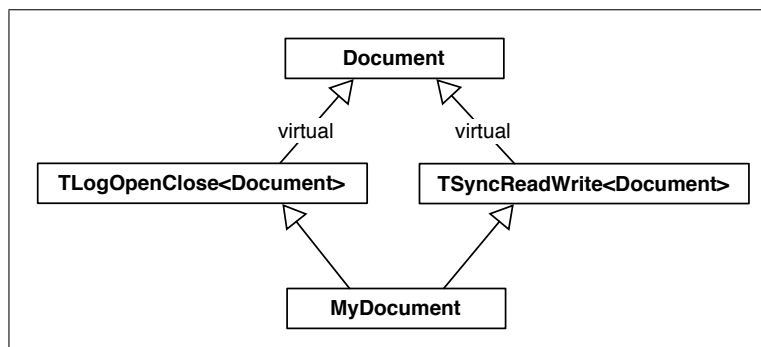


Figure 3.12: Using C++ templates and virtual base classes to simulate trait composition

```

template <class Super>
class TLogOpenClose : virtual public Super {
public:
    virtual void open() { ... };
    virtual void close() { ... };
    virtual void reset() { ... };

protected:
    virtual void log(String s) { ... };
};

template <class Super>
class TSyncReadWrite : virtual public Super {
public:
    virtual int read() { ... };
    virtual void write(int n) { ... };
protected:
    virtual void acquireLock() { ... };
    virtual void releaseLock() { ... };
};

class MyDocument : public TLogOpenClose<Document>,
                  public TSyncReadWrite<Document> {
    ... // glue methods
};
  
```

Figure 3.13: Implementing MyDocument as the composition of two “C++ traits”

difference between the mixins and the corresponding traits is that the traits declare their generic base classes to be virtual.

Declaring the base class to be virtual is crucial as it would otherwise not be possible to correctly compose the traits using multiple inheritance. This is because composing these two traits means instantiating them with the *same* base class `Document` and then combining them using multiple inheritance. According to the semantics of virtual base classes [STRO 97], the resulting diamond situation has the key properties known from traits: the common base class `Document` is only inherited once, and methods in the traits `TLogOpenClose` and `TSyncReadWrite` override methods inherited from the common base class `Document`, while they are overridden by methods in the common subclass `MyDocument`. Furthermore, methods that are implemented by both traits `TLogOpenClose` and `TSyncReadWrite` result in a conflict that needs to be resolved in the subclass `MyDocument`.

C++ allows one to express composite traits by nesting the templates that are representing traits. As an example, we can write a new trait `TLogAndSync` as a template class that is parameterized by `super` and inherits from the virtual base classes `TLogOpenClose` and `TReadWriteSync`, which are both instantiated with the new parameter `super`.

```
template <class Super>
class TLogAndSync : virtual public TLogOpenClose<Super>,
                  virtual public TReadWriteSync<Super> {};
```

A difference between traits and their C++ approximation is the fact that C++ only supports one of the three composition operators of traits: it can express trait sum (+) but not alias ( $\rightarrow$ ) and exclusion ( $-$ ). Whereas aliases can be simulated by disciplined use of the scope modifier `::`, this is not the case for exclusions. This means that instead of excluding one or more conflicting methods from a composition, C++ requires the programmer to resolve every conflict by overriding the conflicting methods. While this may result in the same runtime behavior, it is not equivalent from a compositional point of view. This is because in case of exclusions, the occurrence of a new conflicting method always leads to a conflict that requires explicit resolution, for example by excluding the new method. This is not the case for overrides as a newly occurring conflict method gets implicitly overridden by the old conflict resolution code.

## Discussion

Generalizing the above findings, we can say that C++ allows one to express trait-like composition by using a combination of nested templates and multiple inheritance with virtual base classes. As a consequence, one might jump to the conclusion that traits are not really new and therefore not scientifically relevant. However, on closer examination, such conclusions are premature for several reasons.

We first observe that although C++ does not support the complete set of trait composition features, expressing traits in C++ can be achieved only by using a quite sophisticated combination of advanced language mechanisms such as nested templates and virtual base classes. As a consequence, using traits in C++ not only requires a deep understanding of these mechanisms, but it also requires a lot of coding discipline to achieve the robustness benefits promised by the traits mechanism. As an example, the programmer has to avoid using nested scope modifiers (*e.g.*, `Super::Super::reset()`) to avoid fragility with respect to changes. Similarly, all explicit calls to overridden trait methods have to be factored out into separate methods rather than scattering them throughout other methods.

The intrinsic complexity may be part of the reasons why this particular combination of C++ mechanisms was, to the best of our knowledge, not previously identified and suggested as a general composition pattern in C++. This is similar to template-based mixins in C++, which were scientifically investigated and described by VanHilst and Notkin [VANH 96b, VANH 96a] as well as by Smaragdakis and Batory [SMAR 98, SMAR 00] only after mixins were proposed as a fundamental composition mechanism by Moon [MOON 86] and later analyzed by Bracha and Cook [BRAC 90]. As noted by VanHilst [VANH 96b], templates were previously used, for example in the C++ Standard Template Library (STL) [MUSS 96], for genericity (*i.e.*, writing data structures such as collections that can be used in the context of different types), but not for role composition using inheritance.

Another reason for the relevance of traits as a general composition mechanism is the fact that this form of composition cannot be expressed by any of the more recent object-oriented languages such as Java, Python, and Ruby. Even C#, the C++ successor by Microsoft, cannot express traits because neither templates nor multiple inheritance are supported. In fact, the designers of C# and Java came to the conclusion that it is better to reduce the power of these complex and error-prone mechanisms so that they can be used only for *typing* rather than for *composition*. This means that multiple inheritance was replaced by conformance declarations to multiple interfaces (*i.e.*, types), whereas templates were replaced by a generics mechanism that enables the type system to express generic data structures but which cannot express classes with a parameterized superclass<sup>3</sup>.

This is a strong indication that the C++ approach, which provides the programmer with an overwhelming and hard to understand variety of mechanisms for feature composition, is not the right way to go. The main problem is that there is just too much of a risk that the average programmer uses and combines these mechanisms in a non-optimal way that leads to code that is fragile, hard to understand and maintain.

In contrast, traits stand for a *single* composition mechanism that guarantees certain composition properties. By applying traits to a language such as Smalltalk, Java, or C#, we can therefore get most of the compositional power known from C++ without its drawbacks.

---

<sup>3</sup>In Java, generics were added only with version 1.5

## Chapter 4

# Implementation

Traits as described in the previous chapter of this thesis are fully implemented in Squeak [INGA 97], an open-source dialect of Smalltalk-80. Before we describe the traits implementation, we first give a brief overview of the Smalltalk-80 kernel (Section 4.1), which facilitates the understanding of the rest of this chapter. Then we present the design rationale for our implementation (Section 4.2). The actual description of our implementation consists of two parts: the description of the new language kernel that allows us to express traits in Squeak (Section 4.3), and the description of the compile-time flattening process that allows us to avoid any changes to the virtual machine (Section 4.4). Note that a more detailed description of this implementation can be found in Lienhard’s master thesis [LIEN 04].

### 4.1 Overview of the Smalltalk-80 Kernel Architecture

The beauty of pure object-oriented languages like Smalltalk lies in their conceptual simplicity, which can be summarized as “everything is an object”. This means not only that primitive values such as integers are real objects but also that language entities such as classes are represented as objects. Because each object is an instance of a class, each class must consequently be an instance of another class, which is called a *metaclass* [INGA 76, COIN 87, KICZ 91, DANF 94, FORM 99]. This means that in the same way that classes define the properties of ordinary objects, metaclasses define the properties of classes.

In this section, we give a brief overview of how classes and metaclasses are represented and organized in the Smalltalk-80 kernel. For a more detailed description of the Smalltalk architecture, we recommend the excellent book of Goldberg and Robson [GOLD 83].

#### 4.1.1 Organization of Classes and Metaclasses

In Smalltalk, each class is a first-class object, an instance of a so-called metaclass. Smalltalk metaclasses are *implicit*. This means that whenever a class is created, the system automatically and implicitly creates a corresponding metaclass. All of those metaclasses are anonymous: instead of having a globally known name, they can be accessed only by sending the pseudo-message `class` to the corresponding class object. As a consequence, the metaclass of a class named `A` is usually referred to as `A class`, which is also the convention we use in this thesis.

As an illustration, suppose that the programmer defines a class `Point` as a new subclass of `Object`. In the process of creating this new class, the Smalltalk system first creates a new metaclass `Point class` as a subclass of `Object’s` metaclass `Object class`. Only then can the class `Point` be created as the sole instance of this new metaclass.

An important property of this mechanism is that the resulting class and metaclass hierarchies are always *parallel*. (See Chapter 7 for a discussion of the advantages and disadvantages of this design decision.) This

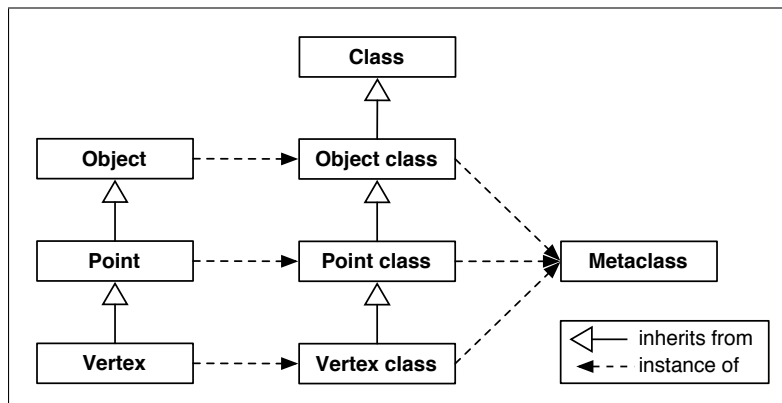


Figure 4.1: Parallel class and metaclass hierarchies in Smalltalk

means that if a class B is a direct subclass of A, then B’s metaclass B class is a direct subclass of A’s metaclass A class.

This is illustrated in Figure 4.1, which shows the classes Object, Point, and Vertex, and their corresponding metaclasses. It also shows the role of the kernel classes Class and Metaclass. As indicated by their names, all the metaclasses such as Object class and Point class are instances of the class Metaclass. Similarly, all classes such as Object and Point are “sub-instances” of the class Class, which means that they are instances of (direct or indirect) subclasses of the class Class.

### 4.1.2 The Kernel Classes

After getting an understanding of the organization of classes and metaclasses in Smalltalk, we now look at how the corresponding classes Class and Metaclass are embedded into the language kernel. The class hierarchy of the kernel is shown in Figure 4.2. At the bottom of this hierarchy, there are the two classes Class and Metaclass. Because these two classes share many commonalities, they inherit from a common superclass called ClassDescription. This class in turn inherits from the class Behavior, which is the root of the kernel hierarchy (*i.e.*, it is a direct subclass of Object). The difference between Behavior and ClassDescription is a conceptual one: the concrete class Behavior is carefully designed so that it provides only the minimal functionality necessary for creating usable instances; the abstract class ClassDescription provides all the other functionality shared between Class and Metaclass.

In the following, we give a brief overview of the responsibilities and the implementation of these kernel classes.

**Behavior.** The concrete class Behavior provides the minimum facilities to create usable instances. This means that it provides basic functionality for instance creation, compiling methods, and managing a method dictionary. It also provides basic facilities for inheritance and class hierarchy management, which is important because of the crucial role of inheritance in the method lookup mechanism. Note that unlike full-fledged classes, behaviors (*i.e.*, instances of the class Behavior) are anonymous (*i.e.*, they have no name), and they can define only anonymous slots and not named instance variables.

To fulfill these responsibilities, the class Behavior defines three instance variables: superclass to store a reference to the superclass, methodDict to store the method dictionary, and format to store a number that encodes important structural information of a behavior’s instances (*e.g.*, whether these instances are variable (like arrays) or fixed in size, and if the latter how many slots they contain).

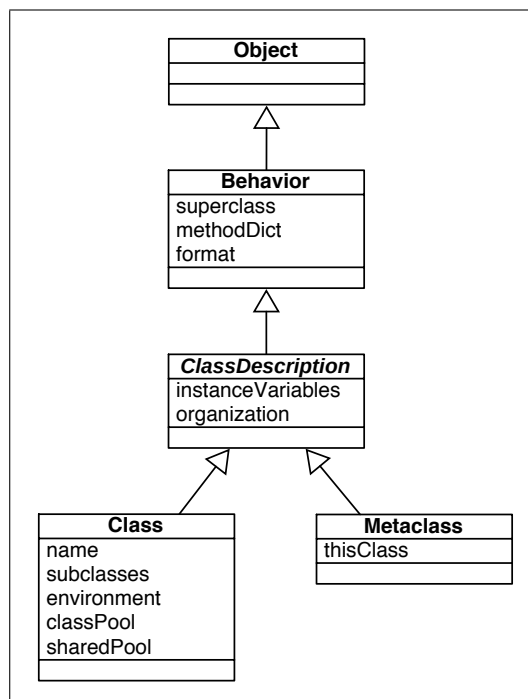


Figure 4.2: Class hierarchy of the Smalltalk kernel

**ClassDescription.** While the purpose of the concrete class Behavior is to allow a programmer to create “minimal classes”, ClassDescription is an abstract class with the sole purpose of defining the common functionality of Class and Metaclass. As such, ClassDescription adds a number of concepts to the ones inherited from the class Behavior: named instance variables, a category organization for methods (*i.e.*, protocols), the notion of a class name, a class comment, logging of changes, and a mechanism for filing the definition of the class in and out.

For this purpose, ClassDescription declares two instance variables: the variable instanceVariables holds an array of instance variable names while organization stores the organization of the methods into named protocols. Note that ClassDescription defers the instance variable holding the class name to its subclass Class. This is because metaclasses are anonymous and only derive a “pseudo-name” from the name of the associated class. Therefore, metaclasses do not (and should not) contain a variable for holding the class name, while classes do have such an instance variable.

**Class.** As indicated by its name, the concrete class Class defines some common behavior of all Smalltalk classes. (As illustrated in Figure 4.1, each class is a sub-instances of the class Class.)

The class Class adds more comprehensive programming support facilities to the basic attributes of Behavior and the descriptive facilities of ClassDescription. In particular, it concretizes its superclass ClassDescription by providing five instance variables and accessors for the class name (name), the set of subclasses (subclasses), the namespace where the class is stored (environment), and pool dictionaries (classPool and sharedPool). The pool dictionaries are used to declare variables that are shared between sets of classes and their instances.

**Metaclass.** While all Smalltalk classes are sub-instances of the class Class, all metaclasses are instances of the class Metaclass. This means that Metaclass defines the behavior of all Smalltalk metaclasses.

Unlike Class, the class Metaclass does not define instance variables for name, subclasses, environment and

pools. Instead, this functionality is derived from the associated class. Thus, the class `Metaclass` declares only a single instance variable `thisClass` to hold the *base-class* (its singleton instance) that is associated with the metaclass.

## 4.2 Traits Design Rationale

The task of adding traits to Squeak can be separated into two parts. First, we need to extend the kernel so that it can represent traits and trait composition. Then we need to make sure that instances of classes composed from traits exhibit the runtime behavior specified by our formal model of traits (cf. Section 3.1).

**Following the structure of the existing kernel.** For reasons of backward compatibility and understandability, we decided that the new language kernel should follow the structure of the existing kernel [GOLD 83] as closely as possible. On one hand, this means that we wanted to preserve the purpose, relationships and responsibilities of the existing kernel classes `Behavior`, `ClassDescription`, `Class`, and `Metaclass`, but it also means that the new classes that are necessary to represent traits should be structured in a similar way.

However, since single inheritance alone is not expressive enough to achieve these goals without code duplication, we decided to bootstrap our new kernel with traits. This means that the new kernel does not only allow one to express traits but is itself built as a composition of traits.

**Avoiding changes to the Squeak virtual machine.** To avoid changing the Squeak virtual machine and to minimize the runtime overhead of traits, our implementation flattens the trait structure at composition time. This means that when a class is composed from traits, its method dictionary is changed to incorporate all the relevant trait methods (*i.e.*, the methods that are not overridden or excluded as well as the aliased methods).

Since compiled methods in traits do not usually depend on the location where they are used, the actual method objects (*i.e.*, the instances of `CompiledMethod` that contain byte-code) can be shared between the trait that defines the method and all the classes and traits that use it. Thus, for most methods this process requires no byte-code to be duplicated. The only exception are the methods that contain super-sends, because they store an explicit reference to the superclass in their literal frame. Duplication of source code is never necessary.

## 4.3 The New Language Kernel

The class hierarchy of the new language kernel with traits (Figure 4.3) clearly resembles the hierarchy known from the traditional Squeak kernel. In fact, the traditional kernel hierarchy described in Section 4.1 is embedded into the new hierarchy by making the class `Behavior` a subclass of the new root class `PureBehavior`. Also the new part of the hierarchy that is used for representing traits (*i.e.*, the classes `TraitBehavior`, `TraitDescription`, `Trait` and `ClassTrait`) has as its root the class `PureBehavior` and mimics the structure of the traditional kernel classes.

### 4.3.1 The New Kernel Classes

The responsibilities and the implementation of the new kernel classes are similar to their traditional counterparts. The main difference is that some of the responsibilities of the class `Behavior` were moved into the new root class `PureBehavior` and that we introduced new responsibilities for dealing with traits. In the following, we give a brief overview of the responsibilities and structure of the new kernel classes, while their decomposition into traits is presented afterwards.

**PureBehavior.** The class `PureBehavior` is the new abstract root class of the kernel hierarchy, and it describes the common facilities to represent behavior (but not state) in classes and traits. This means that it provides the basic functionality for compiling methods, managing a method dictionary and trait composition. Note that

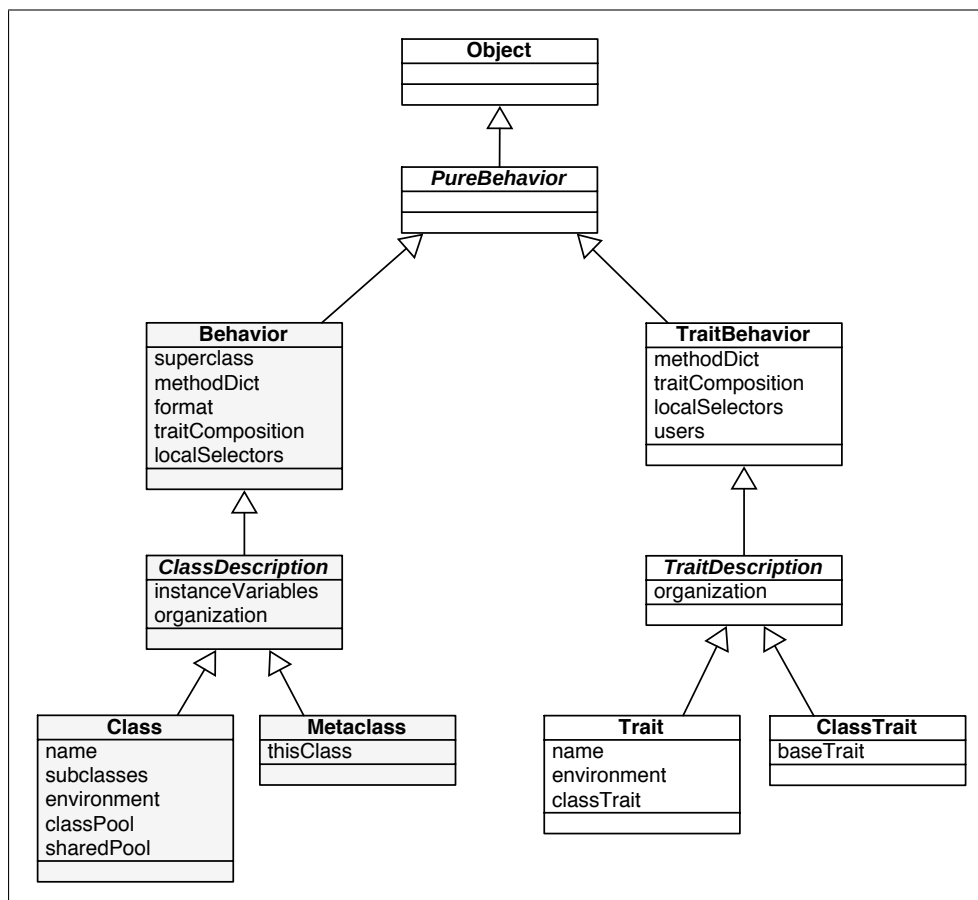


Figure 4.3: Classes hierarchy of the new Smalltalk kernel supporting traits

the facilities for inheritance are provided by the subclass **Behavior** as inheritance is used for classes but not for traits.

To implement these facilities, the implementation of **PureBehavior** accesses three instance variables: **methodDict** to store the method dictionary, **traitComposition** to store the composition clause, and **localSelectors** to store the local selectors (*i.e.*, the selectors in the method dictionary that are actually implemented locally rather than being obtained from subtraits as a result of the flattening). However, because the order of a class' instance variables is hard-coded in the virtual machine, and the instance variable **superclass** (which conceptually does not belong to **PureBehavior**) must be first, the declaration of instance variables is deferred to the subclasses, and they are accessed within **PureBehavior** through abstract accessors.

**Behavior.** As in the traditional Squeak kernel, the concrete class **Behavior** provides the minimum facilities to create usable instances. This means that in addition to the functionality inherited from **PureBehavior**, it provides for instance creation, inheritance, and basic class hierarchy management. Besides the three instance variables that are deferred from the superclass **PureBehavior**, the class **Behavior** declares the two instance variables **superclass** and **format** to contain the superclass and the format of its instances.

**TraitBehavior.** In the same way as **Behavior** provides the minimum facilities for classes, the concrete class **TraitBehavior** provides the minimum facilities for traits. In addition to the inherited functionality, it provides

the facility for keeping track of where a trait is used. This means that it declares an instance variable users to hold all the classes and traits that are composed from the current trait. Like Behavior it also declares the three instance variables deferred from the superclass PureBehavior.

**ClassDescription.** The new class ClassDescription has the same responsibilities as in the traditional kernel. Also the implementation is essentially the same as before, except that it takes into account that the class may be built from traits. For example, the methods for adding new methods check whether there will be a conflict.

**TraitDescription.** The purpose and most of the implementation of the class TraitDescription is identical to ClassDescription, with the main difference being that TraitDescription is used for traits and does therefore not provide the facilities for named instance variables. This means that this class adds categorization for methods, the notion of a trait name, a trait comment, logging of changes, and a mechanism for file-in and file-out to TraitBehavior. Note that we avoid duplicating the code that is shared between TraitDescription and ClassDescription (as well as the code shared between the other kernel classes) by factoring it out into traits (see Section 4.3.2).

**Class.** The purpose of the class Class is the same as in the traditional kernel; *i.e.*, it defines the behavior of all Smalltalk classes. The implementation of Class is essentially the same as before and only differs in that it takes into account that classes may be built from traits.

**Trait.** Each trait in the system is represented as an instance of the class Trait. Like Class, the class Trait concretizes its superclass by providing instance variables name and environment. Since traits do not define variables, the class Trait does not provide facilities for pool variables. However, Trait declares an instance variable classTrait to hold the associated *class-trait*, which is an instance of the class ClassTrait described below.

**Metaclass.** The purpose of the class Metaclass is the same as in the original kernel. Regarding the implementation, the only change is that some of the new methods needed to be changed to reflect the presence of traits. For example the method definitionST80, which returns a textual representation of the metaclass' definition, now also contains the composition clause.

**ClassTrait.** While a class has an associated metaclass, a trait can have an associated class-trait, which is an instance of the class ClassTrait. To preserve metaclass compatibility [GRAU 89, BOUR 98, DUCA 05], the associated class-trait (if there is one) is automatically applied to the metaclass, whenever the trait is applied to a class. Consequently, a trait with an associated class-trait can only be applied to classes, whereas a trait without a class-trait can be applied to both classes and metaclasses.

Like metaclasses, the class ClassTrait declares only a single instance variable to hold the base-trait associated with the class-trait. Since class-traits are anonymous, their name and the environment are then derived from this base-trait.

Despite the conceptual similarity between metaclasses and class-traits, there are two crucial differences. The first difference is that each class is a singleton instance of its associated metaclass, while each trait is an instance of the class Trait rather than an instance of its associated class-trait. This difference is also the reason why we chose the name ClassTrait rather than MetaTrait: a class-trait is just a trait whose methods are applied to the class-side rather than the instance-side. The second difference is that while each class has an associated metaclass, a trait does not necessarily need to have an associated class-trait. If the methods provided by a trait do not rely on any methods on the class-side, there is no need for a trait to have a class-trait.

See Chapter 7 for more information about class-traits, metaclasses, and metaclass compatibility issues.

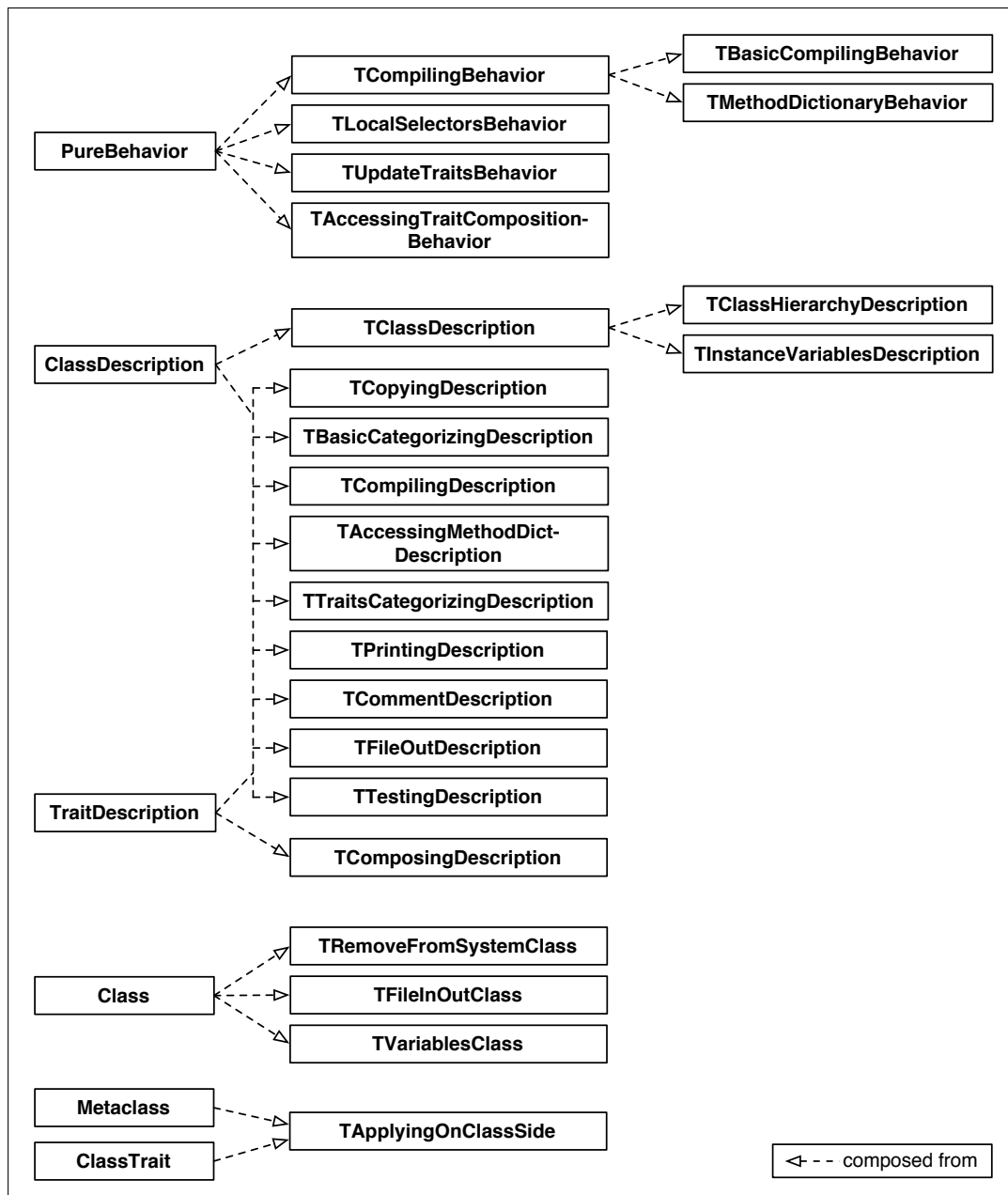


Figure 4.4: Decomposition of the new kernel classes into traits

### 4.3.2 Decomposition into Traits

In the previous section we have described how the traditional kernel classes are embedded into the new kernel hierarchy and how the sub-hierarchy for representing traits mimics the well-known hierarchy for classes. This design has the advantage that it is backward compatible as well as easy to understand and extend. As an example, the conceptual separation between the classes `Behavior` and `ClassDescription` allows a programmer to introduce another notion of classes with different properties (*e.g.*, anonymous classes that organize their instance variables and methods in a different way) by creating another subclass of `Behavior`. Similarly, the distinction between `TraitDescription` and `TraitBehavior` allows one to create another notion of trait by subclassing `TraitBehavior`.

However, this hierarchy cannot be implemented with single inheritance without code duplication. This is because some of the corresponding classes in the trait-part and the class-part of the hierarchy (*e.g.*, `ClassDescription` and `TraitDescription`) share a significant amount of code, but it is not possible to move this code into a common superclass without sacrificing the conceptual distinction between the “behavior” and the “description” of classes and traits.

This is precisely the kind of problem for which traits are designed, and it is only natural to decompose our kernel classes into traits that can then be shared between the different branches of the hierarchy.

Figure 4.4 shows how the classes in the new kernel hierarchy are decomposed into traits. In total, the kernel classes are composed from a total of 23 traits (including all subtraits). The most fine-grained decomposition occurs in the classes `ClassDescription` and `TraitDescription`, which are each composed from 10 traits, 9 of which are shared between the two classes. Note that the classes `Behavior`, `TraitBehavior`, and `Trait` are not shown in the figure as they do not use any traits.

## 4.4 Flattening Traits at Composition Time

In our implementation, we achieve the correct runtime behavior by flattening the traits structure at composition time. According to the formal model presented in Section 3.1, this allows us to ignore the traits structure at runtime and use the ordinary method lookup algorithm that takes only the (flattened) method dictionaries of classes into account.

The flattening process affects the method dictionary of a class `C` that is composed from at least one trait as follows (see Proposition 2 for a formal definition of flattening).

- The method dictionary of `C` is extended with an entry for each provided trait method that is not excluded, is not overridden in `C`, and does not conflict with another method.
- For each alias that does not conflict with another method, we add to the method dictionary of `C` a second entry that associates the new name with the aliased method.
- For each conflicting method, we add to the method dictionary of `C` an entry that associates the method selector with a special method representing a method conflict.

Since compiled methods in traits do not usually depend on the location where they are used, the `CompiledMethod` objects (*i.e.*, the byte-code) can be shared between the trait that defines the method and all the classes and traits that use it. The only exception is the methods that use the keyword `super` because they store an explicit reference to the superclass in their literal frame. Therefore, these methods need to be copied with the entry for the superclass changed appropriately.

Since explicit references to the superclass in the literal table of a method are used only for performance reasons, we do not take these literals into account when determining whether two methods are equal. Therefore, the approach based on copying methods with superclass references does not affect conflict detection. Note that copying of these methods could be avoided by modifying the virtual machine to compute `super` when needed.

## 4.5 Conclusions

Because traits are simple and completely backwards compatible with single inheritance, implementing traits in a reflective single inheritance language like Squeak proved to be quite easy. The fact that traits cannot specify state is a major simplification. We were able to avoid most of the performance and space problems that occur with multiple inheritance, because these problems are related to compiling methods without knowing the offsets of the instance variables in the object [DIXO 89].

Our implementation never duplicates source code, and duplicates byte-code only if it includes sends to `super`, which is not very common within trait methods. A program with traits therefore exhibits the same performance as the corresponding single inheritance program in which all the methods provided by traits are implemented directly in the classes that use those traits. This is especially remarkable because our implementation did not introduce any changes to the Squeak virtual machine. There may be a small performance penalty resulting from the use of accessor methods, but such methods are in any case widely used because they improve maintainability. JIT compilers routinely inline accessors, so we feel that requiring their use in conjunction with traits is entirely justifiable.

Although the new language kernel is relatively small, the fact that we were able to cleanly bootstrap this core part of the system with traits is a good indication for the practical applicability of the traits approach. Besides the fact that traits allowed us to design the new kernel as a backward compatible extension of the traditional one, the use of traits has other advantages for the programmer. For example, it facilitates experimentation with the language because the different aspects of the kernel (*e.g.*, the management of method dictionaries) are now available as traits and can therefore be recomposed to create new kernel classes with different properties.

## Chapter 5

# Tools and Methodology

Adding a new language feature such as traits to an integrated programming environment such as Smalltalk raises two important questions. One question is how the new features interact with the existing tools, and in particular, whether and how these tools need to be modified or extended so that they allow the programmer to take full advantage of the new features. The other question is how the new features affect and change the programming process [BLAC 04].

In this chapter, we first focus on the question of programming tools: we analyze what tools are necessary to allow the programmer to take full advantage of the theoretical properties offered by traits (Section 5.1), and we then present the *trait browser* — the tool that we developed based on this analysis (Section 5.2). Subsequently, we describe the programming methodology that we developed around the use of traits (Section 5.3), and we give examples that illustrate the application of this methodology using the trait browser. Finally, we conclude this chapter by highlighting the relationship between traits, the tools that are motivated by them, and the programming methodology that we developed around the use of traits and these tools (Section 5.4).

### 5.1 Traits and Tools: Analysis

When Smalltalk was first introduced more than 20 years ago, the idea of a graphical programming environment that tightly integrates the language with the corresponding tools was revolutionary. In the intervening years, the concept of the “Integrated Development Environment (IDE)” — for that is what the Smalltalk environment would now be called — has proved to be so successful that similar environments have been created for other programming languages. For example, IBMs VisualAge for Java [JETL 99] was essentially a retargeting of the Smalltalk IDE to Java; more recently the cross-language environment Eclipse [HOLZ 04] has made similar tools available for many popular languages.

An important advantage of such IDEs has been achieved through their integration with the underlying language and its characteristic features. As a result, modern IDEs more than offer the programmer purely syntax-based features such as syntax highlighting. They are also aware of the fundamental language constructs, which allows them to display the dependencies between classes (*e.g.*, inheritance hierarchies), group and organize methods (*e.g.*, sort the methods of a class or hide their bodies), and assist programmers with message completion as they type.

In Smalltalk, the integration between the language and the corresponding tools is particularly tight. In fact, much of the power and elegance of Smalltalk comes from its programming environment and tools, and it is sometimes hard to draw the line between the tools and the actual language, especially for less experienced Smalltalk programmers. As a consequence, adding a major new language construct such as traits to the language immediately raises the question of how this affects the tools [BOBR 80]. In other words, what are the requirements for the Smalltalk programming tools so that they allow the programmer not only to use traits but

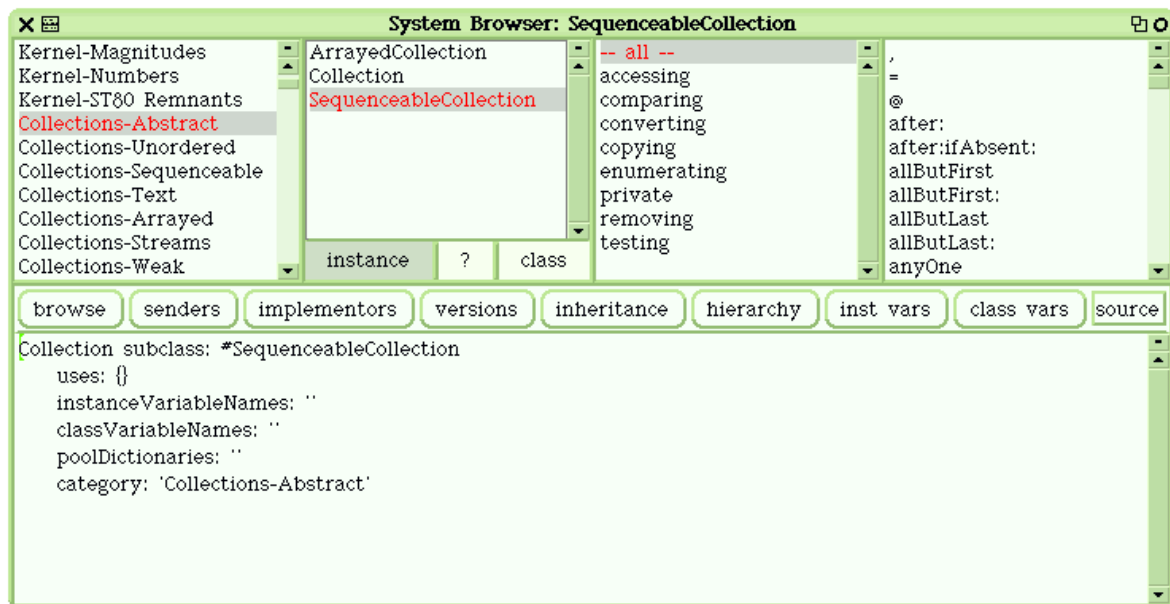


Figure 5.1: The traditional Squeak (Smalltalk) system browser

also to take full advantage of traits and their theoretic properties? An analysis of this situation, led us to the following four requirements.

1. *Defining and using traits.* For traits to be at all useful, the Smalltalk tools must allow the programmer to define new traits easily and use them to compose other traits and classes.
2. *Multiple views.* One of the motivations for traits is the idea that multiple views of a program are better than one, but this idea can only be realized by a programming tool that lets the programmer switch from one view to another without losing context [BOBR 80].
3. *Highlighting collaborations.* The usage of traits leads to programs that are built as fine-grained compositions of classes and traits. In order to understand these compositions, it is important to have a tool that shows the programmer at a glance how the different components collaborate.
4. *Highlighting requirements.* Traits suggest an incremental and iterative style of programming, and it is therefore important to have a programming tool that allows the programmer to keep track of incomplete compositions and indicates what needs to be done to make the program, or at least a part of it, complete.

## 5.2 The Trait Browser

The most important of the Smalltalk programming tools is the system browser, which allows the programmer to examine, modify and extend the code of applications and of the system itself. In this section, we show how we extended this browser to obtain the *trait browser*, which meets all the requirements that we identified in the previous section. We start by giving a brief overview of the traditional browser and our extensions.

### 5.2.1 Overview

**The traditional system browser** Figure 5.1 shows the Squeak system browser. The top half of the browser is split into four panes. The leftmost list pane shows the different class categories, where we have selected the

category *Collections-Abstract*, which contains the abstract collection classes that are consequently shown in the next pane to the right. In this pane, we have selected the class *SequenceableCollection*. As a consequence, the third pane shows the different method categories (*i.e.*, protocols) that are available for this class, and the fourth pane shows the corresponding methods. In the figure, we have selected the method category – *all* –, and thus all the methods implemented in the class *SequenceableCollection* are shown in the method pane.

Because we have not selected any of these methods, the text pane in the bottom half of the browser shows the definition of the selected class *SequenceableCollection* and allows the programmer to modify it. By selecting one of the methods, the programmer could view and edit the implementation of the selected method in the bottom pane. This text pane is also used for defining new classes: if the programmer chooses the “new class” command from the context menu, a template for a new class appears and the programmer can fill in the details such as the class name and the names of the instance variables.

**Towards the trait browser.** According to our list of requirements in Section 5.1, a fundamental requirement for the new browser is that it must allow the programmer to define and use traits (requirement 1). We addressed this by adding a new menu item “new trait” that corresponds to the item “new class” except that it shows the template for a new trait (rather than the class template) in the bottom pane of the browser. We also made sure that the class definitions shown in browser contain the composition clause, which allows the programmer to define how classes are composed from traits by modifying this part of the class definition.

To give the programmer a visual understanding of composite entities, we modified the class pane so that it displays how classes and traits are composed from other traits and allows the programmer to choose the most appropriate view by flattening the nested trait structures at any level (2).

Another important feature of the new browser is that it automatically and interactively categorizes methods into *virtual categories*, which present information that we identified as key to understand how classes and traits collaborate with each other (3). In addition, these virtual categories show the programmer which components and compositions are unfinished and what is necessary to complete them (4).

This information is valuable even when no traits are involved and the only relationship between classes is inheritance [SCHÄ 04c]. However, as the number of components used to build a class increases, it becomes increasingly important to know how these components collaborate with each other. It is therefore not surprising that the presence of traits not only gives the virtual categories a more sophisticated meaning, but also calls for additional categories. These additional categories result either from dividing existing categories into more fine-grained subcategories, or from unique features of trait composition (*e.g.*, conflicting methods).

In the following, we give more details about the distinctive features of the trait browser and their implementation. We first focus on the virtual categories and their meaning in a single inheritance setting. Then, we show how the browser is used with traits and describe how this affects the meaning and diversity of the virtual categories.

## 5.2.2 Virtual Categories

Figure 5.2 shows the trait browser. At first glance it looks like the standard system browser, but some extra features help the programmer understand the relationships between components (*i.e.*, classes and traits). Let us start by summarizing the features that the browser supplies when it is used to examine standard Smalltalk classes.

We have selected class *SequenceableCollection* in the second pane in the top half of the browser window. The third pane, which in the standard browser contains a manual categorization of a class’s methods, now contains in addition some automatically maintained virtual categories.

The category *-requires-*, which is colored blue, includes all of the messages that the class *SequenceableCollection* sends to itself but for which it does not define or inherit a method. If there were no such messages, this category would not appear, but *SequenceableCollection* is abstract, and requires the method *add:*, which is selected and is consequently displayed in the large pane at the bottom of the browser. The implementation shown, *self require*, is a marker method generated by the browser to indicate that *add:* is an unsatisfied

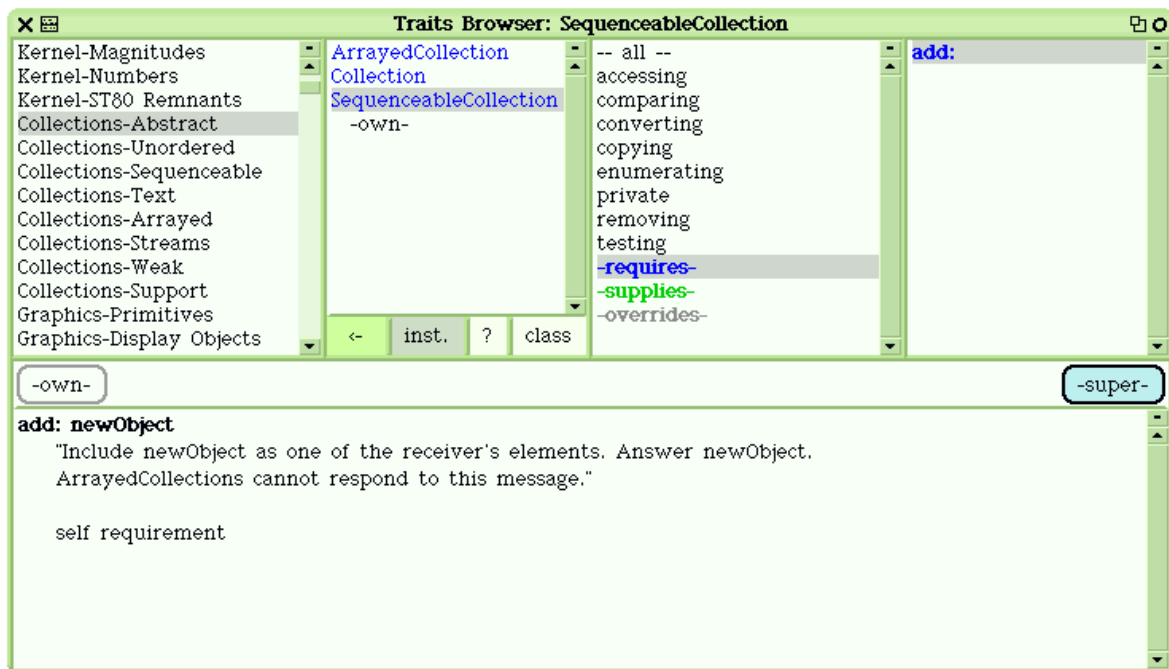


Figure 5.2: The trait browser

requirement. By grouping all the requirements, the *requires* category tells the programmer how a class is parameterized, that is, which methods are necessary in order to make it complete.

The next virtual category, *-supplies-*, lists methods that are *required* by some other class (or trait) in the hierarchy and *provided* by *SequenceableCollection*. There are three methods in this category, namely *atRandom:*, *do:*, and *remove:ifAbsent:*. This tells the programmer that *SequenceableCollection*'s superclass (*Collection*) is parameterized by these three methods and shows the concrete implementations that *SequenceableCollection* supplies for these parameters.

The third virtual category, *-overrides-*, lists those methods provided by *SequenceableCollection* that override inherited methods. In our example, this virtual category includes methods such as *asArray*, *collect:*, and *select:*. Inspecting these methods shows that the reason for overriding the inherited versions is that their functionality can be implemented more efficiently for sequenced collections.

The *-overrides-* category is important for two reasons. First, it provides a view of the class as a “delta”; the methods in this category characterize the parts of the behavior of the superclass that are changed by this subclass. Second, the overriding methods, together with the supplied methods, are the most critical for understanding the class *SequenceableCollection* and for reasoning about its correctness. This is because inheritance breaks encapsulation [SNYD 86]: subclasses typically collaborate with their superclasses through a much broader interface than the public interface of the superclass. Taken together, overridden and supplied methods represent the hooks through which *SequenceableCollection* collaborates with its superclass *Collection*. In particular, this means that the behavior implemented in each of these methods of *SequenceableCollection* needs to conform to the specification implied by *Collection*, and that these are the methods in *SequenceableCollection* that must be adapted if the specification implied by *Collection* is changed.

The fourth virtual category is called *-sending super-*; it contains all of the methods that perform super-sends. This category is important because it tells the programmer which of the methods explicitly collaborate with behavior from their parent class. Furthermore, all of these methods depend on their position in the inheritance hierarchy, so special care has to be taken if they are moved during refactoring. This category is not shown in

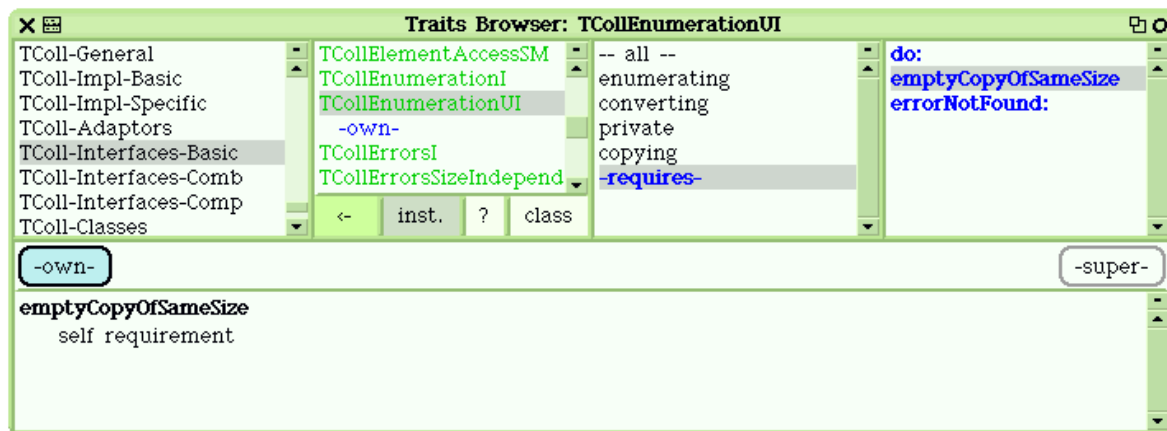


Figure 5.3: The trait TCollEnumerationUI

the figure because it is empty. In fact, each of the virtual categories is displayed by the browser only when it is not empty.

Each of these generated categories has a characteristic emphasis: blue for *requires*, green for *supplies*, grey for *overrides*, and underlined for *sending super*. Even when browsing methods using the ordinary, manually-defined message categories, the names keep their characteristic emphasis. So a supplied method that sends to super will always be shown in green and underlined. The blue color-coding is also applied to the name of the class itself in the second pane whenever the set of required methods is not empty. This serves as a reminder that the class is incomplete, *e.g.*, it may be an abstract class, or the programmer may still be working on it.

As one uses the browser, even if one is not using traits, one becomes accustomed to the subtle hints provided by these colors and to the instant availability of the virtual categories. They provide valuable reminders of work that remains to be done and of dependencies between classes that would otherwise be invisible.

### 5.2.3 Using the Browser With Traits

The browser displays traits in much the same way as classes, and the virtual method categories described in the previous section are also available for traits. In Figure 5.3 we see how the browser shows the trait TCollEnumerationUI, which encapsulates the part of the enumeration protocol that contains only readonly methods that do not require the underlying collection to be sequenced. (The letters U and I in the suffix of the trait name stand for *unsequenced* and *immutable*, respectively. The Smalltalk enumeration protocol and its decomposition into traits is further discussed in Sections 6.1.1 and 6.3.1).

Note that in order to distinguish traits from classes, the browser displays the trait name in green, whereas a class name is either black (if the class is concrete) or blue (if it is abstract or incomplete). At a glance, the browser shows us that the trait TCollEnumerationUI *requires* only the methods `do:`, `emptyCopyOfSameSize` and `errorNotFound`: in order to implement all of the methods that it provides. This means that TCollEnumerationUI can be added to any class that provides these three methods. It does not matter whether or not the candidate class is a subclass of Collection.

Composite traits are a little more interesting because more features of the new browser are visible. Figure 5.4 shows the composite trait TCollEnumerationI, which represents the immutable part of the enumeration protocol and also includes methods that require the underlying collection to be sequenced. When this trait is first selected, the flattened view is shown, which means that all the method categories and methods available in this trait — whether they are defined in the trait itself or obtained from a subtrait — are displayed in the two rightmost panes of the browser.

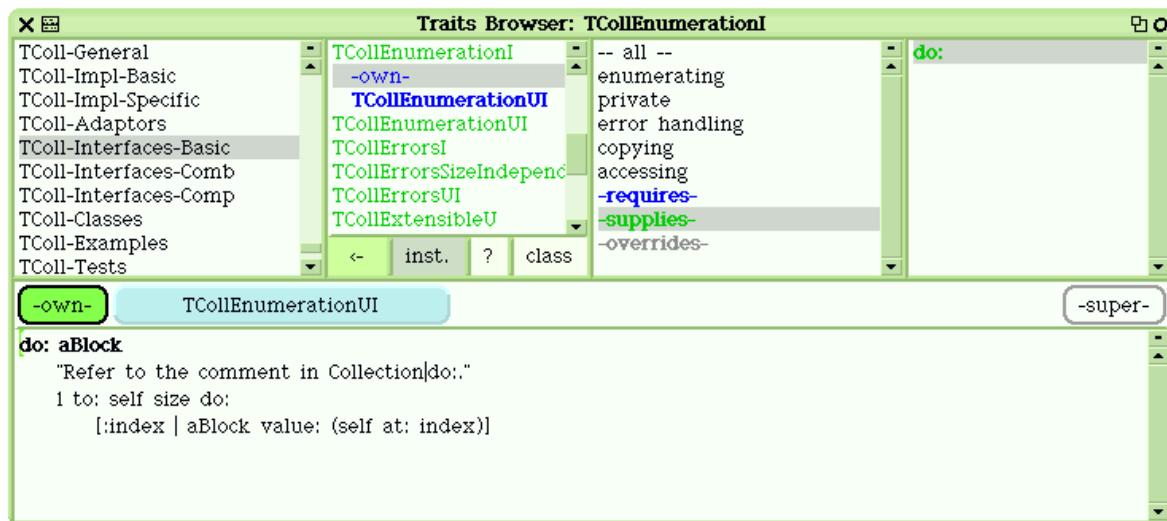


Figure 5.4: The composite trait TCollEnumerationI

In addition, the browser shows the compositional details of the selected trait in the class pane (second from left), which consist of an entry named `-own-` and an entry corresponding to the name of each of the selected trait's subtraits. This not only shows the programmer the subtraits from which the selected trait is composed, it also makes it possible to view this trait in different ways: selecting `-own-` shows only the methods defined in the composite trait itself, while selecting a subtrait, such as `TCollEnumerationUI`, shows the methods obtained from this subtrait together with the corresponding glue methods.

In our example shown in Figure 5.4, the information in the class pane of the browser tells us that the composite trait `TCollEnumerationI` contains a single subtrait called `TCollEnumerationI`, and we could view the methods obtained from this trait by selecting its name. Alternatively, we can view only the methods implemented in the composite trait `TCollEnumerationI` by selecting the entry `-own-`.

Since multiple subtraits are composed with the commutative sum operation, not with inheritance, the categories `-supplies-` and `-overrides-` have a slightly different meaning for traits than for classes. The `-supplies-` category contains the methods that are required by a subtrait and are provided by the currently selected component. In Figure 5.4 we have selected `-own-`, and therefore the `-supplies-` category contains the methods required by the subtrait `TCollEnumerationUI` and implemented by `TCollEnumerationI` itself. Similarly, when the component `-own-` is selected, the category `-overrides-` shows the methods that are provided by a subtrait and then overridden by an “own” method of the trait `TCollEnumerationI`.

In a class built using traits, the virtual category `-overrides-` may contain two kinds of methods: those that override methods of the superclass and those that override methods of a subtrait. Similarly, the virtual category `-supplies-` may contain methods that satisfy requirements originating from the superclass or from a subtrait. To make this distinction visible to the programmer, these virtual categories of a composite class consists of two subcategories: the subcategory *super* contains methods related to the superclass and the subcategory *traits* contains methods that related to subtraits. This is also the case for the other virtual categories.

In addition to the four virtual categories described in Section 5.2.2, there is also a fifth virtual category: `-conflicts-`. This category applies only to entities composed from traits, and lists the methods defined by more than one subtrait: these are conflicts that the programmer must resolve.

The *button bar* in the center of the browser is another feature that helps the programmer to understand and navigate a composition. It shows the programmer all the available versions of a certain method, indicates the version that takes precedence and the version that is on display, and allows the selection of another version. The buttons are arranged from left to right according to the precedence that follows from the trait composition rules:

“own” methods override methods from subtraits, which in turn override methods inherited from the superclass. Our usual color scheme also applies to these buttons: blue buttons indicate required methods, green buttons stand for methods that satisfy a requirement, and red buttons indicate a conflicting method. Thus, the leftmost button that is active (*i.e.*, that is not greyed out) corresponds to the currently applicable method.

In Figure 5.4, the button bar shows that there are two `do:` methods that play a role in our composition: a method that is required by the subtrait `TCollEnumerationUI` and an “own” method that satisfies this requirement. The -super- button to right is grayed out, which indicates that there is no `do:` method inherited from a superclass. The black border on the -own- button in Figure 5.4 indicates that `TCollEnumerationUI`’s “own” version of `do:` is on display; clicking the button `TCollEnumerationUI` would switch the display to the version obtained from that trait.

### 5.2.4 Implementation

The first<sup>1</sup> version of the trait browser is implemented on top of the traditional Squeak system browser. The most challenging aspect of this implementation is computing the contents of the virtual categories so that they can be updated in real-time. Whereas this computation is fairly straightforward for most virtual categories, this is not the case for the *-required-* category.

In this section, we therefore focus on an algorithm for computing the *requires set* of a class efficiently enough for it to be displayed in real-time. For understandability, we do this in a single inheritance setting, which means that we take only the requirements induced by the inheritance chain into account. However, once this algorithm is understood, the more general situation can be solved by applying the same algorithm also to the subtrait chains.

#### Self-Sends and Super-Sends

To compute the required methods of a class we must consider all of the class’s *reachable* methods, that is,

1. all methods that are locally defined in the class,
2. all non-overridden methods defined in its superclasses, and
3. all methods that may be reached by super-sends from other reachable methods.

The *requires set* of a class contains all of the message selectors sent to `self` in one of the reachable methods, minus the selectors of the methods provided by the class (including the ones inherited from its superclasses). (The definition of reachable and required methods are formalized in Section 3.1.7.)

To compute this set, we must first identify the messages that are sent to `self` and `super` in any of the class’ method. Whereas the messages sent to `super` can be immediately retrieved from the byte-code, computing the messages sent to `self` is more complicated, because they do not all emanate from a single syntactic construct. Consider, for example, a method `fasten` that is implemented as follows:

```
fasten
| anObject |
self hook.
anObject := self.
anObject button.
self yourself clip.
```

From the byte-code of this method, it is immediately clear that `hook` is sent to `self` because the receiver is the keyword `self`. What about `button` and `clip`? These messages are also sent to `self` (the method `Object>>yourself` simply returns `self`), and so they are potential requirements too. However, detecting this requires a deeper analysis of this method, as well as of the method `yourself`.

<sup>1</sup>We are currently working on a new version of the trait browser.

Our current implementation does not carry out such an analysis; the only requirements we detect are the (syntactic) self-sends<sup>2</sup>. This means that in the above method, `hook` is the only requirement that we would detect. We compensate for this deficiency by allowing the programmer to declare *explicit requirements* by implementing a method with the body `self explicitRequirement`.

To compute the self-sends and super-sends of a method, we created a subclass of the class `InstructionStream`, which provides the basic facilities to interpret the byte-encoded Smalltalk instruction set. When an instance of this subclass is applied to a `CompiledMethod` object, it interprets the byte-codes without actually executing any primitives or sending any messages. Instead, it scans for message send byte-codes and checks for each of them whether the receiver, which is at this point the topmost element on the stack, is `self`. If this is the case, we first identify the sent message selector, which is either stored in the literal frame of the `CompiledMethod` object or is encoded into the byte-code (in case of very common selectors such as `=`, `+`, `size`, and `at`). The byte-code also tells us whether the message send is a self-send or a super-send. If it is the former, we add the selector to the set to contain the self-sends of the method that is being examined; otherwise, we add it to the set to contain the super-sends.

Even with this simplification, computing the requires set of entire classes in real-time is quite challenging. The main problem is that a single change in a class may affect the requires set of all its subclasses, and that the impact of the change is not limited to the selector that is being changed. Thus, a change in `Object` may mean that we have to update the required methods of all the classes in the system. A naive implementation based on the above definition would be far too slow to provide the programmer with useful feedback (see Section 5.2.4 for a performance comparison). Updating the requires set in real-time required an optimized algorithm that caches critical data and takes advantage of the coherence of the inheritance hierarchy.

### Caching Self-Sends and Super-Sends

When computing the required set of a class, looking for methods that self-send a particular selector is far more common than looking for the selectors sent by a particular method. To avoid having to search all the methods of a class whenever we check whether a certain selector is a potential requirement, we therefore maintain per-class caches that associate the self-sent selectors with the methods where those sends occur.

Concretely, this means that for each class `C` we maintain a dictionary whose keys are the selectors that are self-sent by the methods directly implemented in `C`, and whose values are the sets of selectors that name the methods that perform those self-sends. For example, if the selector `x` is self-sent by the local methods named `a` and `b` in `C`, looking up `x` returns the set `{#a #b}`.

Super-sends are critical for determining the set of reachable methods, so we also maintain a cache of the super-sends that are issued by the local methods of each class; this cache is similarly indexed by the super-sent selectors rather than by the selectors of the methods that perform the super-sends.

In the following, we assume that there are two functions `localSelfSendCacheLookup` and `localSuperSendCacheLookup`, which perform a lookup in these two caches, respectively. This means that the function `localSelfSendCacheLookup` takes a class `C` and a selector `x` as an argument, and returns a set containing the selectors of all methods that are implemented in `C` and self-send `x`. In the above example, the code

```
localSelfSendCacheLookup(C, x)
```

returns the set `{#a #b}`.

### Using the coherence of the inheritance hierarchy

When a method is added, modified or removed, we need to check its class, and all its subclasses, to see whether there is any effect on the requirements. The heart of this computation is checking whether a given selector is self-sent in a given class. Especially in the case of large hierarchies, performing this check separately for each

<sup>2</sup>Note the subtle difference between “messages sent to self” and “self-sends”: the former means all messages that are sent to the current receiver object, whereas the latter includes only the messages sent to the keyword `self`.

```

getSelfSendingClasses(rootClass, sel) {
    var result;

    result := new Set();
    collectSelfSendingClasses(rootClass, sel, new Set(), result, new Dictionary());
    return result;
}

collectSelfSendingClasses(class, sel, inheritedSelfSenders, result, globalSelfSendCache) {
    var selfSenders, dummy;

    selfSenders := inheritedSelfSenders - class.selectors();
    if selfSenders.isEmpty() then
        (selfSenders, dummy) := findSelfSenders(class, sel, new Set(), globalSelfSendCache);
    if selfSenders.notEmpty() then result.add(class);
    for each subclass in class.subclasses() do
        collectSelfSendingClasses(subclass, sel, selfSenders, result, globalSelfSendCache);
}

```

Figure 5.5: Pseudo-code for collecting the classes that self-send a given selector in a hierarchy

subclass proved to be far too slow. Therefore, we developed a recursive algorithm that takes advantage of the coherence that typically exists between neighbor classes in the inheritance hierarchy.

Pseudo-code of this algorithm is shown in Figures 5.5 and 5.6. The algorithm consists of three functions. The function `getSelfSendingClasses` shown in Figure 5.5 is the one to be called from the outside: it takes two arguments `rootClass` and `sel`, and it returns the subset of the classes `rootClass` and all its subclasses that self-send the selector `sel`. This function relies on a second function called `collectSelfSendingClasses`, which collects recursively the classes that self-send the given selector `sel`, starting with `rootClass` and continuing with all its subclasses. As this function proceeds down the inheritance hierarchy, it keeps track of a subset of the methods that are known to self-send `sel`. Unless *all* of these methods are overridden in the next subclass, we immediately know that the subclass also self-sends self. Otherwise, the third function called `findSelfSenders` (Figure 5.6) comes into play. As indicated by its name, this function recursively searches for methods that contain self-sends to `sel` and are actually reachable from this subclass.

We now give a more detailed description of these three methods.

**The function `getSelfSendingClasses`** searches through the argument class `rootClass` and all its subclasses and returns the subset of these classes that self-send the argument selector `sel`. Because the purpose of this function is to provide a more convenient interface for the function `collectSelfSendingClasses`, its implementation consists of only four lines. After we initialize the set to contain the result, we call the function `collectSelfSendingClasses`, which populates the result set, and we return the result.

**The function `collectSelfSendingClasses`** takes five arguments: `class`, `sel`, `inheritedSelfSenders`, `result`, and `globalSelfSendCache`. While `class` is the class to be searched for self-sends to the selector `sel`, `inheritedSelfSenders` is a set containing selectors of inherited methods that are known to self-send `sel`. The argument `result` is the set into which the classes that self-send `sel` are being collected, and `globalSelfSendCache` is a temporary cache holding “global” information about which methods in a class self-send `sel`. This information is “global” in the sense that *all* methods (*i.e.*, locally defined and inherited methods) of a class are considered. This means that `globalSelfSendCache` is a dictionary of associations from a class `C` to a set containing the selectors of all methods that are implemented or inherited in `C` and self-send `sel`. The computation then proceeds as follows.

We first identify the subset `selfSenders` of selectors in `inheritedSelfSenders` that are not overridden in `class`. If this subset is empty, we call the function `findSelfSenders` to search for methods that self-send `sel` and are

```

findSelfSenders(class, sel, unreachable, globalSelfSendCache) {
  // This function returns a pair consisting of:
  // - a subset of all methods that self-send 'sel' (empty set  $\Rightarrow$  no self-senders)
  // - the complete set of all methods that self-send 'sel' or nil if this set has not been computed
  var selfSenders, reachableSelfSenders, translations, allSelfSenders, allInheritedSelfSenders;

  // 1. Check whether the set of all methods self-sending 'sel' in 'class' are cached
  if globalSelfSendCache.includesKey(class) then {
    selfSenders := globalSelfSendCache.at(class);
    return pair(selfSenders - unreachable, selfSenders);
  }

  // 2. Check whether there are local methods that self-send 'sel' and are reachable
  selfSenders := localSelfSendCacheLookup(class, sel);
  reachableSelfSenders := selfSenders - unreachable;
  if isNil(class.superclass()) then { // if this is root of the hierarchy, all methods are local!
    globalSelfSendCache.add(class, selfSenders);
    return pair(reachableSelfSenders, selfSenders);
  }
  if reachableSelfSenders.notEmpty() then return pair(reachableSelfSenders, nil);

  // 3. Compute the set of unreachable superclass methods and perform recursive call
  translations := new Dictionary();
  for each s in class.selectors() do {
    superSenders := localSuperSendCacheLookup(class, s) - unreachable;
    if superSenders.isEmpty() then
      unreachable.add(s);
    else
      translations.add(s, superSenders);
  }
  (reachableSelfSenders, allInheritedSelfSenders) :=
    findSelfSenders(class.superclass(), sel, unreachable, globalSelfSendCache);

  // 4. Replace selectors that are super-sent with the methods that issue the super-sends
  for each s in translations.keys do {
    if reachableSelfSenders.includes(s) then {
      reachableSelfSenders.remove(s);
      reachableSelfSenders.addAll(translations.at(s));
    }
  }
  if isNil(allInheritedSelfSenders) then return pair(reachableSelfSenders, nil);

  // 5. If the set of all self-senders in the superclass has been computed and cached,
  // compute the set of all self-senders in this class and add it to the cache as well
  allSelfSenders := allInheritedSelfSenders - class.selectors() + selfSenders;
  for each s in allInheritedSelfSenders do
    allSelfSenders.addAll(localSuperSendCacheLookup(class, s));
  globalSelfSendCache.add(class, allSelfSenders);
  return pair(reachableSelfSenders, allSelfSenders);
}

```

Figure 5.6: Pseudo-code for finding methods that self-send a given selector

reachable from class. The set containing the selectors of these methods (*i.e.*, the first value of the pair returned by `findSelfSenders`) is then assigned to the variable `selfSenders`. If any such selectors exist (*i.e.*, `selfSenders` is not empty), class is added to the result set. Finally, we perform a recursive call for each of the direct subclasses of class, passing the subclass, `sel`, `selfSenders`, `result`, and `globalSelfSendCache` as parameters.

**The function `findSelfSenders`** takes the four arguments `class`, `sel`, `unreachable`, and `globalSelfSendCache`, and it searches recursively, in class and all its superclasses, for methods that self-send the selector `sel` and are not in the set `unreachable` containing unreachable method selectors. It does this using a twofold strategy. The primary strategy is to return as soon as *some* of these methods are found. However, if this requires searching all the classes up to the root of the inheritance hierarchy anyway, this function also computes, for each of the searched classes, *all* local or inherited methods that self-send `sel` and stores their selectors in the temporary cache `globalSelfSendCache`. This is helpful because it avoids searching these classes again in the future (*i.e.*, when a call to `findSelfSenders` is performed for another subclass).

To accomplish both of these strategies, the function `findSelfSenders` returns a pair of values. The first value is a set containing the selectors of some reachable methods that self-send `sel`. Although this set does usually not contain all such selectors, it is guaranteed to contain at least one of them if one exists. The second value is either `nil` or the set containing the selectors of all methods that are locally implemented or inherited in class and self-send `sel`.

As indicated in Figure 5.6, the computation performed by the function `findSelfSenders` consists of the following five steps:

1. First, we check whether `class` is in the cache `globalSelfSendCache`. If this is the case, we retrieve the associated value, a set containing the selectors of all local or inherited methods in `class` that self-send `sel`, and we return the reachable subset of these method selectors as the first value and the complete set as the second value.
2. If `class` was not found in the cache, we check whether it contains local methods that issue a self-send to `sel` and whose selectors are not in the set of unreachable selectors. This is done by calling the function `localSelfSendCacheLookup`, which accesses the pre-computed cache and retrieves the set containing the selectors of all local methods that self-send `sel`. If this set is not empty, we return the reachable subset of it as the first value and exit. Note that in this step, we also check whether `class` is the root of the inheritance hierarchy. If this is the case, `class` only contains local methods, which means in particular that the identified set of local selectors that self-send `sel` actually contains *all* such selectors, and we therefore return this set also as the second value.
3. Before we use recursion to search the superclass methods, we compute the set of unreachable superclass selectors by adding to `unreachable` all the selectors for which methods are defined in `class` (and therefore potentially override superclass methods) and which are not super-sent by a reachable method in `class`. This is done by calling the function `localSuperSendCacheLookup`, which accesses the pre-computed super-send cache. Then, we perform the recursive call to search superclass methods and store the first value of the resulting pair in the variable `reachableSelfSenders` and the second one in `allInheritedSelfSenders`.
4. For each selector in the set `reachableSelfSenders`, we check whether it is reachable *only* by a super-send. If so, we replace it by the selectors of the local methods that perform the super-send. If the recursive invocation did not compute all the methods that self-send `sel` in the superclass (*i.e.*, if `allInheritedSelfSenders` is `nil`), we return `reachableSelfSenders` as the first and `nil` as the second value and exit.
5. Otherwise, we use `allInheritedSelfSenders` to compute the set `allSelfSenders` containing the selectors of all the methods that are implemented or inherited in `class` and self-send `sel`. Then, we set this set to the cache `globalSelfSendCache` and return it rather than `nil` as a second value.

### Evaluation of the Cache Design

We observe that the design of the pre-computed caches is well-suited to the algorithm. In step (2) of the function `findSelfSenders`, we can find all the local methods that self-send `sel` using a single lookup in the self-send cache (`localSelfSendCacheLookup`). Similarly, in steps (3) and (5) of the same function, we are able to check whether a selector is reachable via a super-send with a single lookup in the super-send cache (`localSuperSendCacheLookup`).

The pre-computed caches can also be kept in a consistent state quite cheaply. This is mainly because these caches contain only local data, that is, the caches for a particular class are independent of all the other classes in the hierarchy. Thus, modifying a class requires updating at most the local cache for this class. Furthermore, changing the place of a class in the hierarchy does not affect the caches at all.

In contrast, the per-class information that is kept in the dictionary `globalSelfSendCache` is global; *i.e.*, the information associated with a class `C` depends not only on the methods implemented locally in `C` but also on `C`'s superclasses and their implementations. Nevertheless, this does not cause any cache maintenance problems because this cache is used only during a single invocation of the function `getSelfSendingClasses`: it is built up and accessed during the execution of this function and is discarded as soon as the function terminates.

### Performance

Our first implementation of the browser cached the self-sends and super-sends of every method. However, unlike the approach presented above, these caches were indexed by the selectors of the methods performing the self-sends rather than by the selectors that were sent. Furthermore, our initial algorithm did not take advantage of the coherence in the class hierarchy. This meant that the method for finding out whether a class self-sends a certain selector was applied to each class separately. Using this implementation to find out which classes in the system required the selector + took several minutes (188 seconds)<sup>3</sup>, which made it impossible to provide immediate feedback.

In a second version of the algorithm, we used the same caching strategy, but took advantage of the coherence in the class hierarchy. Performance improved significantly, but the same computation still took over 9 seconds. Finally, using the caching strategy and the algorithms presented above, the same test takes less than 100 milliseconds and thus meets our requirement for instantaneous feedback.

We also experimented with lazy versions of the collection classes that are used to maintain the set of inherited self-senders and the set of self-senders that are computed and returned as the first value by the function `findSelfSenders`. Unlike the regular collection classes, the operations implemented in these lazy versions compute only limited results. This means for example that the intersection of the sets `{#a #b #c #d}` and `{#a #b #c #e}` may return `{#a #b}` rather than `{#a #b #c}`. Because we used these lazy collections only to maintain sets that do not need to be complete, this does not affect the correctness of our algorithm, but it leads to an additional performance gain because typically, only few of these elements are actually used. Note however that our experiments with these lazy collection classes were only preliminary; further experiments will have to be done to get final answers for the detailed design of these classes and to quantify the performance gain.

## 5.3 Programming Methodology

There are hundreds of books that explain how to write object-oriented programs based on classes that are organized in inheritance hierarchies and millions of programmers have ingrained this style of programming. If such programmers start using traits, it is therefore important to tell them how traits affect the programming process with which they are so familiar.

<sup>3</sup>All the performance data provided in this paper were measured in a Squeak 3.2 image consisting of 1860 classes, and were executed on a Mobile Pentium III 1.2GHz with 512MB RAM.

Based on our experience, both from designing traits and from using them together with other programmers in several case studies (see Chapters 6 and 7), the conceptual difference between traits and classes leads to a natural distinction regarding how and when these two concepts should be used.

In a nutshell, the practical role of traits is to capture the different variants of individual protocols, which can then be composed to build composite protocols and finally classes. This frees the class hierarchy to be used for the conceptual classification of objects.

In this section, we first give an overview of the different roles of classes and traits in an object-oriented program. Then, we illustrate in more detail how traits change the programming process and how the trait browser supports this new process.

### 5.3.1 The Roles of Classes and Traits

Most class-based languages overload the class concept with too many responsibilities. In his thesis, Bracha lists no less than 11 distinct roles for classes [BRAC 92]. At a coarser granularity, we distinguish 5 roles for classes that are relevant to the use of traits:

1. conceptual classification of objects,
2. definition of protocols (interfaces) for objects,
3. modularization — the grouping of related methods,
4. reuse (sharing) of implementation , and
5. incremental modification of an existing class.

It is often difficult, and sometimes impossible, to make a single class hierarchy play all of these roles. Usually, it is the conceptual relationship between the class hierarchy and the domain that suffers, because corrupting this relationship does not immediately break the program. In the case of the Smalltalk collection classes, Cook [COOK 92] has shown how the inheritance hierarchy fails to capture the conceptual relationships between the various collections: the conceptual hierarchy has been subverted to allow greater reuse. For example, whereas dictionaries are conceptually a kind of updatable collection, the Smalltalk class Dictionary is implemented as a subclass of Set. The problem with subverting the inheritance hierarchy in this way is that the code no longer models the domain, and thus it is likely to be more difficult to understand, and harder to modify in response to changes in the requirements or in the domain (cf. Chapter 6).

A trait-based programming methodology avoids this problem. Traits support modularization directly (role 3), and methods encapsulated in a trait can be reused at any point in an inheritance hierarchy (4). In particular, traits provide for incremental modification of an existing class and for the reuse of the “delta” (5).

Most importantly, traits concretize the important but otherwise abstract notion of protocol, and thus make it much simpler for a number of classes to define the same interface, whether or not they are related by inheritance (2). This allows the programmer to implement individual protocols, such as a rectangle protocol or different variants of the enumeration protocols. Because trait composition allows one to combine multiple traits in a commutative way, it is well-suited to combine such protocols, which are naturally unordered, to form composite protocols and finally classes. This frees the class hierarchy to be used for conceptual classification of objects (1).

### 5.3.2 Uniform Protocols

The notion of protocol, also known as interface, is crucial in all object-oriented programs, whether or not the language in which they are written has a syntactic construct to capture it. Uniformity of protocol is encouraged by inheritance, because by default the protocol of a subclass will be a superset of the protocol of its superclass. But classes that are not related by inheritance should also, very often, share the same protocol.

More recent languages such as Java and C# address this issue with an interface construct that allows a programmer to state that two classes should share a uniform protocol. However, they still do nothing to help

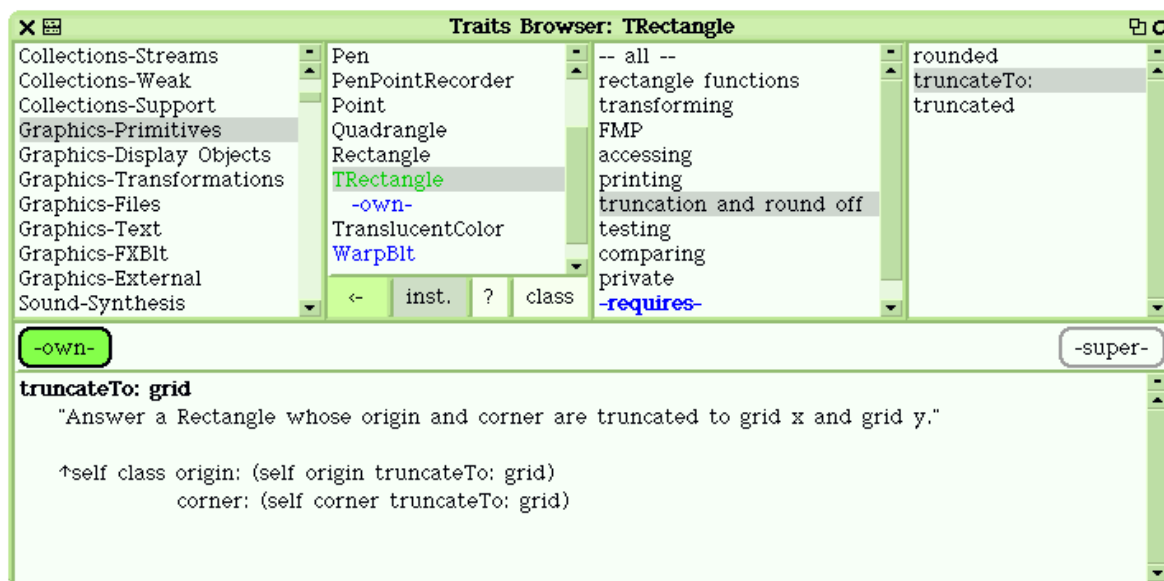


Figure 5.7: The trait TRectangle

translate that desire into executable code, which means that inheritance remains the only tool available to induce this uniformity. The problem is that if inheritance is used for another purpose, the programmer must instead construct the protocols “by hand,” one method at a time. In addition to the dangers of code duplication, protocols constructed by hand are unlikely to stay uniform: over time, one of the classes is likely to be extended while the other is forgotten.

Traits solve this problem by making it possible to construct classes by trait (*i.e.*, protocol) composition as well as by inheritance. Given perfect foresight, any protocol that must eventually be supported by disparate classes can be implemented in a trait, and re-used wherever it is needed. Unfortunately, mere mortals tend to have difficulty applying methodologies that rely on perfect foresight. Instead, we allow programmers to build classes in the conventional way, implementing protocols by placing methods directly in whichever class needs them. If it becomes apparent that the same protocol needs to be supported in an additional (and unrelated) class, the trait browser enables the programmer to extract the protocol into a trait, and then to use this trait as a component of both the original and the additional class.

### Illustration

As an illustration, consider the class `RectangleMorph`. `RectangleMorph` is a subclass of `Morph`, the root of the GUI hierarchy in Squeak. Because `RectangleMorph` looks like a rectangle and also contains the state required to be a rectangle, a user will reasonably expect it to understand the protocol of a `Rectangle`. Paradoxically, `RectangleMorph` does not understand the protocol of class `Rectangle`. This is because inheritance is already used to incorporate the GUI features into `RectangleMorph`, and it is therefore not possible to include the 70 additional rectangle methods without code duplication.

With traits, we can easily overcome this problem by extracting the 70 rectangle methods from the class `Rectangle` into a trait that can then be shared by all the classes supporting the `Rectangle` protocol. In a first step, we use the browser’s functionality to extract the rectangle methods from the class `Rectangle` into a new trait `TRectangle`. In this process, the browser *abstracts* the references to `Rectangle`’s two instance variables (`origin` and `corner`). The result is shown in Figure 5.7.

Notice the effect of the *abstract variable* refactoring on the displayed method: whereas the `truncateTo:`

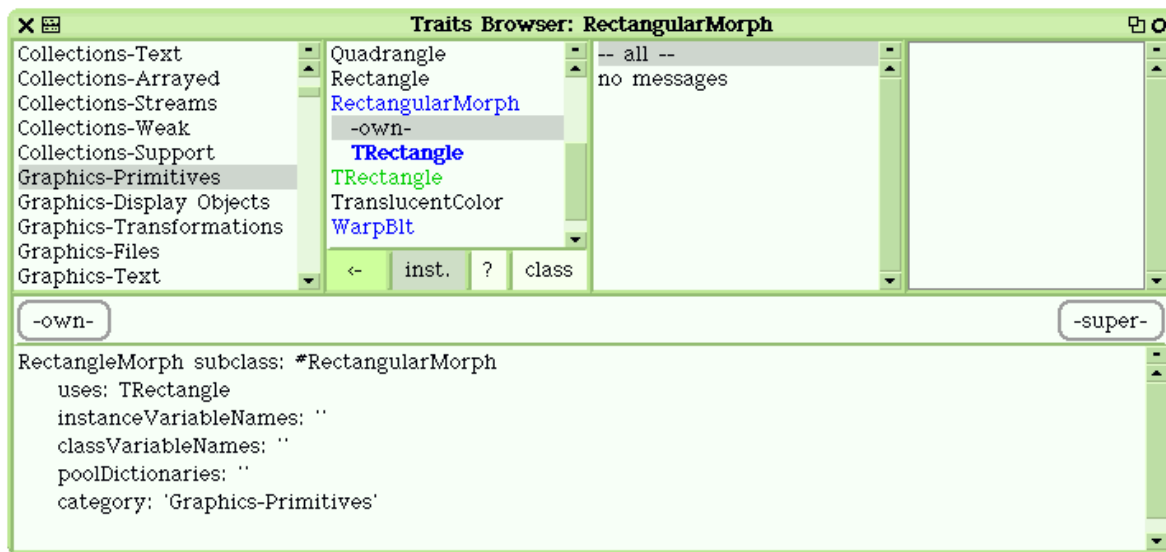


Figure 5.8: The new class RectangularMorph immediately after its creation

method in class Rectangle accessed the instance variables origin and corner directly, the version in TRectangle sends the messages self origin and self corner.

The *-requires-* virtual category of TRectangle contains three methods: origin, corner and species. These methods represent the places where the trait TRectangle must connect to any class in which it is used; they are in effect parameters of the trait.

Continuing with the example, we now define a new class RectangularMorph, which is a subclass of RectangleMorph and uses the newly created trait TRectangle. The state of the browser once this template has been completed is show in Figure 5.8. However, the job is not yet done, which the browser indicates to the programmer by displaying the name of the newly created class RectangularMorph in blue. This means that the new class is incomplete, *i.e.*, that some of its requirements are unsatisfied.

We therefore select the newly created class in the browser and consequently see a list of its components, which appears indented beneath it in the class pane. While the *-own-* pseudo-component is empty (we have not yet written any methods for RectangularMorph), the component TRectangle contains all of the methods that we placed in the trait TRectangle in the previous step. The component TRectangle too is blue, showing that it also has unsatisfied requirements.

To view and edit the required methods, we can select the virtual category *-requires-* of the component TRectangle. For example, we can define

```
corner
  ↑ self bounds corner
```

and similarly for origin.

These newly defined methods populate the *-own-* component of the class RectangularMorph. This is because we defined the methods while browsing RectangularMorph; if instead we had been browsing TRectangle, then the methods would have populated that trait.

Once a required method has been defined, the corresponding selector remains in the *-requires-* category of the trait, but turns green because it is now *supplied* by the class in which the trait is used (see Figure 5.9). It is important that satisfied requirements remain visible: the list of required methods is a useful aid to understanding the dependencies inside a class, whether or not they have been satisfied. However, once all of the requirements



Figure 5.9: The glue method corner in RectangularMorph

have been satisfied, the class RectangularMorph and its component TRectangle change from blue to black, which indicates that the composition is now complete. Note that the required methods species and perform:with: were green from the first because RectangularMorph inherits an implementation for these methods from Object.

To assure that the composition is also correct, we now examine the places where we have overridden methods inherited from RectangleMorph with methods from trait TRectangle. These methods are listed in the *-overrides-* virtual category. The button bar in the browser (see Figure 5.10) lets the programmer view both the superclass and the trait methods for the currently selected message. Using these buttons, the programmer can easily view the various competing methods, and decide which is appropriate for the new class.

In the RectangularMorph example, most of the overrides provided by the trait are appropriate, but =, hash and printOn: are not. A browser menu (see Figure 5.10) gives us a choice of two ways to exclude these methods. “Set exclusion” modifies the composition clause of RectangularMorph so that the selected method (=) is excluded from the composition. If we were to use this menu item three times, for methods =, hash and printOn:, the browser would modify the definition of RectangularMorph to read as follows:

```
RectangleMorph subclass: #RectangularMorph
  uses: TRectangle - {#=. #hash. #printOn:}
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'TraitsPaperExample'
```

In these three cases, the more appropriate action is to *remove selector from trait TRectangle*. That is, we see that these methods should not be in the trait TRectangle at all: by removing them from the trait, they no longer override the inherited methods in RectangularMorph.

Note that as pointed out in Section 5.2.3, the browser lets us view and edit our new class in two ways. By selecting the name RectangularMorph in the browser, we can view it as a conventional Smalltalk class. Alternatively, by selecting its components -own- and TRectangle, we can view it as a composite entity. Editing a method, in either view, changes just the class RectangularMorph: even if the method originally came from the trait TRectangle, a modified version is created for the class. Deleting a trait method sets an exclusion, so that

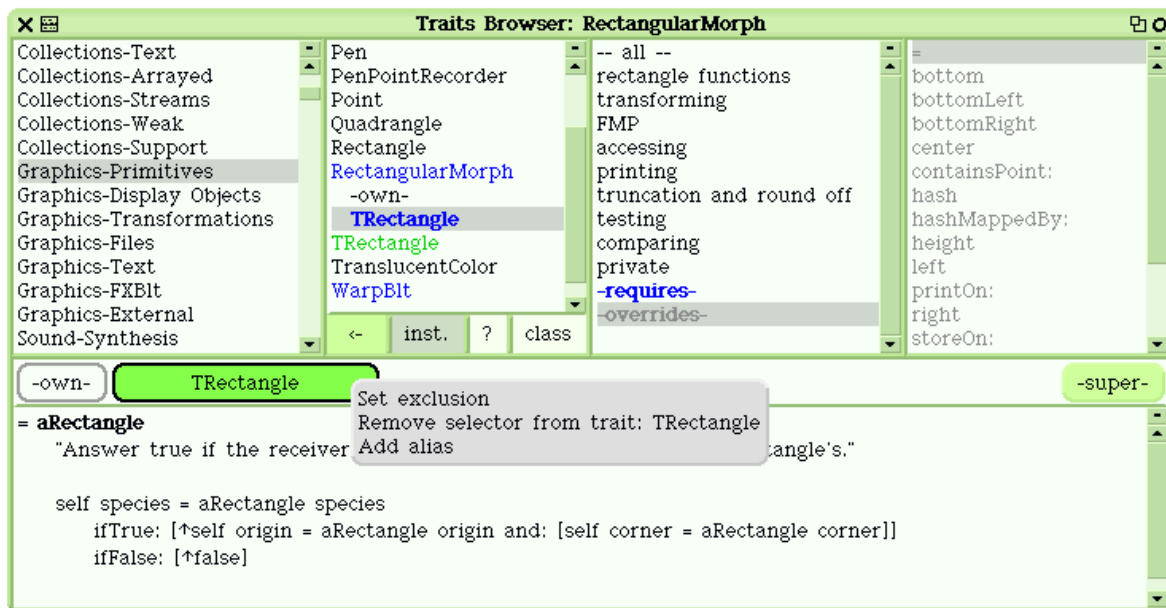


Figure 5.10: The method = in RectangularMorph.

the method is no longer part of the composite class. If instead the programmer wants to modify the component trait, by editing or deleting one of its methods, the browser must first be focussed on the trait.

Now RectangularMorph is complete, but our task is not yet finished, because we have duplicated all of the methods that we extracted from Rectangle when we constructed TRectangle. We can now simply eliminate this duplication by refactoring the Rectangle class so that it uses TRectangle as well.

### 5.3.3 Uncovering Hidden Structure

One of the more difficult tasks of program maintenance is discovering the latent structure hidden in the old code. The complete protocol of a class is usually the union of several smaller protocols, but programmers seldom make these component protocols explicit. The standard system browser allows the programmer to categorize methods into protocols, but this is mere documentation, and thus often wrong: getting the categorization right has no immediate payoff. Java provides interfaces and the implements declaration, but their use is optional: it is more common to program to the implicit interface of a whole class. The trait browser has proved to be a powerful tool for modularizing a class into separate traits, each of which implements a coherent protocol.

The first major programming task that we undertook with the browser was the refactoring of the Smalltalk collection classes into traits (cf. Chapter 6). This required us to discover the various protocols that were understood by the existing collection classes. We developed a methodology that involved two copies of the browser.

In the first browser we took a class from the existing hierarchy, say, Collection, and extracted a trait from it, which we called TempCollection, and which thus contained copies of all of the methods of Collection. We then moved (by dragging) the methods from TempCollection into an appropriate trait in the second browser. If an appropriate trait did not yet exist, we created one. For example, we might drag methods like collect: into a new trait called TCollEnumeration. The methods do: and species would then immediately be colored blue and appear in the *-requires-* virtual category.

The immediate updating of the *-requires-* category provided valuable feedback about the protocol that we were constructing. For example, if the method addAll: were mistakenly dragged into TCollEnumeration, then

add: would immediately show up in the *-requires-* category, providing a strong clue that something was amiss. If necessary, the menu item “*local senders of ...*” let us see why a particular method was required.

Sometimes we would realize that what we had imagined as one trait was actually two, in which case we would pause in our work while we split the trait. For example, we eventually split `TCollEnumeration` into two traits: `TCollEnumerationUI`, whose methods will work on unsequenced collections, and `TCollEnumerationI`, which contains `TCollEnumerationUI` as a subtrait, but which also includes some methods (like `findFirst:` and `from:to:do:`) that require the collection to be sequenced, and which thus have `at:` as a additional requirement. Making *-requires-* visible enables the programmer to see this distinction.

The process of discovering the latent protocols embedded in `Collection` proceeded until there were no more methods in `TempCollection`. At this point we could delete the temporary trait, confident that all of its methods had found a home in some trait or other.

If the new traits that are constructed by this process are to be reusable, the semantics of each of the required methods must be clear. The programmer can document these semantics by adding a comment describing the required behavior to the marker methods self requirement. The few cases in which we were unable to understand the required semantics represented design flaws in the original code. For example, we have already stated that several of the methods in `TCollEnumerationUI` required `species`. But so did `=` and `hash!` What is going on here is that `species` is actually playing two different roles. According to the “Blue Book” [GOLD 83], the message `species` should return the preferred class for reconstructing the receiver, and the expression `self species new` is therefore used within enumeration methods such as `collect:` or `select:` that create new collections. But according to LaLonde and Pugh [LALO 90], `species` also plays a critical role in equality comparisons: two collections must be of the same `species` if they are to be considered equal (`=`).

These two roles are not always compatible. The classes `Interval` and `Heap`, for example, implement `species` so that it returns the class `Array`, which means that intervals, heaps and arrays are considered equal if they contain the same set of elements. At the same time, the class `OrderedCollection` implements `species` to return itself, which means that an instance of `OrderedCollection` can never be equal to an instance of `Array`. Thus, part of our refactoring was to replace occurrences of `self species new` (the first role) with a self-send of the new message `emptyCopyOfSize`.

### 5.3.4 Traits and Agile Methodologies

In recent years, agile methodologies, in particular Extreme Programming (XP) [BECK 00], have begun to influence the software engineering process. Traits and the trait browser are compatible with several of the XP practices such as continuous design, constant refactoring, and testing.

**Continuous design.** Extreme programming suggests that there is no up-front design phase. Instead, design takes place incrementally throughout the development process: the design of a program is always subject to change. Traits support this style of programming because they provide an additional way to adapt a program to a design change. Specifically, in addition to refactoring the class hierarchy in the conventional way, traits allow one to factor out an arbitrary set of methods and then reuse them wherever it seems most appropriate.

Traits even allow one to start implementing before any design exists. This is because traits enable a *behavior-based* or bottom up strategy that is appropriate when it is clear that a certain behavior is needed, but not yet clear in what class it should be placed. The enumeration behavior already discussed is a typical example. Traits let us forge ahead and define as a trait the coherent set of methods that captures the appropriate behavior; we can defer the decision about where it should be placed in an inheritance hierarchy to best enable reuse.

Of course, it is still possible to adopt a *class-based* or top-down strategy: when it is clear that some behavior has to be in a class, we can still use all the familiar techniques of single-inheritance programming. We can just implement a class, or a small hierarchy of classes, as if traits did not exist. Later, we can structure the classes by dragging and dropping certain methods into traits. Perhaps this is just for documentation, but it can also

be essential if we eventually see that one of the classes contains some reusable behavior that we would like to share with an unrelated class.

**Refactoring** is the technology that makes continuous design feasible: “if you believe that the future is uncertain, *and that you can cheaply change your mind*, then putting in functionality on speculation is crazy” [BECK 00, page 57, our emphasis]. Refactoring is simplified by the presence of traits, because they enable us to move a whole group of logically related methods from one class to another with a single edit. By putting methods into traits, we keep our options open: if there later turns out to be an abstract superclass that is a suitable home for such behavior, then that superclass can use the trait, and its subclasses can inherit the corresponding behavior, without making it any harder for other, unrelated classes to also use the same behavior. This flexibility seems to have no cost in understandability: indeed, we argue that traits *increase* understandability. This is because critical behavior can be made explicit and given a descriptive name.

**Testing.** Traits allow one to specify tests in a very fine-grained and reusable manner. This is because tests can be associated with traits as well as with classes, and traits represent a smaller and more primitive unit of functionality. Even for very simple traits such as equality, magnitude, or emptiness, tests can be written very early, placed in the trait, and then applied to all of the classes that use the trait. Alternatively, the tests can be placed into separate traits. These “test traits” can then be used to create test cases (*i.e.*, subclasses of the class `TestCase`) for all the classes to which the corresponding traits are applied.

## 5.4 Interaction between Language, Tools and Methodology

Our experience with traits confirmed the principle known from Smalltalk, which says that programming language, tools and methodologies should be developed together: the best methodology is little more than pious hope without appropriate language features and supporting tools, and powerful tools can be powerfully dangerous without a guiding methodology.

In theory, the difference between traits on one hand and mixins and multiple inheritance on the other arise from the way in which we defined the composition operators on traits. These operators ensure that trait composition is commutative, that conflict resolution is always explicit, and that a composite entity can always be flattened into a simple, unstructured one.

However, our experience has shown that without appropriate tools the programmer cannot really take advantage of these theoretical advantages. For example, the programmer can only benefit from the flattening property if there are tools that actually provide the different views (*i.e.*, the flat class-based view and the structured trait-based views) and allow the programmer to switch between them. Similarly, the requirement for explicit conflict resolution is rather painful if there are no tools that indicate all conflicts as soon as they occur.

Once we had implemented the trait browser, the tool that provided us with all these important features, we also realized that it encouraged and helped the development of a sound methodology. While traits as a language feature made it possible to factor out protocols as individual traits, it was the trait browser that actually helped us identify and understand the hidden protocols, and that made it easy and pleasant to perform the corresponding refactorings.

## Chapter 6

# Case Study: Refactoring the Smalltalk Collection Classes

We performed two case studies to evaluate the practical benefits of traits in realistic scenarios. The first case study, presented in this chapter, is a refactoring of the Smalltalk collection hierarchy. We identified this hierarchy as particularly appropriate for such a case study because its classes can be categorized in so many dimensions that the traditional implementation suffers from problems related to code reuse and conceptual categorization [COOK 92]. These are exactly the kinds of problems for which traits are designed, and it is therefore interesting to see whether and how they can solve these problems in practice.

We start by giving a brief overview of the design and organization of the Smalltalk collection classes (Section 6.1). Then, we analyze the deficiencies of the traditional implementation (Section 6.2) and present our refactored implementation that uses traits to avoid these deficiencies (Section 6.3). We discuss the lessons we learned from this refactoring and elaborate on the differences between our refactoring and alternative refactorings based on approaches such as mixins and multiple inheritance (Section 6.4). Finally, we conclude this chapter by summarizing and interpreting our results (Section 6.5)

### 6.1 The Smalltalk Collection Classes

The collection classes are a loosely defined group of general purpose subclasses of `Collection` and `Stream`. The group of classes that appears in the “Blue Book” [GOLD 83] contains 17 sub-classes of `Collection` and 9 sub-classes of `Stream`, for a total of 28 classes, and had already been redesigned several times before the Smalltalk-80 system was released. This group of classes is often considered to be a paradigmatic example of object-oriented design.

In Squeak, the abstract class `Collection` has 98 subclasses, and the abstract class `Stream` has 39 subclasses, but many of these (like `Bitmap`, `FileStream` and `CompiledMethod`) are special purpose classes and hence not categorized as “collections” by the system organization. For the purposes of this study, we use the term *collection hierarchy* to mean `Collection` and its 37 subclasses that are *also* in the system category `Collections`. We use the term *stream hierarchy* to mean `Stream` and its 10 subclasses that are *also* in the system category `Collections`.

The full list of collection classes is shown in Figure 6.1. Indentation indicates subclassing. Abstract classes are *italicized*; the classes that we refactored are named in **bold**. These 49 classes respond to 794 messages and define a total of 1236 methods.

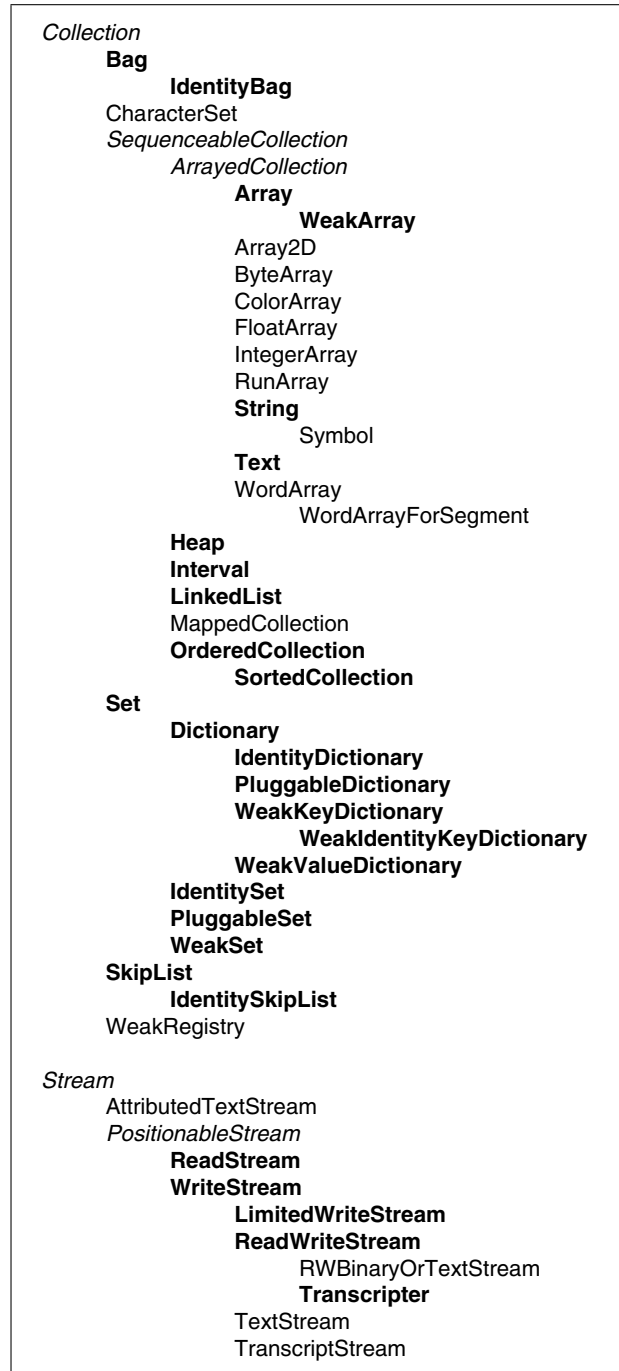


Figure 6.1: The collection classes in Squeak

| Sequenceable                                                                                |                        | Not Sequenceable                                        |                                                          |
|---------------------------------------------------------------------------------------------|------------------------|---------------------------------------------------------|----------------------------------------------------------|
| Accessible by Index                                                                         | Not Indexable          | Accessible by Key                                       | Not Keyed                                                |
| Interval<br>SortedCollection<br>Array<br>ByteArray<br>OrderedCollection<br>String<br>Symbol | LinkedList<br>SkipList | Dictionary<br>IdentityDictionary<br>PluggableDictionary | Set<br>IdentitySet<br>PluggableSet<br>Bag<br>IdentityBag |

Figure 6.2: Some collection classes categorized by functionality

### 6.1.1 The Varieties of Collection

To understand the challenge of refactoring the collection hierarchy, the reader needs at least a superficial knowledge of the wide variety of collections in these classes, their commonalities and their differences. Those familiar with the Smalltalk collection classes may safely skip this section.

Programming with aggregates rather than individual elements is an important way of raising the level of abstraction of a program. The Lisp function `map`, which applies an argument function to every element of a list and returns a new list containing the results, is an early example of this style, but Smalltalk-80 adopted aggregate-based programming as a central tenet. Modern functional programming languages such as ML and Haskell have followed Smalltalk's lead. So, why is this a good idea?

Suppose you have a data structure containing a collection of student records, and wish to perform some action on all of the students that have a particular property. Programmers raised to use an imperative language will immediately reach for a loop. But the Smalltalk programmer will write

```
students select: [ :each | each gpa < threshold ]
```

which evaluates to a new collection containing precisely those elements of students for which the bracketed function returns true<sup>1</sup>.

It is important to note that the message `select:` is understood by *all* collections in Smalltalk. There was no need to find out if the student data structure was an array or a linked list: the `select:` message is understood by both. Note that this is quite different from using a loop, where one must know whether students is an array or a linked list before the loop can be set up.

In Smalltalk, when one speaks of a collection without being more specific about the kind of collection, one means an object that supports well-defined protocols for testing membership and enumerating the elements. *All* collections understand the testing messages `includes:`, `isEmpty` and `occurrencesOf:`. *All* collections understand the enumeration messages `do:`, `select:`, `reject:` (which is the opposite of `select:`), `collect:` (which is like Lisp's `map`), `detect:ifNone:`, `inject:into:` (which performs a left fold) and many more. It is the ubiquity of this protocol, as well as its variety, that makes it so powerful.

Beyond this basic uniformity, there are many different kinds of collection. This is illustrated in Figure 6.2, which shows a few collection classes and how they are categorized according to their functionality. `Array` and `LinkedList` are examples of collections that are sequenceable, which means that an enumeration of the collection starts from a first element and proceeds in a well-defined order to a last element.

`Array` and many other collections are also indexable, that is, `anArray at: n` retrieves the  $n^{th}$  element of `anArray`, and `anArray at: n put: v` changes the  $n^{th}$  element to `v`. However, `LinkedList`, although sequenceable, is not indexable, that is, its instances understand `first` and `last`, but not `at:`. Of the non-sequenceable collections, dictionaries can be accessed by an arbitrary key, such as a string, while sets and bags cannot.

<sup>1</sup>The expression in brackets can be thought of as a  $\lambda$ -expression defining an anonymous function  $\lambda x. x \text{ gpa} < \text{threshold}$ .

| Arrayed Implementation    | Ordered Implementation                                | Hashed Implementation                                                                                               | Linked Implementation  | Interval Implementation |
|---------------------------|-------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------|------------------------|-------------------------|
| Array<br>String<br>Symbol | OrderedCollection<br>SortedCollection<br>Text<br>Heap | Set<br>IdentitySet<br>PluggableSet<br>Bag<br>IdentityBag<br>Dictionary<br>IdentityDictionary<br>PluggableDictionary | LinkedList<br>SkipList | Interval                |

Figure 6.3: Some collection classes categorized by implementation technique

The class `OrderedCollection` is more general than `Array`; the size of an `OrderedCollection` grows on demand, and it has methods for `addFirst:` and `addLast:` as well as `at:` and `at:put:`. An `Interval` is an immutable collection defined by a computational rule when it is created. For example, `5 to: 16 by: 2` is an interval that contains the elements 5, 7, 9, 11, 13 and 15. It is indexable with `at:`, but cannot be changed with `at:put:`.

### Functional Differences

The differences between the various kinds of sequenceable collection manifest themselves in several different dimensions.

1. How is the order established? Sorted collections use a supplied total ordering function, intervals are implicitly ordered, while arrays and ordered collections are ordered explicitly when elements are inserted.
2. Is the size fixed (intervals and arrays) or variable (sorted collections, ordered collections, and linked lists)?
3. Is the collection immutable (intervals and symbols) or mutable (the others)?
4. Is the collection constrained to hold a particular kind of object, or is it completely general? For example, `LinkedLists` are constrained to hold elements that conform to the Link protocol, while `CharacterArrays`, `Strings` and `Symbols` must contain characters.<sup>2</sup>

The non-sequenceable collections (sets, bags and dictionaries) can be categorized in a different set of dimensions.

1. Are duplicates allowed (dictionary and bag) or disallowed (set)?
2. Can the elements be accessed by a key (dictionaries), or not (sets and bags)?
3. How are the keys (in a dictionary) or the values (in a set or a bag) compared, *e.g.*, what test is used to ascertain whether two elements added to a set are “equal”? For example, `Dictionary`, `Set` and `Bag` use the `=` method provided by the elements; the `Identity` variants of these classes use the `==` method, which tests whether the arguments are the same object, and the `Pluggable` variants use an arbitrary equivalence relation supplied by the creator of the collection.

### Implementation Differences

These categorizations by functionality are not our only concern; as re-implementors of the collection hierarchy we must also understand how the collection classes are implemented. As shown in Figure 6.3, five main implementation techniques are employed.

<sup>2</sup>A `Symbol` is a unique instance of an immutable string, used heavily in the language implementation. The unique instance property means that equality tests are particularly efficient.

1. Arrays store their elements in the (indexable) instance variables of the collection object itself; as a consequence, arrays must be of a fixed size, but can be created with a single memory allocation.
2. OrderedCollections and SortedCollections store their elements in an array that is referenced by one of the instance variables of the collection. Consequently, the internal array can be replaced with a larger one if the collection grows beyond its storage capacity.
3. The various kinds of Set and Dictionary also reference a subsidiary array for storage, but use the array as a hash table. Bags use a subsidiary Dictionary, with the elements of the bag as keys and the number of occurrences as values.
4. LinkedLists use a standard singly-linked representation.
5. Intervals are represented by three integers that record the bounds and the step size.

In addition to these classes, there are also “weak” variants of Array, Set and of the various kinds of dictionary. These collections hold onto their elements weakly, *i.e.*, in a way that does not prevent the elements from being garbage collected.

Readers interested in learning more about the Smalltalk collections are referred to LaLonde and Pugh’s excellent book [LALO 90].

### 6.1.2 Streams

The collection protocol supports the storage, removal and enumeration of the elements of a collection, but does not allow these operations to be intermingled. For example, if the elements of an OrderedCollection are processed by a `do:` method, it is not possible to add or remove elements from inside the `do` block. Nor does the collection protocol allow us to perform a merge sort by sequencing through two OrderedCollections and repeatedly removing the smallest first element. Procedures like these require that a traversal index or position reference is maintained outside of the collection itself, as captured in the iterator pattern [GAMM 95]. Smalltalk Streams perform exactly this function.

All stream objects are defined to *stream over* some collection. For example:

```
r := ReadStream on: (1 to: 10).
r next.           "evaluates to 1"
r next.           "evaluates to 2"
r atEnd.          "evaluates to false"
```

WriteStreams are analogous:

```
w := WriteStream on: (String new: 5).
w nextPut: $a.
w nextPut: $b.
w contents.      "evaluates to 'ab'"
```

It is also possible to create ReadWriteStreams that support both the reading and the writing protocols; defining such a class without code duplication is a challenge for single inheritance. Squeak chooses to make ReadWriteStream a subclass of WriteStream, as shown in Figure 6.1.

## 6.2 Analysis of the Collection Classes

This section presents the results of an analysis of the collection hierarchy as it existed before our refactoring. We will see that the collection hierarchy contains unnecessary inheritance, duplicated code, and other shortcomings.

Given the many dimensions in which the Smalltalk collection classes can be categorized, it is inevitable that any attempt to organize them into a single inheritance hierarchy will run into severe difficulties. As Cook

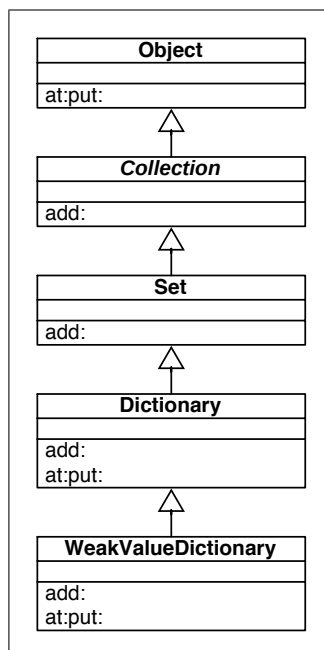


Figure 6.4: Redefinition of `at:put:` and `add:` in the superclasses of `WeakValueDictionary`

showed [COOK 92], the hierarchy attempts to maximize reuse at the expense of conceptual categorization, with the consequence that, for example, `Dictionary` is a subclass of `Set` because it shares much of the same implementation, even though it presents a very different interface.

Another way that the designers of the hierarchy attempted to maximize reuse was to move methods high up, so that all possible classes have a chance to inherit them. For example, `collect:` is implemented in `Collection`, but the implementation is appropriate only for those collections that understand `add:`. Consequently, this implementation is overridden by the abstract class `SequenceableCollection` in favor of an implementation using `at:put:`. This second implementation is inherited by all of the `ArrayedCollections`, but also by `OrderedCollection` and `SortedCollection`, for which it is not appropriate, and which override it again. All told, there are 9 implementations of `collect:` in the collection hierarchy in Squeak 3.2, whereas the refactored hierarchy contains only 6 of them.

The following subsections examine these effects more systematically.

### 6.2.1 Unnecessary Inheritance

Inheritance is used quite heavily in the collection classes, mostly for sharing implementation, but also for classification [COOK 92]. As a measure of the complexity of the inheritance relationships, we counted the number of inheritance chains in which a method is defined three or more times. Figure 6.4 illustrates two examples: the methods `at:put:` and `add:` in the inheritance chains terminating in the class `WeakValueDictionary`. We found 79 such inheritance chains.

There is nothing intrinsically wrong with redefining a method inherited from one's superclass. On the contrary, the ability to use `super` to call the inherited definition from within the new method gives inheritance much of its power, and many people consider that adding behavior before or after a `super-send` is the epitome of inheritance-oriented programming.

However, for the most part, redefinition using `super` is *not* what is going on here. A total of 258 methods are involved in the 79 method redefinition chains mentioned above. Since 79 methods are at the top of a

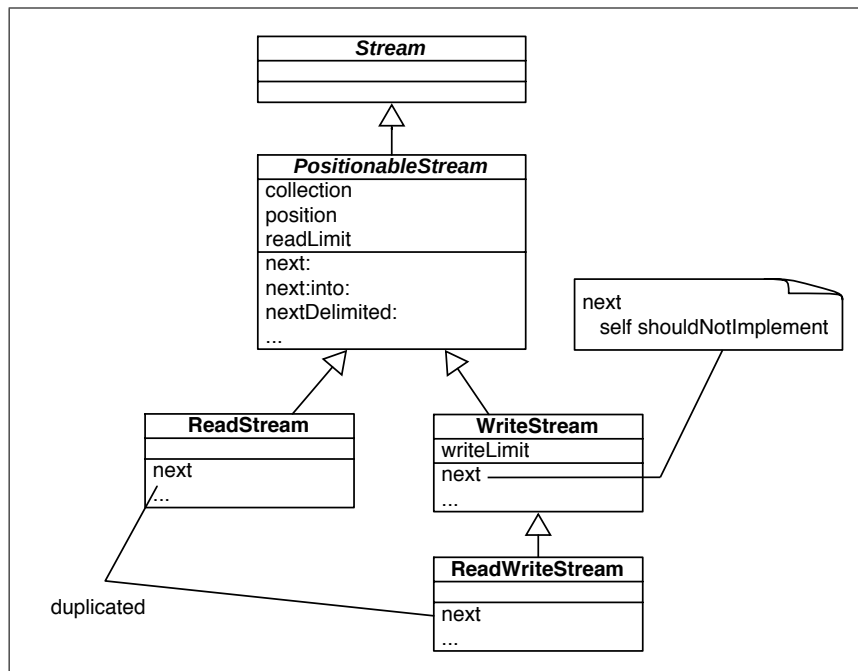


Figure 6.5: Duplicated methods and methods implemented “too high” in the stream hierarchy

chain,  $258 - 79 = 179$  methods have the opportunity of sending to super: only 15 actually do so. Neither are these redefinitions examples of “hook” methods that are being used to parameterize the behavior of a template method [ALPE 98]: all of the redefined methods are part of the functional interface. We deduce that for the most part these redefinitions are *correcting*, rather than augmenting, the behavior of the inherited method so that it is appropriate for the new subclass. In other words, we have identified 164 places where a method was inherited unnecessarily.

What is the problem with unnecessary inheritance? The cost is not in execution time nor in code space but in lost development time. The task of understanding a class that inherits several methods but does not use them is more complicated than necessary. Inheritance is often considered to be an aid to understanding a complex class, since the programmer can work down the inheritance chain, comprehending only the *differences* between a subclass and its superclass, rather than having to comprehend the entirety of the final subclass in a single step. To the extent that methods are inherited unnecessarily, this process is made more difficult, and inheritance begins to hinder rather than to assist us in understanding legacy code.

### 6.2.2 Code Duplication and Inappropriate Hierarchies

When a new subclass *does* want to re-use a method from an existing class, it may nevertheless be unable to do so because of the nature of single inheritance. For example, `PluggableSet` and `PluggableDictionary` share some methods, but there is no place from which both classes could inherit them. `PluggableDictionary` is a subclass of `Dictionary`, and `PluggableSet` is a subclass of `Set`; there is no appropriate common superclass in which methods shared by the two pluggable classes can be placed. There is an *inappropriate* superclass: `Set`. The programmer is left with the choice of placing a method “too high” in the hierarchy (in `Set`), or duplicating it.

The stream classes provide a classic example of methods being implemented too high, which is illustrated in Figure 6.5. Conceptually, the abstract class `PositionableStream` provides the facilities for accessing a sequence of objects named by external indices (such as characters in a string or a file). The *positionable* protocol includes messages to set and reset the position of this index. However, `PositionableStream` is also the lowest common

superclass of both `ReadStream` and `WriteStream`. Because `ReadWriteStream` is a subclass of `WriteStream`, many of the methods that support reading (for example `next:`, `next:into:` and `nextDelimited:`) are implemented in `PositionableStream` rather than in `ReadStream`. All these “reading methods” are defined in terms of the method `next`, which is a requirement of `PositionableStream` and is implemented in its subclass `ReadStream`. In the subclass `WriteStream`, the method `next` is explicitly disabled, and it is then re-enabled in `ReadWriteStream`, which implements a duplicate of the method `next` defined in `ReadStream`.

The other reading methods (such as `nextDelimited:` and `next:into:`) are directly implemented in `PositionableStream` and are *not* explicitly disabled in `WriteStream`, which means that they are inherited by `ReadStream`, `WriteStream`, and `ReadWriteStream`. Thus, to avoid code duplication, methods specific to reading are implemented in the common superclass of `ReadStream` and `WriteStream`. The tactic succeeds (except in the case of `next`), but the price is high: `PositionableStream` is polluted by many methods that have nothing to do with positioning, and `WriteStream` appears to implement many reading methods, although these methods will fail if they are ever used.

There is no easy way to ascertain how much duplication is caused by the fact that methods can be inherited only from a superclass. We made a superficial check by looking for methods whose decompile strings were identical. This check detected as duplicates methods those that differed only in formatting, comments, or the names of temporary variables. We excluded from our count error marker methods such as `self shouldNotImplement`, which is used to cancel an inherited method. Using this check we found 28 pairs of duplicated methods and 3 triples in the collection classes.

In the majority of cases the duplication was of a method defined in another part of the class hierarchy, which consequently could not be inherited, or of an unreachable method defined in an indirect superclass. For example, the classes `Dictionary` and `Collection` both implement the method `occurrencesOf:` identically, but even though `Dictionary` is a subclass of `Collection`, there is an intervening definition of `occurrencesOf:` in the class `Set` that prevents `Dictionary` from reusing the method from `Collection`.

However, these duplication counts are just the tip of the iceberg. Our primitive duplicate detection technique certainly misses many methods that differ in structure but not in semantics. For example, if two methods compare `x` and `y` for equality, but one expresses this as `x = y` while the other uses `y = x`, this duplication will not be revealed by our search. During our refactoring of the collection classes we also noticed many deeper examples of code duplication, where a method had clearly been copied from an established class into a newly created class, and then a single crucial statement had been changed to obtain a different semantics. In addition, there is also undoubtedly duplication of collection code in classes outside of the collection hierarchy, which we did not attempt to quantify.

### 6.2.3 Conceptual Shortcomings

In addition to the above implementation problems, the collection classes also suffer from some conceptual shortcomings.

One of the reasons that there are so many collection classes is that the designers have attempted to compensate for the fact that classes are hard to reuse by providing all possible combinations of features. For example, sets, bags and dictionaries must compare elements (or keys) for equality. Thus, each structure needs three variants: one that uses equality (`=`) between elements, one that uses identity (`==`), and one that uses an equality function that is “plugged in” when the structure is created. Thus, we have the three classes `Set`, `IdentitySet`, and `PluggableSet`; the same is true for `Dictionary` and `Bag`, except that `PluggableBag` is missing. A similar situation exists with the “weak” variants of the collection classes, which hold onto their elements weakly, *i.e.*, in a way that does not prevent them from being garbage collected.

It would be nice if these characteristics could be captured as reusable components, so that programmers could combine pluggability with, say, `SkipLists`, so that they could build the data structure that suits their application. This would simultaneously simplify the collection hierarchy (by eliminating the combinatorial explosion of features) and give the programmer the flexibility to choose from a wider range of collections.

Immutability is a “feature” not provided in the current hierarchy except in two special cases: symbols and

intervals. Nevertheless, immutable collections can be useful in many contexts. Strings are almost always used as if they are immutable, as are literal arrays, but this cannot be captured by the current collection classes.

The stream classes also exhibit many orthogonal features, such as read vs. write, binary vs. text, positionable (seekable) vs. not-positionable. The more necessary combinations are implemented by duplicating code; many other combinations are simply unavailable.

Another problem with the collection hierarchy, also observed by Cook, is that sometimes the interfaces of the classes are not what one would expect: certain methods are missing. The classes `String` and `Text` provided an example. `String` adds 142 new methods to the protocol of its superclass (the abstract class `ArrayedCollection`). Most of these methods are related to parsing, converting to HTML, searching for substrings and regular expressions, and other operations specific to character strings, and so inevitably these methods must be defined specifically for the class `String`. However, Squeak also defines a class `Text`, for representing character strings that have been attributed with font changes, hyperlining, *etc.* All 142 `String` messages ought to be understood by `Text` objects, but in the standard Squeak system, only 15 of them actually are. The remaining 127 are missing. Why is this? `Text` is not a subclass of `String`, so the “missing” methods cannot be inherited; the situation is similar to the problem with `RectangleMorph` and `Rectangle` described in Section 5.3.2. Fixing this problem in Squeak would require either code duplication or 127 delegation methods.

Finally, we mention that collection-like behavior is often desired for objects that are not primarily collections. For example, the class `Path` is a subclass of `DisplayObject` and thus not able to inherit from `Collection`. A `Path` represents an ordered sequence of points; arcs, curves, lines and splines are all implemented as subclasses of `Path`. `Path` implements some of the more basic kinds of collection-like behavior; for example, it has methods for `at:`, `at:put:`, and `collect:`. But `Path` does *not* attempt to implement the full range of collection behavior. For example, `Path` does not provide methods for `select:` and `do:`: there are simply too many such methods to make it viable to re-implement them, and the existing implementation cannot be reused. Section 6.3.4 discusses how traits make collection-like behavior available outside of the collection hierarchy.

## 6.3 Refactoring Results

In this section we explain how traits are used in the collection hierarchy that emerged from our refactoring efforts. We start by describing how we distributed behavior from the pre-existing abstract and concrete classes into traits, and how those traits are used to construct a new set of classes. We then analyze the new hierarchy with respect to code duplication, possibilities for reuse, and other issues.

### 6.3.1 The New Collection Hierarchy

Figure 6.6 shows the new hierarchy for the 23 common concrete collection classes that we have re-implemented, and 6 abstract superclasses. Classes with italicized names are abstract; below the class name we show the top-level trait(s) from which the class is composed. Each of these traits is in turn composed from several subtraits, as shown in Figures 6.7 and 6.8. The names of implementation traits end in “Impl”. Individual methods and instance variables are not shown.

The classes in the refactored hierarchy are divided into three layers. At the top of the Figure 6.6 is the abstract class `Collection`, which is composed from two traits, and provides a small amount of behavior for all collections. Next we have a layer of 5 abstract classes that represent different combinations of the externally visible properties of collections. We call these properties *functional*, to distinguish them from the *implementation* properties, that is, properties that characterize the internal data structures used in the implementation rather than the external behavior. Inheriting from the functional classes we have 23 concrete classes, each of which also uses one or more traits that specifies its implementation. We now describe the functional and the implementation traits in turn.

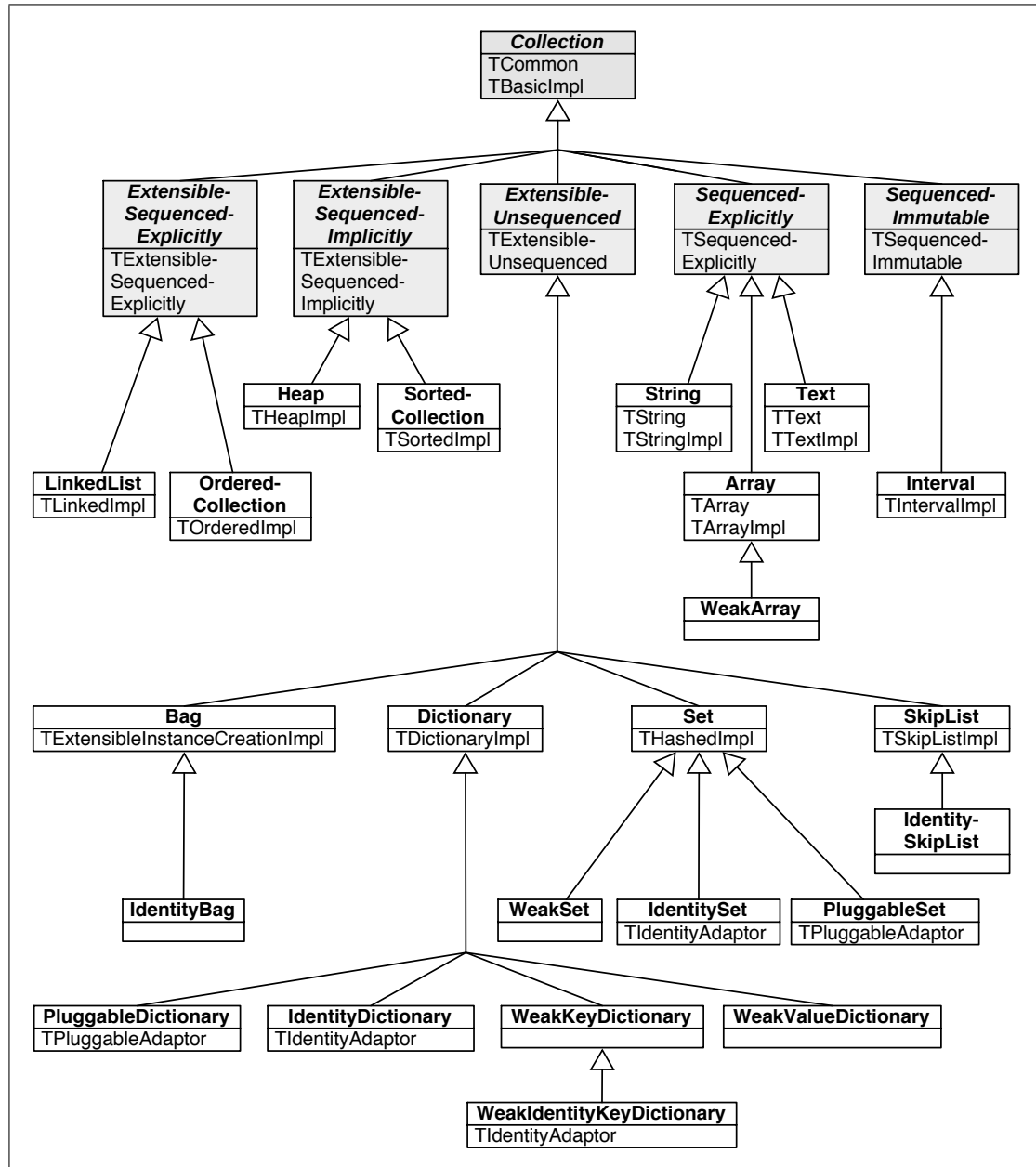


Figure 6.6: The refactored collection hierarchy

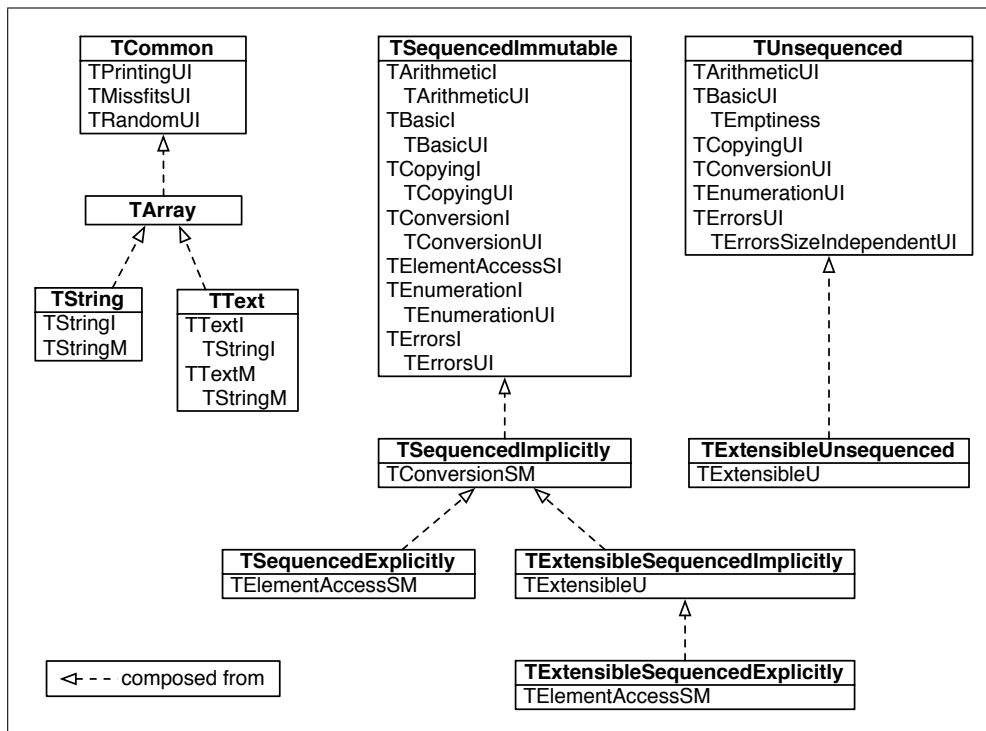


Figure 6.7: The hierarchy of traits corresponding to functional properties

### The Functional Traits

Each kind of collection can be characterized by several functional properties such as being explicitly ordered (e.g., Array), implicitly ordered (e.g., SortedCollection), unordered (e.g., Set), extensible (e.g., Bag), immutable (e.g., Interval), or keyed (e.g., Dictionary); see Section 6.1.1 for more discussion. The various combinations of these properties can be represented by combining the respective traits. All that is necessary is to create a trait for each functional property and then combine them to build the abstract classes of Figure 6.6.

Figure 6.7 shows how the functional traits are built from each other and from more primitive traits. The 11 boxes represent the larger composite traits; subtrait relationships between them are shown by arrows. For clarity, we do not show the primitive traits graphically. Instead we indicate their usage in the bottom part of the rectangles representing the traits, and use indentation to indicate nested traits.

We use the following naming convention. Some names have a suffix consisting of letters from the sets {S, U} and {M, I}. The letter S indicates that all of the methods in the trait require the collection to be sequenced, whereas U means that none of the methods in the trait requires the collection to be sequenced. Similarly, M means that all the methods require the collection to be mutable, and I means that no method requires the collection to be mutable. If the suffix does not contain a letter from one of these sets, the trait contains some methods with each characteristic.

As an example, the trait TEnumerationUI contains the part of the enumeration behavior that does not require sequencing (U), whereas TEnumerationI — which uses TEnumerationUI as a subtrait — contains both methods that require sequencing and methods that are indifferent to whether the collection is sequenced. Furthermore, none of the methods in these traits treats the target object as mutable (I).

To allow maximal reuse, we made the combinations of property traits available in two forms: as composite traits that can be reused outside of the collection hierarchy, and as superclasses that can be inherited within it.

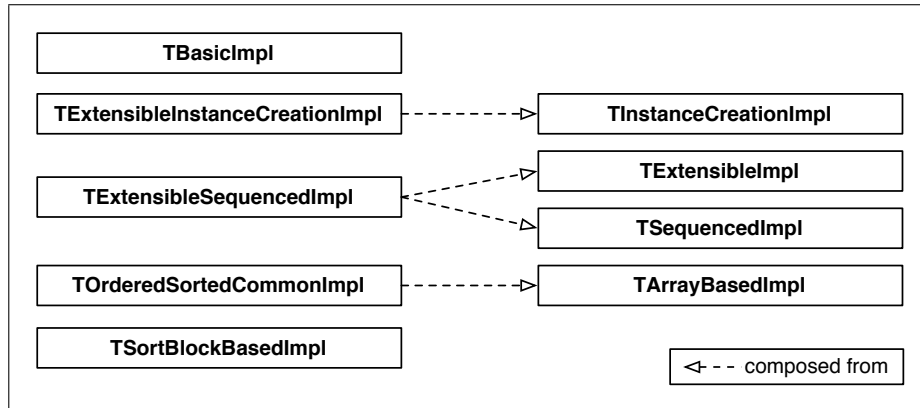


Figure 6.8: Common implementation traits

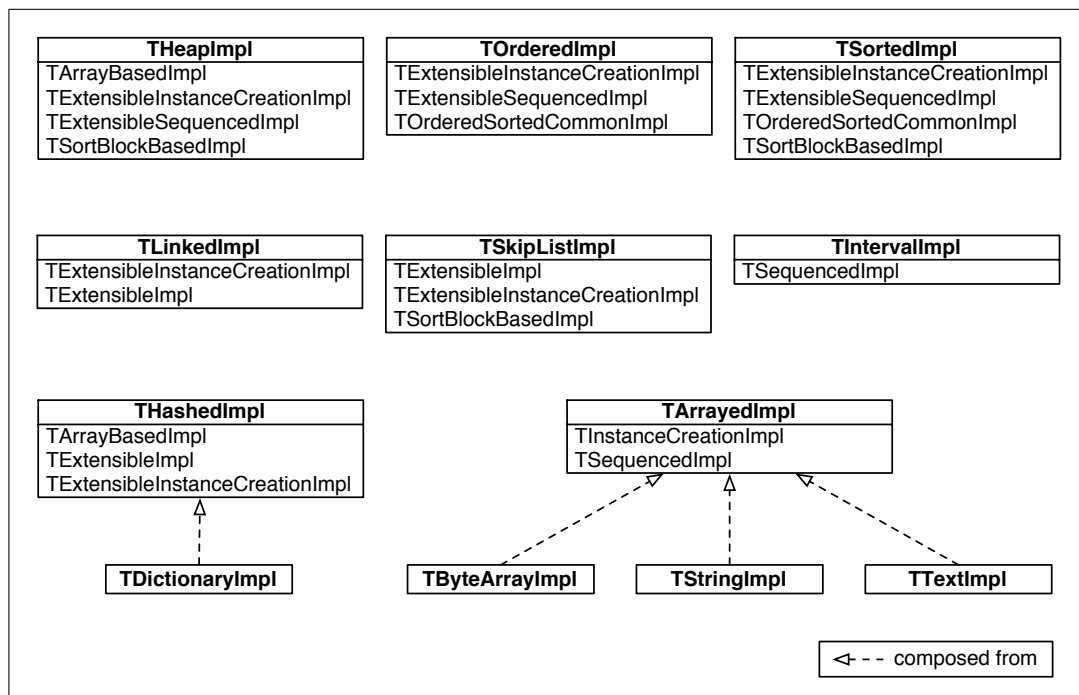


Figure 6.9: Specific implementation traits

We modularized the primitive properties more finely than would have been necessary if our only goal were to avoid code duplication. This results in a fine-grained structure that gives us, and future programmers, more freedom to extend, modify and reuse parts of the new hierarchy. In addition, some of the property traits contain many methods, and creating subtraits corresponding to individual sub-properties gives them internal structure that makes them easier to understand. For example, the composite trait `TUnsequenced` is built from 7 sub-properties such as `TArithmeticUI`, `TCopyingUI`, `TConversionUI`, and `TEnumerationUI`. Because of the flattening property, there is no cost to this fine-grained structure: it is always possible to flatten it out and to work with the code in a less structured view.

### The Implementation Traits

Besides the functional properties, which are visible to a client, each collection class is also characterized by an implementation, which is normally hidden from a client. The functional and implementation traits that capture these properties are largely independent.

The refactored hierarchy separates the traits specifying the implementation of a collection from the traits specifying the functional properties. This allows us to combine the different functional property traits (*e.g.*, `TExtensibleSequencedExplicitly` and `TExtensibleSequencedImplicitly`) with any of the suitable implementations (*e.g.*, linked and array-based). In one place we also extended the functionality of a concrete class: our `LinkedLists` are indexable, *i.e.*, they understand `at:`.

Figures 6.8 and 6.9 show the structure of the implementation traits. The 9 implementation traits shown in Figure 6.8 are “common” in the sense that they are used as components for several of the specific traits shown in Figure 6.9. As an example, the behavior for creating new instances (`new`, `with:`, `withAll:`, *etc.*) is collected into the trait `TInstanceCreationImpl`, which is then used on the class-side of `TOrderedImpl` and four other implementation traits. Each of the 12 traits in Figure 6.9 captures the behavior of a specific concrete class, and is built from a combination of local methods and the common implementation traits.

The trait `TBasicImpl` contains default implementations for methods like `includes:` and `hash`. These defaults are written so as to be independent of the implementation of the underlying collection, but may be unnecessarily slow for certain implementations. For example, `includes:` is implemented using `anySatisfy:`; this is always correct, but is  $O(n)$ , whereas in hashed collections `includes:` should be  $O(1)$ . Instead of using `TBasicImpl` as a subtrait of all the specific implementation traits, we decided to use it in the root class of the collection hierarchy, from where its methods are inherited (and possibly overridden) by the various implementations. For example, `THashedImpl` and `TIntervalImpl` have their own implementations of `includes:`.

### 6.3.2 The New Stream Hierarchy

The refactored version of the Stream hierarchy retains the abstract class `PositionableStream` from the standard Squeak hierarchy, but uses it only to capture the notion of positionability, and not as a place to put methods just so that they can be shared. Thus, the protocol of `PositionableStream` is reduced from 84 messages to 29. `Read-WriteStream`, which was formerly a subclass of `WriteStream`, is now a direct subclass of `PositionableStream`, and shares traits with both `ReadStream` and `WriteStream`, as shown in Figure 6.10.

Seven traits are used to build the Stream classes; they have the simple structure shown in Figure 6.11. `TReadablePositionable` and `TWritablePositionable` actually add *different* sets of methods to `TReadable` and `TWritable` because “positionability” means different things for readers and writers.

### 6.3.3 Measurements of the Refactored Classes

The refactored part of the collection class hierarchy shown in Figure 6.6 contains 23 concrete classes and 6 abstract classes. These classes are built from a total of 60 traits. The stream hierarchy shown in Figure 6.10 contains 7 concrete classes and 2 abstract classes, which are composed from 7 traits.

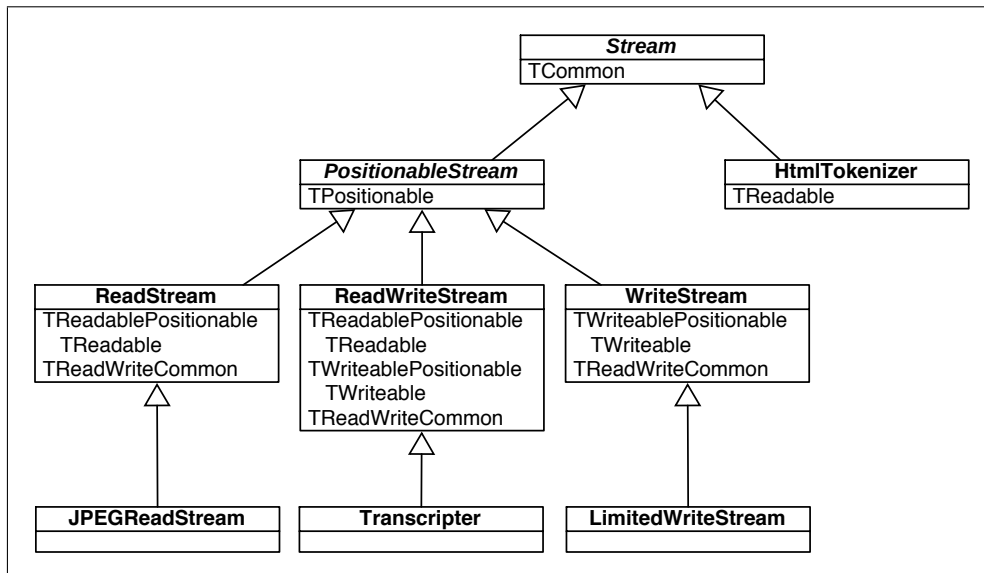


Figure 6.10: The refactored Stream hierarchy.

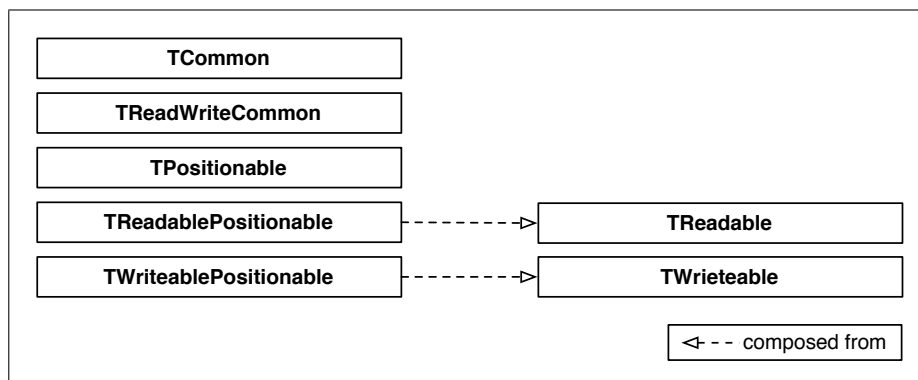


Figure 6.11: Traits used to build the Stream classes.

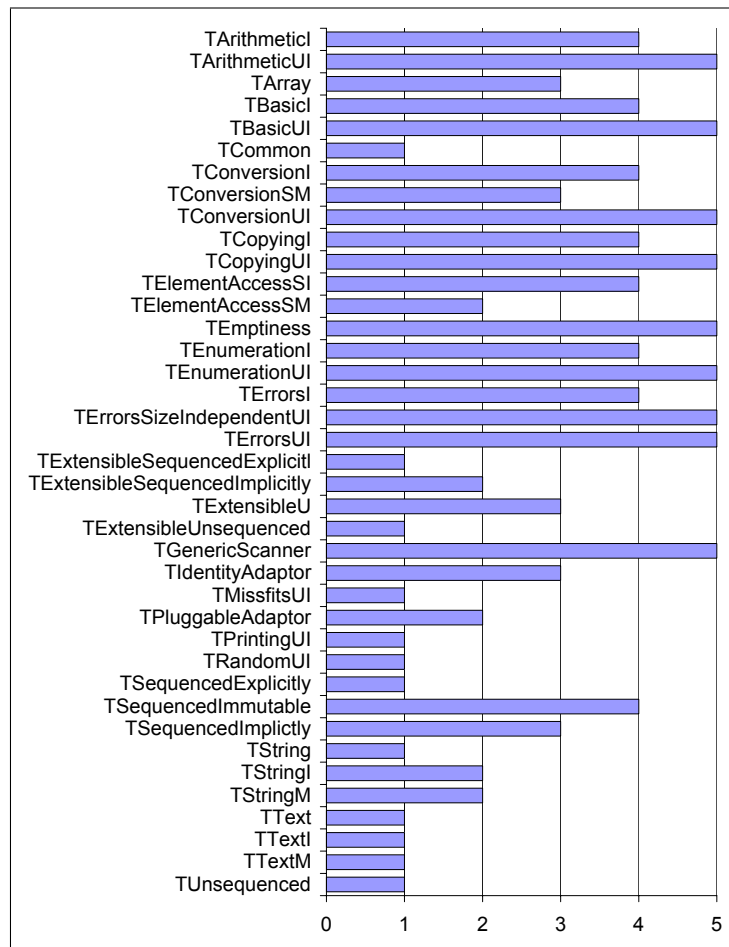


Figure 6.12: Usage histogram of the functional traits in the Collection classes

The average number of traits used to build a class is more than 5; the maximum that we used in any one class is 22. The average number of classes that use a certain trait is 2.8, which shows that a significant amount of reuse that occurs in the refactored hierarchies is based on trait composition rather than inheritance.

The histograms in Figures 6.12, 6.13, and 6.14 illustrate for each trait the number of classes using it. Figure 6.12 shows the functional traits used in the Collection hierarchy. While there are traits such as `TArithmeticUI` and `TCopyingUI` that are used by up to 5 different classes, there are also traits such as `TCommon` and `TText` that are only used by a single class. While the latter traits do not currently contribute to code reuse, they make the code more structured and facilitate future reuse.

The histogram shown in Figure 6.13 covers the implementation traits in the Collection hierarchy. It shows that the trait `TInstanceCreationImpl` is used by 11 different classes, while many other implementation traits such as `TBasicImpl`, `THeapImpl`, and `TLinkedImpl` are currently only used for a single class. Finally, Figure 6.14 shows the same data for the traits used in the Stream hierarchy, where the maximum number of classes that use a single trait is 3.

Further statistics are presented in Tables 6.1 and 6.2. Because `String` is something of an anomaly, containing many methods that do not appear elsewhere in the hierarchy (as explained in Section 6.2.3), we initially excluded `String` and `Text` from our measurements (see the first column of Tables 6.1 and 6.2). Most of the numbers are self-explanatory. “Methods saved” is the difference between the number of methods in the original

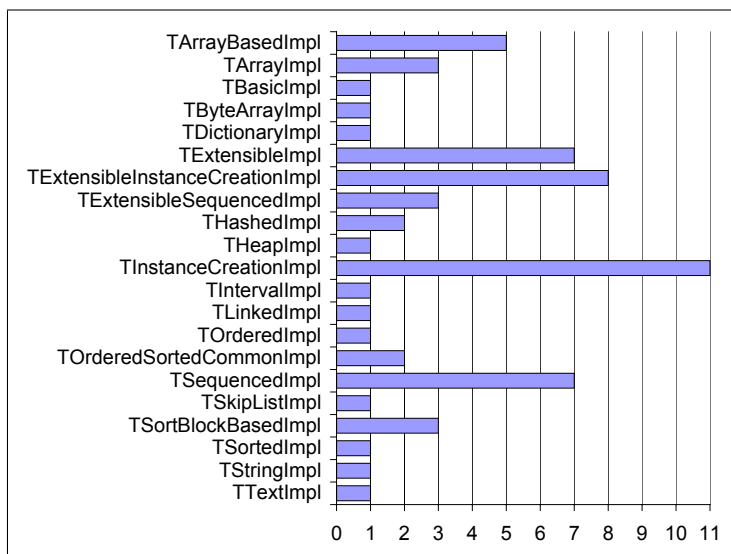


Figure 6.13: Usage histogram of the implementation traits in the Collection classes

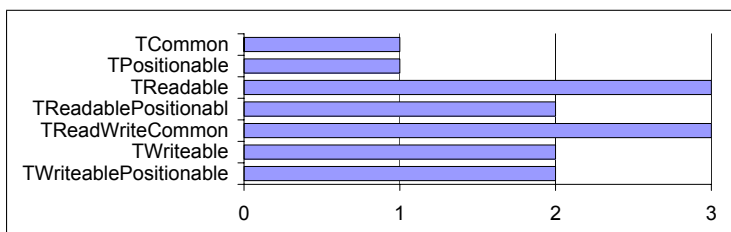


Figure 6.14: Usage histogram of the traits in the Stream classes

and the trait versions of the subject code. “Source code saved” is the difference in the size of the two versions, measured in bytes, and computed by decompiling the methods. This excludes comments and automatically adjusts for differences in formatting, naming of variables, and so on.

The phenomenon of methods being implemented “too high” was mentioned briefly in Section 6.2.2. Suppose that one needs to use a method in two classes A and B that are not in a subclass relationship. In a single inheritance language, the only way to do this (other than duplicating the code) is to promote the method to the common superclass of A and B. Thus, each instance of a method being implemented “too high” represents a method that *would have had to be duplicated* if it were implemented in the logically correct place. In fact, some of these methods would have to be duplicated several times. Thus, the “Total savings” row in Tables 6.1 and 6.2 is simply the sum of the number of methods that were found to be implemented too high, and the number that were duplicated.

Although the evils of duplication are well-known, the problem with implementing methods too high may not be so obvious. Implementing methods too high means that inherited behavior is inappropriate and must be canceled. For example, in the part of the collection hierarchy that we refactored, 15 messages are explicitly disabled in subclasses (typically by defining them as `self shouldNotImplement`). More problematic are the other 40 methods that are *implicitly* disabled because they directly or indirectly call explicitly disabled methods.

Implementing methods too high may be better than code duplication, but it nevertheless makes the whole hierarchy very much harder to understand. For example, the method `addAll:` in `Collection` sends `self add:` for each element of the argument. Consequently, it *appears* that every collection understands `addAll:`, although an

|                                              | Collection<br>Classes <i>without</i><br>String & Text | Collection<br>Classes <i>with</i><br>String & Text | Stream<br>Classes |
|----------------------------------------------|-------------------------------------------------------|----------------------------------------------------|-------------------|
| Number of concrete classes                   | 21                                                    | 23                                                 | 7                 |
| Number of methods in original version        | 635                                                   | 1 044                                              | 208               |
| Number of methods in trait version           | 567                                                   | 840                                                | 190               |
| Methods saved ( $m$ )                        | 68                                                    | 204                                                | 18                |
| Methods saved (ratio)                        | 10.7%                                                 | 19.5%                                              | 8.7%              |
| Source code saved (in bytes)                 | 9 527                                                 | 14 586                                             | 1 307             |
| Source code saved (ratio)                    | 11.9%                                                 | 10.4%                                              | 4.4%              |
| Methods “too high” in original ( $h$ )       | 55                                                    | 55                                                 | 76                |
| Methods “too high” as percentage of original | 8.7%                                                  | 5.3%                                               | 36.5%             |
| Methods “too high” not explicitly disabled   | 40                                                    | 40                                                 | 66                |
| Total methods saved ( $m + h$ )              | 123                                                   | 259                                                | 94                |
| Total methods saved (ratio)                  | 19.4%                                                 | 24.8%                                              | 45.2%             |

Table 6.1: Per hierarchy summary of the refactoring

|                                              | Totals                          |                              |
|----------------------------------------------|---------------------------------|------------------------------|
|                                              | <i>without</i><br>String & Text | <i>with</i><br>String & Text |
| Number of concrete classes                   | 28                              | 30                           |
| Number of methods in original version        | 843                             | 1 252                        |
| Number of methods in trait version           | 757                             | 1 030                        |
| Methods saved ( $m$ )                        | 86                              | 222                          |
| Methods saved (ratio)                        | 10.2%                           | 17.7%                        |
| Source code saved (in bytes)                 | 10 834                          | 15 893                       |
| Source code saved (ratio)                    | 9.9%                            | 9.4%                         |
| Methods “too high” in original ( $h$ )       | 131                             | 131                          |
| Methods “too high” as percentage of original | 15.5%                           | 10.5%                        |
| Methods “too high” not explicitly disabled   | 106                             | 106                          |
| Total methods saved ( $m + h$ )              | 217                             | 353                          |
| Total methods saved (ratio)                  | 25.7%                           | 28.2%                        |

Table 6.2: Summary of the refactoring (totals)

attempt to use this method on, say, an `Array`, will always cause a runtime error. In the trait implementation, there is no need to resort to this tactic: each method is present in exactly the classes that need it, and in no others. This makes the classes much easier for a programmer to understand: browsing the protocol of a class tells one exactly which methods can be used.

The second column of the table includes the refactored `String` and `Text` classes. The comparison is between the refactored classes and a version of the collection hierarchy containing an augmented `Text` class that defines all of the methods found in `String` but missing from the standard `Text` class. Because regularity of the interfaces of objects is so important to object-oriented programming, we argue that this augmented `Text` class is the one that really ought to be provided by standard Squeak — if it were feasible to do so with existing technology. The refactored version uses traits to provide `Text` with all the methods of `String`, and thus achieves the same interfaces with far fewer methods.

The third column shows the situation with the stream classes. Note that in this hierarchy there were fewer duplicated methods, and so the reduction in the number of methods achieved with traits is slightly lower. However, the reason that there was less duplication is that an enormous number of methods were implemented “too high”, precisely so that they could be shared. Moreover, many of these methods were not explicitly disabled. This makes the `Stream` classes hard to understand, because it appears that these methods ought to work, but in fact they will break when they call another method that is explicitly disabled.

The final two columns present the totals for `Collections` and `Streams`, excluding and including the `String` and `Text` classes.

### 6.3.4 Assessment of the Refactored Classes

Besides the quantitative improvements in the refactored part of the collection classes noted above, the trait implementation has other advantages that will have impact both inside and outside the collection hierarchy.

We undertook the refactoring in two phases. In the first phase we refactored 13 concrete collection classes, and none of the `Stream` classes. This phase set up the basic structure of the functional traits and the implementation traits; we developed 46 traits in all. We did not need to use exclusion or aliasing, because we were free to split any trait that was too large for our needs. As we continued our refactoring of the collection hierarchy, we found an *increasing* percentage reduction in code size as we applied the same reusable traits to remove code from more and more classes.

Returning to the refactoring three months later, we found that we were able to reuse many of our traits when we extended our refactoring to other parts of the hierarchy. Some of the traits were split into smaller pieces, but since these could be recombined without any change in semantics, the classes that had already been refactored were not affected. In some places we used exclusion to avoid disturbing an existing trait.

The traits that we have written will also allow us to construct new kinds of collection in the future, simply by composing the necessary traits and implementing a few glue methods. For example, we can build a class `PluggableBag` by using the trait `TPluggableAdaptor` in a subclass of `Bag`, and we can create immutable variants of collections by omitting the mutable interface traits. In addition, the availability of these traits frees the implementors of the collections framework from the need to ship pre-built collection classes for rarely used special cases. Instead, the main responsibilities of the implementor become the provision of (1) a basic class hierarchy that contains the more common collection classes and (2) a set of well-designed set of traits that can be composed with these classes. Using this basis, another programmer can then easily recompose the traits in order to build special-purpose collections.

Another advantage of the new framework is that some of the traits can be used *outside* of the collection classes. As an example, the trait `TEmptiness`, which requires `size` and provides `isEmpty`, `notEmpty`, `ifEmpty:`, `isEmptyOrNil ifEmpty:` and `ifNotEmpty:`, can be used in *any* class that defines `size`. Similarly, the trait `TEnumerationUI` can be used to provide a class with 24 methods from the enumeration protocol, provided that it implements `do:`, `emptyCopyOfSameSize`, and `errorNotFound`.

Why is this important? We believe that much of the power of the object-oriented paradigm comes from having many *different* objects understand the *same* protocol in corresponding ways. For example, it can be

quite frustrating to find that a `SoundBuffer`, although it understands `size` and `isEmpty`, does *not* understand `ifEmpty`. The availability of fine-grained traits at last makes it possible to make protocols more uniform across all of the classes in the system, with no cost in code size or maintainability, and with a *reduction* in the effort required to find one's way around the system.

### 6.3.5 Design Decisions

The availability of both trait composition and single inheritance gave us a lot of freedom in designing the new collection classes. Why did we choose the particular combination of trait reuse and inheritance described above? An alternative approach would have been to use trait composition exclusively and to minimize — or eliminate — the use of inheritance. If we had done this, all the concrete collection classes would have been built using trait composition alone, and every collection class would be a direct subclass of `Object` (or of an empty common superclass `Collection`).

We decided against this approach primarily for reasons of familiarity. Using both inheritance and trait composition in the new hierarchy makes it easier for programmers who are familiar with single inheritance code, and especially for programmers who know the old collection classes, to understand and extend the new ones. Indeed, a flattened view of the new collection classes exhibits a structure quite similar to the old ones, although the abstract superclasses do not correspond one-to-one.

The combination of single-inheritance and trait composition also turns out to be well-suited for explicitly representing a functional property layer with abstract classes and an implementation layer with concrete classes. This is particularly true because the separation between functional methods and implementation methods is not always very clear. For example, it is sometimes the case that a particular implementation trait defines a optimized variant of a method that is generically defined in a functional trait. Because the concrete classes are *composed* from the implementation traits but *inherit* from superclasses built from the functional traits, we can be sure that in these situations the implementation methods override the functional methods.

## 6.4 Discussion

In this section we first discuss some of the things we learned about traits during our refactoring. Then we examine some places where the theoretical benefits of traits were of practical importance, and argue that a similarly fine-grained decomposition would be much harder to accomplish with mixins or multiple inheritance.

### 6.4.1 Lessons Learned

During this refactoring we learned a number of things about traits and our programming tools, and also some more general things about refactoring. All of these findings confirmed our expectations about the importance of tool support and corresponding programming methodology described in Chapter 5.

**Traits simplify refactoring.** Using traits, refactoring a major hierarchy such as the Smalltalk collections is not as hard a task as one might think. We are not wizards; when we started refactoring we did not have a very clear plan of where we would end up. We just started pair programming, doing the simplest thing that could possibly work, until we found that it didn't work — at which point we did something just slightly more sophisticated.

When we started dragging methods out of existing classes and dropping them into traits, it was quite easy to identify the necessary traits. We had a superficial familiarity with the Smalltalk collection classes, and had re-read Cook's 1992 study [COOK 92]. So we expected to find traits related to the five different implementations and the major categories of functionality described in Section 6.1.1. When we found a method that did not seem to fit into one of the traits that we had already defined, we simply created a new trait. Often, the hardest

part was finding appropriate names for the traits. Naming is important and difficult; the naming scheme used in this chapter can surely be improved upon, even though it represents our third or fourth attempt.

**Tools are important** During the refactoring project, both the standard Smalltalk programming tools (which allow one to look at not just classes but also all the implementors of and senders of a particular message) and the trait specific tools described in Section 5.2 turned out to be an enormous help. It was particularly useful to know that a layer of subtraits could be introduced or eliminated without changing the semantics of any of the methods.

Thus, we could consider our refactoring task as simply grouping the existing collection behavior into coherent traits. For each of the newly constructed traits, the *requires* category in the browser always showed us which methods were missing in order to make the trait complete. Naturally, some of these missing methods belonged most logically in other traits; we simply continued adding methods to the trait until *all* of the unsatisfied requirements belonged in other traits.

**Use fine-grained components.** As our refactoring progressed, we realized that the methods in the collection hierarchy could be grouped into traits at a much finer granularity than we had initially thought. Given good tools, traits do not impose any cost for the finer-grained structure: we didn't have to make the trade-off between the elegance of the implementation and the understandability and usability of the functional interface that characterizes both mixins and multiple inheritance.

**Defer the design of the class hierarchy.** Getting a class hierarchy “right” is known to be hard. The problem is that one is usually forced to make decisions too early, before enough is known about the implementation. Our response was to put off making decisions for as long as possible, which turned out to be almost to the end of the first phase of the refactoring.

The theoretical properties of traits made us confident that things would turn out well in the end, provided that we collected behavior into logically coherent traits. Whether these traits would eventually be combined into complete classes or be used to build a deep hierarchy of abstract and concrete classes did not matter, because we knew that trait composition and inheritance could be freely combined. Once we had built the first few implementation and interface traits, it became obvious how to combine them. The more we combined traits, the more important the flattening property became.

## 6.4.2 Comparison to Other Approaches

In arguing that traits are a valuable contribution to the language designer's arsenal, we must address the question of whether we could not have obtained equally impressive results using mixins or multiple inheritance. We are convinced that the answer is “no”, and in this section will attempt to explain why.

In the previous chapters of this thesis, we have presented theoretical arguments for the superiority of traits over mixins and multiple inheritance. Here we will focus on experience rather than theory.

### Mixins

It is clear that also mixins could be used to tackle the identified problems in the Smalltalk collection classes. However, we found that because of the conceptual problems associated with mixins, achieving an equally fine-grained refactoring would be significantly more problematic with mixins than it is with traits.

Each of our refactored collection classes directly uses up to 22 traits. Counting also the traits inherited from the superclasses, our collection classes are built from a total of up to 35 traits (*e.g.*, `OrderedCollection` and `Text`). This is feasible because the sum operation lets us build a subclass from a group of traits in parallel, and the flattening property allows us to flatten the nested trait structure at any level. In particular, the programmer can work with each class as if it were built without any traits at all.

In contrast, mixins must be applied one at a time, using the ordinary single inheritance operator. This would result in huge and hard to understand inheritance chains with up to 35 levels. This is especially problematic because mixin composition does not adhere to the flattening property: whereas a class composed of a total of 35 traits can be consistently viewed and edited in a flat way (*i.e.*, as if only single inheritance were used), this is not possible with mixins, because mixins are composed using the inheritance operation in the first place, and because the semantics of super-calls depends on the exact occurrence of the call in the inheritance chain.

In our refactored hierarchy, there are also situations where several components are composed, but there is no total order that leads to the appropriate behavior. This is because sometimes there are *multiple* conflicts that need to be resolved by combining the conflicting methods or by excluding the methods that are not relevant.

With traits, this does not pose a problem because trait sum is commutative, and the composite entity can for each conflict *independently* decide how it should be resolved by using exclusions and aliasing. Mixins, in contrast, need to be totally ordered and do not support exclusions and aliasing. Thus, the only way to resolve such conflicts is to introduce additional glue mixins, which make such compositions more complex and harder to understand (see Section 2.2.1).

One example of this is the trait `TSortedImpl` where we had two conflicts: `at:ifAbsent:` and `collect:`, but no single subtrait that takes precedence for both of them. This is exactly the situation exclusion is designed for, and we obtained the desired behavior by excluding `at:ifAbsent:` from the subtrait `TExtensibleSequencedImpl` and `collect:` from `TOrderedSortedCommonImpl`.

Using mixins, the solution would be to either modify `TExtensibleSequencedImpl`, or to introduce a new intermediate mixin corresponding to `TExtensibleSequencedImpl - {at:ifAbsent:}`. Neither choice is desirable. Modifying the components is bad because it may break other places where these components are used. Introducing intermediate “glue mixins” makes the inheritance chains even longer and harder to understand (cf. Sections 2.1.2 and 2.2.1).

While composite mixins [BRAC 92] allow the programmer to improve the comprehension of such complex hierarchies by structuring the involved mixins, they do not solve the conceptual problems related to mixins (cf. Section 2.1.2). This is because composite mixins are based on the same linear form of composition as mixins, and they therefore suffer from the same problems such as fragility with respect to changes and dispersal of glue code. In practice, this means that composite mixins would allow us to structure the 21 mixins used to build `SequencedImmutable` into a handful of composite mixins. However, it also means that the identified problems (*e.g.*, fragility with respect to changes) can then occur both *within* and *between* these composite mixins.

In the process of our refactoring work, we also encountered many situations where adding a new method to a component caused a conflict with another component in distant code. Thanks to the commutativity of trait sum and the requirement of explicit conflict resolution, all of these places were immediately detected, and we were able to re-establish the correct semantics by making an appropriate adjustment to the relevant trait composition clause. It was never necessary to modify other components, so we never found ourselves in a situation where resolving one conflict created two more.

With mixins, this would not have been the case. First, we would not have detected conflicting methods so easily because the order of the mixins *implicitly* “resolves” each conflict, although not necessarily in the way that the programmer intends! Second, even if we had noticed that a conflict had been resolved in an incorrect way, it would have been much harder to actually re-establish the correct behavior.

A comparison of our refactored collection classes to the mixin-based collection framework of Strongtalk [BRAC 93] provides more data on the effectiveness of mixins and traits. Both frameworks are based on Smalltalk-80 and are therefore quite comparable. Strongtalk has more collection classes, but uses only 10 different mixins, compared to 67 traits in our hierarchy. In particular, Strongtalk does not factor out characteristics such as extensible, implicitly sequenced, and explicitly sequenced; neither does it make aspects like enumeration reusable outside of the collection framework.

Of course, the fact that the designers of Strongtalk decided not to pursue a fine-grained decomposition into mixins does not mean that doing so would be impossible. However, it is an indication that the Strongtalk designers decided that the disadvantages of a finer structure outweighed the advantages. In contrast, we have found that with traits the fine-grained decomposition has only advantages.

### Multiple Inheritance

Multiple inheritance would also have solved many of the problems that we identified in the single inheritance version of the Smalltalk collection classes. However, like mixins, multiple inheritance alone would not be expressive enough to achieve a fine-grained refactoring of the collection hierarchy.

For example, our refactored hierarchy uses several adaptor traits such as `TIdentityAdaptor`, which is used to turn collection classes like `Set`, `WeakSet`, and `Dictionary` into new classes that compare elements based on their identity rather than equality. But as described in Sections 2.1.1 and 2.2.2, multiple inheritance alone cannot express such adaptors without code duplication.

## 6.5 Conclusions

We undertook this refactoring primarily to obtain some practical experience with the use of traits. We believed that the theoretical properties that we had given to traits — especially flattening, but also the retention of explicit conflicts in the sum operation — were the right ones. But programming languages are tools, and theoretical elegance is no substitute for usability. Only extensive use on a realistic codebase could validate these beliefs.

It did. Although we had designed them, we were surprised how well the trait mechanism and the corresponding tools worked in practice. The theoretical characteristics do really seem to give the tools desirable practical properties.

However wonderful a language technology may be to those who use it, new language features can be a real obstacle to those who have not previously met them. One of the pleasant properties of traits is that they do not change the method-level syntax of Smalltalk at all. Thus, an ordinary Smalltalk programmer can open an ordinary Smalltalk system browser on our new hierarchy and understand everything that she sees. All of the concrete classes will be there, with all of their methods. Trait methods will appear to be defined directly in a subclass or inherited from a superclass exactly as in ordinary Smalltalk, and the semantics will be exactly the same. If the programmer modifies a method in a conventional class view, and the method is actually defined in a shared trait, then the effect will be to define a customized (unshared) version of the method local to the class. Again, this is exactly the semantics of ordinary Smalltalk.

This property is critically important, because we believe that one of the reasons that previous technologies such as mixins and multiple inheritance have not become popular is because of the complexity that they force on every programmer. For example, the rules for linearizing multiple inheritance chains must be understood by every programmer who looks at or modifies a multiple inheritance hierarchy.

This refactoring also confirmed that working with traits is compatible with an agile style of development (cf. Section 5.3): it does not require one to do all of the design “up front”, when nothing is known about the system, but lets one start by identifying related methods and putting them into traits. The shape of the inheritance hierarchy can emerge later.

Good tool support proved to be critical: the availability of the trait browser (cf. Section 5.2) had a tremendous impact on the efficiency of the refactoring task. It is hard to imagine undertaking this refactoring with an ordinary Smalltalk system browser that does not show virtual categories and does not support the abstract instance variable refactoring. Performing the same task with a file-based tool such as `emacs` is inconceivable to us. The incremental nature of the Smalltalk environment played an important role, because the current state of the composition was instantly visible at all times.

To summarize: we have successfully refactored a significant subset of the Smalltalk collections classes. In the process we:

- removed code duplication;
- increased uniformity;
- improved understandability;

- provided reusable traits that make it easier to write new collection classes; and
- made it possible to reuse collection code *outside* of the collection hierarchy.

The third claim, improved understandability, is necessarily subjective. However, we argue that three *objective* features of the refactored hierarchy support it. First, there is no discrepancy between the apparent and actual interfaces of a class. In other words, we never needed to resort to implementing a method “too high” in the hierarchy just to enable reuse. As a consequence, when browsing the hierarchy, “what you see is what you get”: all of the public methods in a class are actually available. Second, the structured view (with fine grained traits) provides a lot of insight about the functional properties of the methods: which mutate the object, which require sequenceability, which do enumeration, and so on. Since the structured view containing this extra information is optional, there is no tradeoff to be made in supplying it: programmers who do not find it useful can simply not use it. Third, we have reduced the code size of the hierarchy, which also facilitates understandability.

## Chapter 7

# Case Study: Trait-based Composition of Class Properties

As we discussed in Chapter 4, Smalltalk classes are objects as well: they are the singleton instances of classes called metaclasses. However, the Smalltalk metaclass architecture is not expressive enough to share class properties such as *singleton* and *final* between metaclasses [BOUR 98]. In the case study presented in this chapter, we investigate whether we can overcome this deficiency by applying the trait mechanism to metaclasses [DUCA 05].

We will start with an overview of the purpose and problems of metaclasses, which directly leads us to the motivation for this case study (Section 7.1). Then we show how class properties are represented as traits and how they are composed and applied to metaclasses (Section 7.2). We also develop a small framework of composable class properties and show how it is applied in realistic situations (Section 7.3). We end this chapter with an evaluation that compares our approach to other forms of class property composition (Section 7.4).

### 7.1 Motivation and Overview

In class-based object-oriented programming, classes are used as instance generators and to implement the behavior of objects. In object-oriented languages such as CLOS, Smalltalk and Ruby, classes themselves are first-class objects, being instances of so-called *metaclasses* [INGA 76, COIN 87, KICZ 91, DANF 94, FORM 99]. In the same way that classes (like `String`) define the properties of their instances (like `'Hello'`), metaclasses (like `String` class) define the properties of their instances (`String`). Examples of class properties are *singleton*, *final*, *abstract*, etc. [LEDO 96].

Treating classes as first-class objects (and thus treating metaclasses as first-class classes) is important for two primary reasons.

- *Uniformity and Control.* Metaclasses are the natural place to *specify* and *control* class behavior such as object creation and subclassing.
- *Reuse of Class Behavior.* If a metaclass is like any other class, class behavior can be reused by applying conventional reuse and decomposition techniques to the metaclasses [LEDO 96]. Hence the same techniques that are available for base classes (inheritance and overriding of methods, for example) are applicable at the meta level.

When a language has metaclasses, those metaclasses can be *implicit* or *explicit*. The difference is that explicit metaclasses give the programmer full control over which metaclass should be used for a certain class [INGA 76, COIN 87, KICZ 91, DANF 94], whereas implicit metaclasses are, as indicated by their name, automatically and implicitly defined for each class [GOLD 89]. While the programmer can modify, add, and remove

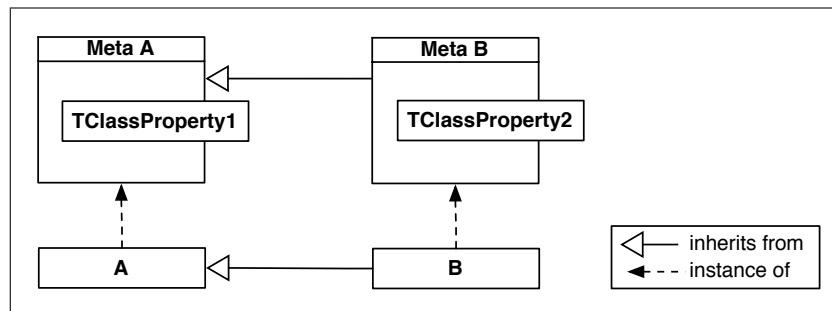


Figure 7.1: Application of class properties represented as traits

methods of such implicit metaclasses, it is not possible to change their superclasses nor is it possible to assign them to other classes.

As a consequence, implicit metaclasses successfully address the goal of “uniformity and control”, but they usually fall short of achieving “reuse of class behavior” because the programmer cannot write reusable metaclasses (such as *Abstract* and *Singleton*) that can be shared between more than one class. Instead, every time one needs an abstract or singleton class, the same code needs to be implemented in the class’ implicit metaclass.

Explicit metaclasses avoid this limitation because the programmer can explicitly define the metaclass of each class. As an example, the singleton behavior can be factored out to a *Singleton* metaclass, which can then be used for all singleton classes.

However languages with explicit metaclasses suffer from the fact that composition can be *unsafe* [COIN 87, GRAU 89] and are often based on *non-uniform* mechanisms; *i.e.* the meta-programmer cannot apply to the meta level the same reuse mechanisms that are available at the base level. This non-uniformity does not only require the meta-programmer to learn a set of specialized composition mechanisms, but it also contradicts the motivation for introducing metaclasses in the first place: the desire for uniformity.

In the case study described in this chapter, we addressed these problems by modeling class properties as traits and using trait composition to safely combine and reuse these properties in metaclasses. This allows us to achieve the benefits of explicit metaclasses without sacrificing the safety and simplicity associated with implicit metaclasses. By using a single, uniform mechanism for both the base and the meta level, this form of class composition enjoys all the conceptual benefits of the trait composition model. In particular, composition conflicts that occur when composing two properties that do not quite fit together are detected automatically, and the conflict resolution is *explicit* and under the control of the composing entity.

## 7.2 Representing Class Properties As Traits

In Squeak (as in Smalltalk), metaclasses are *implicit* and created *automatically* whenever a class is created [GOLD 89]. The new class is then the sole instance of this implicitly created metaclass. The process of creating the classes and metaclasses is designed so that the class and metaclass hierarchies are always parallel: if a class *B* is defined as a new subclass of *A*, then *B*’s metaclass is implicitly defined to be a subclass of *A*’s metaclass. This is illustrated in Figure 4.1, which shows the class hierarchy consisting of *Object*, *Point*, and *Vertex* as well as the parallel metaclass hierarchy.

As we will discuss in detail in Section 7.4.1, this guarantees that the architecture is safe in the sense that compatibility problems between the base and the meta level can never occur. However, this design is so restrictive that it completely prevents reuse of class properties between classes that are not directly related by inheritance. As an illustration, consider the Smalltalk classes *True* and *UndefinedObject*, both of which have only a single instance (*true* and *nil*). Although these two classes are both singleton, it is not possible to share

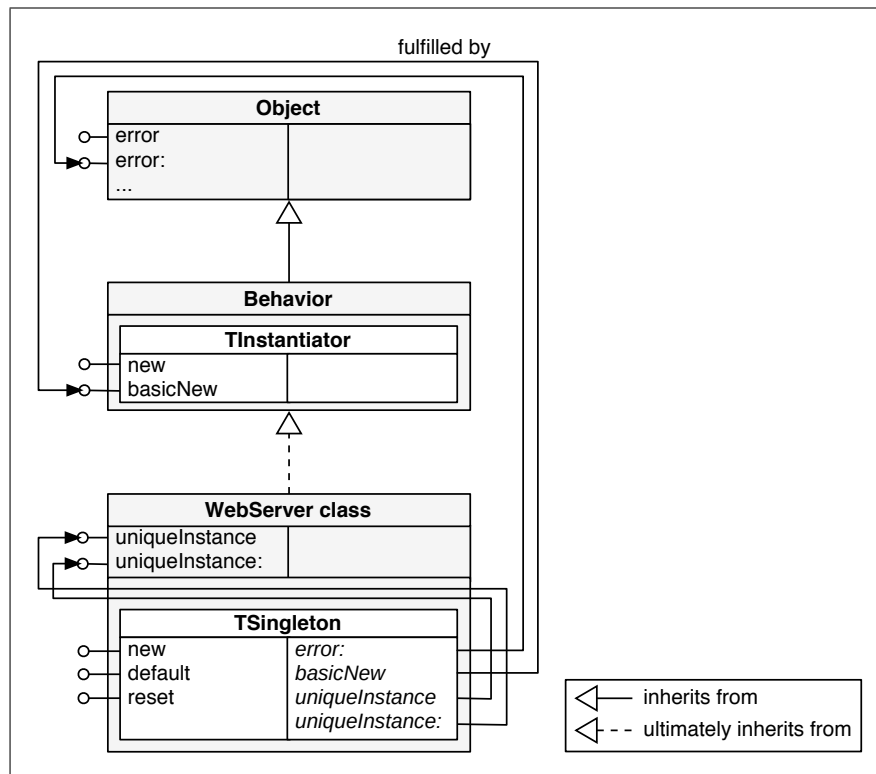


Figure 7.2: Behavior and WebServer class built from the traits TInstantiator and TSingleton

the corresponding implementation between their metaclasses True class and UndefinedObject class. This is because the only common superclass of these two metaclasses is Object class, which is not an appropriate place to implement the singleton behavior: Object is not singleton and has many subclasses that are not singleton.

Figure 7.1 illustrates how we avoid this limitation by representing class properties as traits that can then be composed and applied to arbitrary metaclasses. Because this does not affect the traditional parallel inheritance scheme of Smalltalk, our approach is still safe (cf. Section 7.4.1), but it allows composition and reuse of class properties using trait composition.

### 7.2.1 Singleton

As a simple example, we show how our approach allows us to define the singleton property and apply it to the metaclass WebServer class to declare the class WebServer as singleton. This is illustrated in Figure 7.2, with the corresponding Smalltalk code shown in Figure 7.3.

We first create a trait TSingleton, which represents the singleton property. This trait provides the methods default, which returns the singleton instance (after creating it if necessary), new, which raises an error, and reset, which resets the class by clearing the variable holding the current singleton instance. It requires the methods basicNew, which returns a newly created instance, as well as the methods uniqueInstance and uniqueInstance:, the accessors for the unique instance.

To declare the class WebServer as singleton, we now add the trait TSingleton to the metaclass WebServer class. We satisfy the open requirements by defining the state needed to maintain the singleton instance and generating the corresponding accessor methods uniqueInstance and uniqueInstance:. Note that the required

```

Trait named: #TSingleton uses: {}

default
  self uniqueInstance isNil
    ifTrue: [self uniqueInstance: self basicNew].
  ↑ self uniqueInstance.

new
  self error: 'You should use default'.

reset
  self uniqueInstance: nil.

error:
  self requirement.

basicNew
  self requirement.

uniqueInstance
  self requirement.

uniqueInstance
  self requirement.

WebServer class
  uses: TSingleton
  instanceVariableNames: 'uniqueInstance'

  uniqueInstance
    ↑ uniqueInstance.

  uniqueInstance: aWebServer
    uniqueInstance := aWebServer.

```

Figure 7.3: The metaclass WebServer class implemented using the property trait TSingleton

method `basicNew`<sup>1</sup> is provided by the class `Behavior`, which is an indirect superclass of `WebServer` class (and any other metaclass). Figure 7.2 also shows that `Behavior` is built from traits. In particular, the methods `new` and `basicNew` are provided by the trait `TInstantiator`.

### 7.2.2 Modeling the Boolean Hierarchy

As a more realistic example, we will now show how we used traits to refactor the Smalltalk Boolean hierarchy, which consists of the abstract class `Boolean` and its two singleton subclasses `True` and `False`. In the traditional hierarchy, the necessary behavior for making `Boolean` abstract and its subclasses singleton is implemented in an *ad hoc* manner, *i.e.*, by overriding the methods of the instance creation protocol in each of these classes. Besides the fact that this leads to code duplication, it has the disadvantage that a programmer cannot immediately see that these classes use the abstract and singleton patterns.

Figure 7.4 shows how we avoided these problems by applying the new trait `TAbstract` to the class `Boolean` and applying the trait `TSingleton` introduced above to the subclasses `True` and `False`.

<sup>1</sup>`basicNew` is the name of the instance allocation primitive in Smalltalk (see Section 7.3).

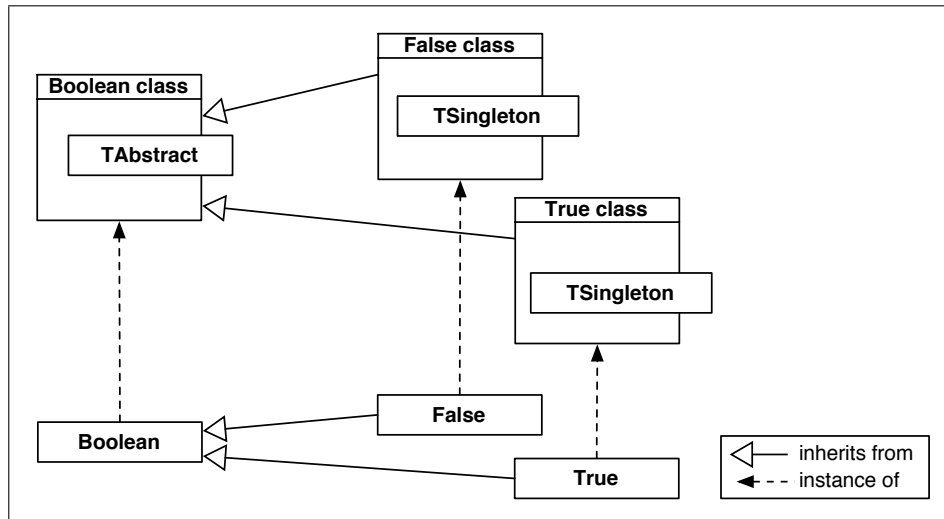


Figure 7.4: The refactored Boolean hierarchy

```

Trait named: #TAbstract uses: {}

new
  self error: 'Abstract class cannot have instances.'

new: size
  self error: 'Abstract class cannot have instances.'

error:
  self requirement.
  
```

Figure 7.5: Implementation of the property TAbstract

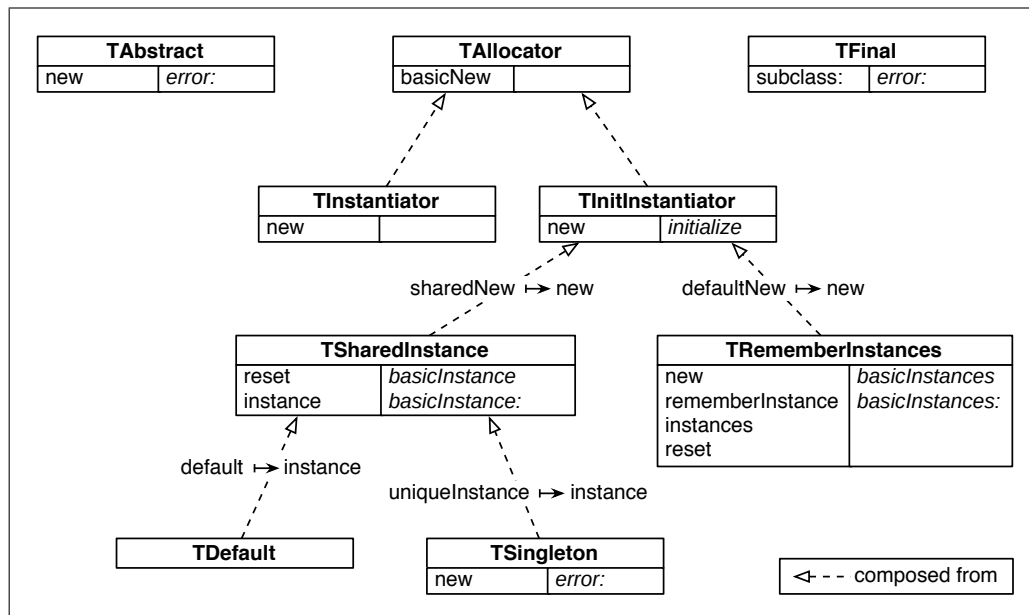


Figure 7.6: A fine-grained architecture of class properties based on traits

**TAbstract.** The implementation of the trait TAbstract is shown in Figure 7.5. It provides the two methods `new` and `new:`, which yield an error whenever they are called to create a new instance. TAbstract requires only the method `error:`.

**Boolean class.** The metaclass Boolean class is built using the trait TAbstract. Note that we did not have to implement any glue methods because the required method `error:` is inherited from the class Object.

**False class and True class.** The metaclasses False class and True class are composed from the trait TSingleton (Figure 7.3), which requires the methods `basicNew`, `uniqueInstance`, and `uniqueInstance:`. Thus, the metaclasses False class and True class declare an instance variable `uniqueInstance` as well as the corresponding accessor methods `uniqueInstance` and `uniqueInstance:`. The required method `basicNew` does not have to be implemented in False class and True class as it is inherited from the class Behavior.

## 7.3 A Framework of Class Properties

We have so far presented some simple examples that show how class properties can be represented as traits that can then be combined or applied to arbitrary classes. In this section, we present a framework of the most common Smalltalk class properties, which we identified by collecting the different properties that are implicitly coded into the Squeak metaclasses.

Since many of these properties are related to instance creation, we first clarify the basic instance creation idiom of Smalltalk. Creation of a new instance involves two different methods, namely `basicNew` and `new`<sup>2</sup>. The method `basicNew` is a low-level primitive that allocates a new instance of the receiver class and is by convention neither called from the outside of a class nor overridden in a subclass [ALPE 98]. The method `new` stands at a conceptually higher level; its purpose is to return a properly initialized instance of the receiver class.

<sup>2</sup>Note that there are also the methods `basicNew:` and `new:`, which are used to create variable sized objects with indexed fields (*i.e.*, arrays). For the sake of simplicity and conciseness, we do not take these methods into account here.

```

Trait named: #TAllocator uses: {}

    basicNew
      <primitive>

Trait named: #TInstantiator uses: TAllocator

    new
      ↑ self basicNew.

Trait named: #TInitInstantiator uses: TAllocator

    new
      ↑ self basicNew initialize.

Trait named: #TFinal uses: {}

    subclass
      self error: 'Cannot create subclasses of a final class.'.

```

Figure 7.7: Implementation of the allocation, instantiation, and final class properties

For most classes, `new` therefore calls `basicNew` to obtain a new instance and then initializes it with reasonable default values.

### 7.3.1 Class Properties

Figure 7.6 gives an overview of the class properties we identified, and how they are arranged in a trait hierarchy. The corresponding Smalltalk code is shown in Figures 7.7 and 7.8 with the exception of the trait `TAbstract`, which was already shown in Figure 7.5. Glue methods are emboldened as usual. However, for conciseness we show the required methods only in the conceptual diagram (Figure 7.6) but not in the corresponding code (Figures 7.7 and 7.8).

**Allocation.** As indicated by its name, the trait `TAllocator` provides the behavior to allocate new instances. It provides a single method, which is the standard `basicNew` primitive.

**Instantiation.** The traits `TInstantiator` and `TInitInstantiator` are two other class properties for instance creation. The trait `TInstantiator` uses the trait `TAllocator` and implements the method `new` in the traditional Smalltalk manner, which means that it does not initialize the newly created instance. The trait `TInitInstantiator` is based on `TAllocator`, but it also initializes the newly created instance by calling the method `initialize` before the instance is returned.

Note that the method `initialize` is called on the new instance, which means that the requirement for `initialize` in the trait `TInitInstantiator` is actually a requirement for the instance side.

**Remembering Instances.** The trait `TRememberInstances` represents an instance creation property that remembers all the instances created by a class. It uses the trait `TInitInstantiator` and aliases the method `new` of the trait `TInitInstantiator` to make it available under the new name `defaultNew`. This allows one to access the original `new` method of the trait `TInitInstantiator` even though it is overridden in `TRememberInstances`.

```

Trait named: #TRememberInstances uses: TInitInstantiator @ {#defaultNew -> #new}

  new
    ↑ self rememberInstance: self defaultNew.

  rememberInstance: anObject
    ↑ self instances add: anObject.

  instances
    self basicInstances isNil
      ifTrue: [self reset].
    ↑ self basicInstances.

  reset
    self basicInstances: IdentitySet new.

Trait named: #TSharedInstance uses: TInitInstantiator @ {#sharedNew -> #new}

  reset
    self basicInstance: self sharedNew.

  instance
    self basicInstance isNil
      ifTrue: [self reset].
    ↑ self basicInstance.

Trait named: #TSingleton uses: TSharedInstance @ {#uniqueInstance -> #instance}

  new
    self error: 'Cannot create new instances of a singleton class.
      Use uniqueInstance instead.'.

Trait named: #TDefault uses: TSharedInstance @ {#default -> #instance}

  "no methods"

```

Figure 7.8: Implementation of different instance creation class properties

The trait `TRememberInstances` provides the methods `new`, `rememberInstance:`, `instances`, and `reset`. The method `reset` initializes and resets the collection containing the remembered instances. The method `instances` serves as accessor of the collection containing the instances: it resets the collection if it is not yet initialized and then returns it. The method `new` overrides the version obtained from the trait `TInitInstantiator`. It first uses the alias `defaultNew` to create a new instance and then calls the method `rememberInstance`, which adds the new instance to the collection of remembered instances.

The trait `TRememberInstances` requires only the methods `rememberedInstances` and `rememberedInstances:`, which access a collection containing the created instances. Alternatively, one could define the methods `reset` and `rememberedInstances:` as requirements. This would allow the metaclasses that use the trait to specify other implementations for keeping track of the created instances.

**Default and Singleton.** The traits `TDefault` and `TSingleton` implement the class properties corresponding to the *default instance* and *singleton* design patterns [ALPE 98]. Whereas a singleton can have only a single instance, a class adhering to the default instance pattern has one default instance but can also have an arbitrary number of other instances.

```

Trait named: #TDefaultAndRememberInstances
uses: TDefault @ {#defaultReset → #reset} +
      TRememberInstances @ {#storeNew → #new. #storeReset → #reset} - {#new}

sharedNew
  ↑ self storeNew.

reset
  self storeReset.
  self defaultReset.

```

Figure 7.9: Composing the conflicting properties TDefault and TRememberInstances

Since these two properties are similar, we factored out the common code into the trait TSharedInstance. To get the basic instantiation behavior, this trait uses the property TInitInstantiator and again applies an alias to ensure that the method new is available under the name sharedNew. It provides the methods reset and sharedInstance. While reset creates a new instance and stores it for later retrieval, the method sharedInstance is an accessor that creates a shared instance if necessary and then returns it.

The property TDefault is then defined as an extension of the trait TSharedInstance that introduces the alias default for the method sharedInstance. Similarly, the property TSingleton is defined by introducing the alias uniqueInstance for the sharedInstance method. Furthermore, TSingleton overrides the method new so that it cannot be used to create additional instances.

**Final.** Another class property, which has been popularized by Java, is the property TFinal which ensures that a class cannot have subclasses. In Smalltalk, this is achieved by overriding the message subclass:<sup>3</sup>. Note that unlike all the other properties presented in this section, TFinal is not concerned with instance creation, and it is therefore entirely independent of the other properties.

### 7.3.2 Advantages for the Programmer

Having a framework of class properties has many advantages for a programmer. Whenever a new class needs to be created, the programmer can choose how instance are created and whether or not the class should be final by composing the corresponding class properties. Besides the obvious advantage of avoiding code duplication, this also makes the design more explicit and hence improves understandability of the class.

In addition, factoring out the properties in such a fine-grained way gives the programmer a simple yet powerful way to control crucial aspects of the system. In the framework presented above, we have for example decided to use the trait TInitInstantiator as the basis for all the other instance creation properties. However, if we later decided to comply to the Smalltalk standard and create uninitialized instances by default, we could still make this change without modifying any of the methods. Instead, we would just need to modify the composition clause of the traits TRememberInstances and TSharedInstance so that they use the trait TInstantiator instead of TInitInstantiator.

Another advantage is that the programmer can use the provided framework as a basis to build more specialized class properties. As a concrete example, imagine that we want to combine the class properties TDefault and TRememberInstances to get a new property that allows both a default instance and also remembers all its instances. Using our trait-based framework, we do this by creating a new trait TDefaultAndRememberInstances which uses TRememberInstances and TDefault and is shown in Figure 7.9.

Note that both traits TRememberInstances and TDefault provide a method new, and that we avoid a conflict by excluding the version of new defined by TRememberInstances when it is composed. As a consequence

<sup>3</sup>In reality, the method to create a subclass takes more arguments.

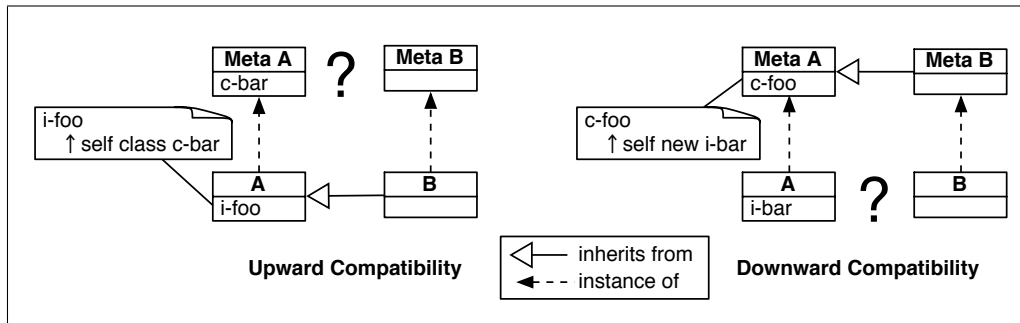


Figure 7.10: Upward and downward compatibility

the composite trait contains the new method provided by TDefault, which uses `sharedNew` to create a new instance. Since we want to make sure that each new instance is also stored, we override `sharedNew` so that it calls `storeNew`, which is an alias for the excluded new method of `TRememberInstances`.

The method `reset` is also provided by both traits, and we use aliasing to make sure that we can access both conflicting methods. We then resolve the conflict by overriding the method `reset` so that it first removes the stored instances (by calling `storeReset`) and then creates a new default instance (by calling `defaultReset`). Note that the newly created instance will be remembered as the default instance and will also be stored in the collection with all the instances of the class.

## 7.4 Evaluation

So far, we have shown how the trait mechanism can be used to represent, compose and apply class properties. We have also given realistic examples of how such class properties can be used in a Smalltalk system, and how this makes the design more explicit and reduces code duplication.

In this section, we further evaluate the proposed mechanism by comparing it to other approaches for class property composition. We start by presenting the set of criteria that serves as the basis for our comparison.

### 7.4.1 Criteria for Class Property Composition

Offering explicit control over metaclasses allows the programmer to reuse class properties but also opens the door for *metaclass compatibility problems* [GRAU 89]. This section defines criteria for characterizing and distinguishing different approaches for metaclass composition. The first three criteria were identified by Bouraqadi *et al.* [BOUR 98] (*upward compatibility*, *downward compatibility* and *non-propagating class properties*) and are related to compatibility between the base and the meta level as well as propagation of class properties in the inheritance hierarchy. The other criteria qualify the mechanisms that are used to compose class properties and apply them to the metaclasses.

**Upward compatibility.** The fact that classes are instances of other classes which define their behavior introduces hidden dependencies in the inheritance relationships between the classes and their metaclasses. Careless inheritance at one level (be it the base or meta level), can break inter-level communication. Bouraqadi *et al.* [BOUR 98] partitioned these metaclass compatibility problems into two cases, named *upward* and *downward* compatibility. The criteria for upward compatibility is defined as follows:

*Let B be a subclass of A, MetaB the metaclass of B, and MetaA the metaclass of A. Upward compatibility is ensured for MetaB and MetaA iff every possible message that does not lead to an error for any instance of A, will not lead to an error for any instance of B.*

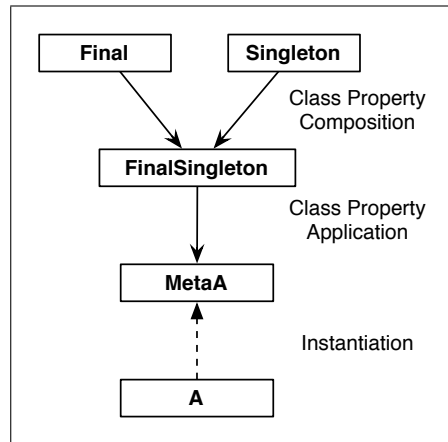


Figure 7.11: Property composition and property application

This is illustrated in the left part of Figure 7.10. When an instance of B receives the message `i-foo`, the message `c-bar` is sent to B. The composition of A and B is upward compatible, if B understands the message `c-bar`, *i.e.* MetaB should implement it or somehow inherit it from MetaA.

**Downward compatibility.** Downward compatibility is defined in an analogous way:

*Let MetaB be a subclass of the metaclass MetaA. Downward compatibility is ensured for two classes B, an instance of MetaB and A, an instance of MetaA iff every possible message that does not lead to an error for A, will not lead to an error for B.*

Downward compatibility is illustrated on the right of Figure 7.10. When B receives the message `c-foo`, the message `i-bar` is sent to a newly created instance of B. The composition of MetaA and MetaB is downward compatible, if that new instance of B understands the message `i-bar`, *i.e.* B should implement it or somehow inherit it from A.

Based on the definition of upward and downward compatibility, we can now define what it means for metaclass composition to be *safe*: *Metaclass composition is safe if it guarantees downward and upward compatibility.*

**Non-propagating class properties.** A class property assigned to a class is *non-propagating* iff it is not automatically propagated to its subclasses. With a mechanism that supports non-propagating class properties, it is for example possible to define a class as abstract while keeping its subclasses concrete without having to apply a class property at the subclasses level.

**Class property composition.** One of the motivations for having explicit class properties is to combine them. For example, one might want to compose the singleton property with the final property in order to get a new property that makes a class both singleton and final. Hence a mechanism is needed that supports such property composition. This can be a general-purpose language mechanism such as multiple inheritance [KICZ 91, DANF 94], mixin composition [BOUR 04], chain of responsibility [MULE 95], or an *ad hoc* mechanism such as generation of new classes and methods [BOUR 98].

**Class property application.** Property application is the mechanism by which simple or composed class properties are applied to metaclasses. Note that we make a conceptual distinction between the *composition* of class properties and the *application* of a class property to a specific class because some approaches employ different techniques for these two purposes. This distinction is illustrated in Figure 7.11, where two properties

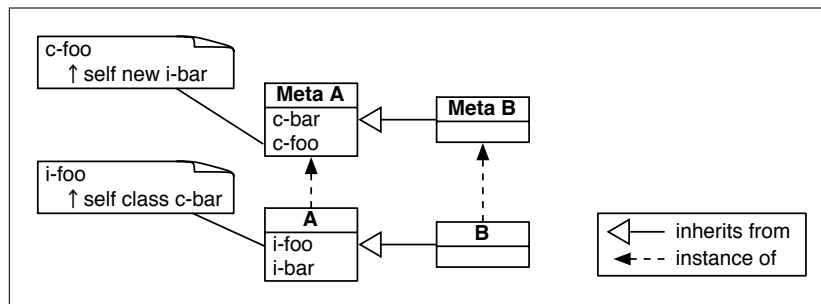


Figure 7.12: Smalltalk metaclasses organized in a parallel hierarchy

Final and Singleton are first composed to a new property FinalSingleton, which is in turn applied to the metaclass MetaA.

**Explicit or implicit control.** The mechanism used to apply and combine class properties can be *implicit* or *explicit*. We call these mechanisms *implicit* if the system automatically combines or applies class properties and resolves conflicts using an implicit strategy that may or may not lead to the desired result. We call such mechanisms *explicit* if the system gives the programmer explicit control over how class properties are combined and applied. In particular, the programmer must have *explicit control* over how conflicts are resolved.

## 7.4.2 Comparison to Other Approaches

In this section, we use the criteria presented above to compare trait-based class property composition to other systems and approaches that feature explicit metaclasses or class properties. We also discuss the solution offered by Smalltalk (although it has implicit metaclasses and does not support class property composition) since it forms the basis for NeoClasstalk and our own approach. Table 7.1 summarizes and compares these approaches.

### Smalltalk

In Smalltalk and more recently in Ruby, metaclasses are *implicit* and created *automatically* whenever a class is created [GOLD 89]. The new class is then the sole instance of this implicitly created metaclass. As explained in Section 7.2 and illustrated in Figure 7.12, this causes the class and the metaclass hierarchy to be parallel, which in turn guarantees upward and downward compatibility. However, this approach prevents the reuse of class properties between inheritance hierarchies.

### Trait-based Composition of Class Properties

Because our approach is based on the parallel inheritance hierarchies known from Smalltalk, it guarantees both upward and downward compatibility. Since class properties are represented as traits, trait composition is used for both property composition and application. This makes the model completely uniform: programming on the meta-level is based on the same language features as programming on the base level. The use of traits also ensures that composition and application of class properties is explicit and that the programmer has full control over how possible conflicts are resolved. The only limitation is that trait-based class properties are always propagated: if a trait is applied to a metaclass, it is automatically inherited by each of its subclasses.

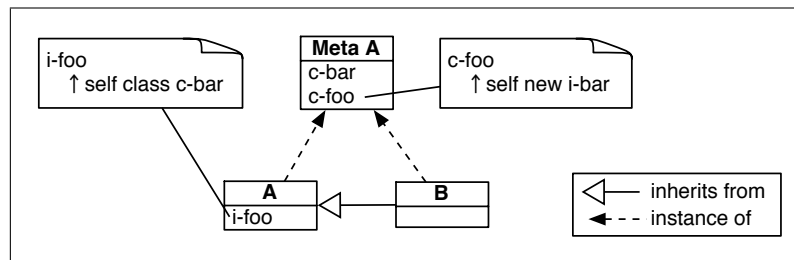


Figure 7.13: By default, a CLOS class and its subclasses are instances of the same metaclass

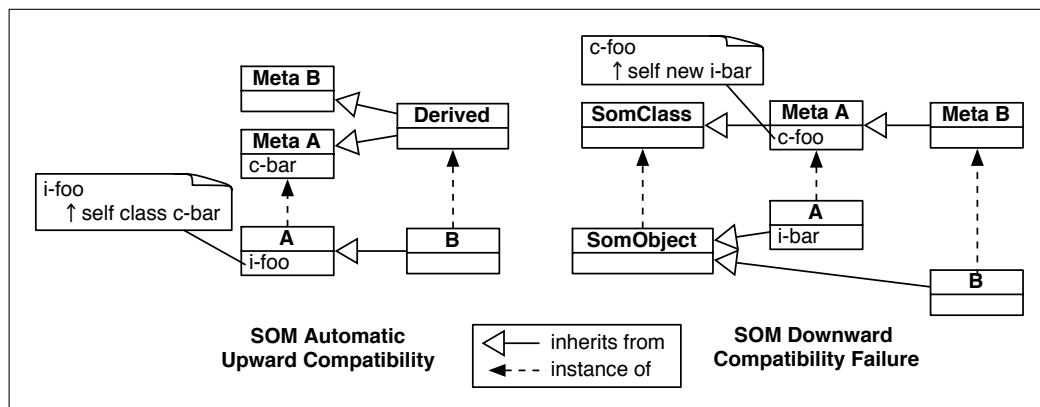


Figure 7.14: SOM guarantees upward but not downward compatibility.

## CLOS

CLOS's approach could be summarized as “do it yourself”. By default, a CLOS class and its subclasses must be instances of the same metaclass (cf. Figure 7.13), prohibiting classes in the same hierarchy from having different class properties. However, since CLOS has explicit metaclasses, multiple inheritance can be used to compose class properties. As illustrated in Figure 7.11, it is for example possible to use multiple inheritance to combine the two CLOS metaclasses representing the properties `Final` and `Singleton` into a new metaclass `FinalSingleton`. Note, however, that this form of composition using multiple inheritance suffers from the same linearization problems as the application of multiple inheritance at the base level (cf. Section 2.2.3).

The general CLOS rule that a class and its subclasses must be instances of the same metaclass can be circumvented using CLOS's metaobject protocol (MOP). Indeed, the generic function `validate-superclass` offers a meta-programmer the possibility to specify that a class and its subclasses can be instances of different classes [KICZ 91]. However, this comes at a very high price because the CLOS MOP does not provide predefined strategies for avoiding compatibility problems or for dealing with possible conflicts.

**Summary.** By default CLOS is upward and downward compatible but it prevents usage of different metaclasses within an inheritance hierarchy and reuse of class properties. Both the composition of class properties and the application of properties are based on multiple inheritance. The control of the composition is explicit, because the user has to use multiple inheritance to create a new metaclass. However, since multiple inheritance in CLOS uses implicit linearization, the well-known problems associated with this form of conflict resolution also apply to the meta level.

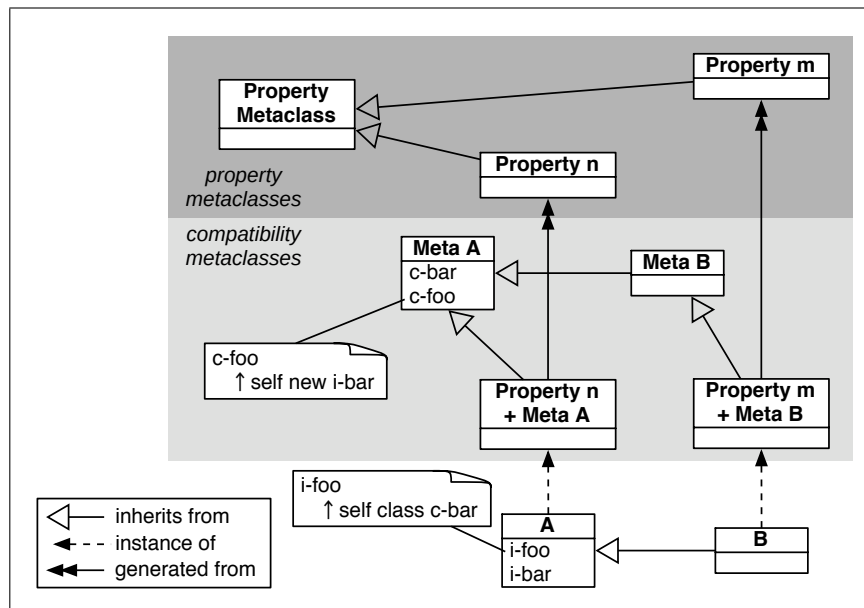


Figure 7.15: Class property composition in NeoClasstalk.

## SOM

The solution proposed by SOM (System Object Model) [FORM 99] is based on the automatic generation of *derived metaclasses*, which inherit multiply from the metaclasses to be composed. When a class is specified to be an instance of a certain metaclass, SOM automatically determines whether upward compatibility is ensured and if necessary creates a derived metaclass. This process is illustrated on the left side of Figure 7.14, where the class B (originally an instance of MetaB), inheriting from class A (instance of MetaA) finally becomes an instance of a derived metaclass inheriting from MetaA and MetaB. Note that SOM ensures that in case of multiple inheritance ambiguities, the existing metaclass MetaB takes precedence over MetaA (since B is a subclass of A).

While SOM guarantees upward compatibility, it does not guarantee downward compatibility [BOUR 98], which is illustrated on the right side of Figure 7.14: when the class B receives the c-foo message, a run-time error will occur because its instances do not understand the i-bar message.

**Summary.** In SOM, the composition of class properties is based on multiple inheritance, while the application of class properties is based on a combination of multiple inheritance and automatic class generation. This happens at compile-time, and the programmer has no explicit control over how possible conflicts are resolved. SOM guarantees upward but not downward compatibility.

## NeoClasstalk

NeoClasstalk enables class property reuse by generalizing the parallel inheritance approach of Smalltalk. This approach is interesting because it guarantees downward and upward compatibility while it also enables class property reuse between different hierarchies [RIVA 97, BOUR 98, DUCA 99]. This is accomplished using two main techniques: composition of metaclasses using *code generation* and *dynamic switching* of a class' metaclass.

To illustrate how NeoClasstalk applies class properties, suppose that there is a class B that inherits from a class A and is an instance of the class Meta B. If we now assign a property m to Meta B, the system automatically creates a new metaclass Property m + Meta B (called a *property metaclass*), which inherits from the metaclass

|               | up  | down | non-propagating | composition                               | application                            | control                            |
|---------------|-----|------|-----------------|-------------------------------------------|----------------------------------------|------------------------------------|
| Smalltalk     | yes | yes  | no              | no                                        | no                                     | no                                 |
| CLOS          | yes | yes  | no              | multiple inheritance                      | multiple inheritance                   | multiple inheritance linearization |
| SOM           | yes | no   | no              | multiple inheritance                      | multiple inheritance + code generation | implicit                           |
| NeoClasstalk  | yes | yes  | yes             | code generation + chain of responsibility | code generation + metaclass switching  | implicit                           |
| MetaclassTalk | yes | yes  | yes             | mixin composition                         | mixin inheritance                      | mixin linearization                |
| Traits        | yes | yes  | no              | trait composition                         | trait composition                      | explicit                           |

Table 7.1: Comparison of the different approaches for class property composition

Meta B and defines the property code. It then *changes the class* of B to be that newly created metaclass. Similarly, the metaclass Property n + Meta A is generated automatically when a new property n is assigned to the metaclass Meta A. The resulting hierarchy is illustrated in Figure 7.15.

To reuse class properties, NeoClasstalk stores them in strings on methods of so-called *meta-meta*classes. The actual metaclasses are then generated from these strings. This is illustrated in Figure 7.15, where the light gray area represents the meta level, and the dark gray area is the meta-meta level — the realm of the class properties.

While NeoClasstalk supports non-propagating class properties, it suffers from its intrinsic complexity and the fact that it does not give explicit control over metaclass composition, which is based on code generation. This means for example that potential conflicts are resolved implicitly. The NeoClasstalk model is also not uniform: programming at the meta-meta level is based on manipulating the strings that represent bodies of metaclass methods, which means that it is conceptually different from programming on the meta level and the base level. This is because the relation between the meta-meta-classes and metaclasses is code generation, and not instantiation, as one would expect.

**Summary.** NeoClasstalk provides both downward and upward compatibility, and it allows one to express non-propagating class properties. Property composition is based on code generation and chain of responsibility, while property application is based on code generation and dynamic switching of a class’ metaclass. The NeoClasstalk model is not uniform, and it implicitly resolves conflicts that may occur when metaclasses are composed.

### MetaclassTalk

MetaclassTalk follows the architecture of NeoClasstalk by offering compatibility and property metaclasses. But unlike NeoClasstalk, MetaclassTalk uses mixin composition to compose metaclass properties [BOUR 04]. This makes MetaclassTalk the closest model to our own approach as it guarantees both downward and upward compatibility while allowing the reuse of class properties using a uniform language mechanism. However, by using mixins for composing class properties, this approach suffers from the mixin-related problems that we have described in the problem section of this thesis (cf. Sections 2.1 and 2.2.1).

### 7.4.3 Discussion

As shown in Table 7.1, the only limitation of the trait-based approach is that it does not support non-propagating class properties, a feature that is supported only by NeoClasstalk and MetaclassTalk and comes at the cost of additional complexity, such as the need for a meta-meta level in NeoClasstalk. At the same time, the trait-based approach is the only one that is fully uniform and gives the programmer explicit control over every aspect of class property composition and application.

This explicit control proved to be crucial while developing and applying our class property framework. The main reason for this is that our framework provides several different properties related to instance creation behavior. As a consequence, different class properties define different variants of the same methods (such as

---

new), which often leads to conflicts when they are composed. But these conflicts can be resolved in a *semantically* correct manner only if the programmer has explicit control over the composition. If class properties are represented as traits, this is ensured in the same way as on the base level: by allowing partially ordered compositions, exclusions, and aliases (see Section 7.3.2).

## Chapter 8

# State of the Art

Our publications on traits have triggered a variety of related research as well as implementations and adaptations of traits to other programming languages. In this chapter, we give an overview of this work in general (Section 8.1) and traits in statically typed languages in particular (Section 8.2). We also give an overview of related work that lies outside the borders of the “traits land” (Section 8.3) and briefly compare it to traits.

### 8.1 Traits in Other Languages

Besides our own implementation of traits in the Smalltalk dialect Squeak, there are other languages that feature the traits mechanism presented in this thesis or a variation of it. For example, there are trait implementations available for Perl 5 [WALL 00] and VisualWorks (Cincom) Smalltalk [CINC], while the language Scala [ODER 04] contains traits as a built-in language mechanism.

In this section, we give a brief description of these trait variations, and we compare them to traits and their Squeak implementation described in Chapters 3 and 4. We also outline current plans for integrating traits into Perl 6.

#### 8.1.1 Traits in Perl

Our publications on traits led to an implementation of traits in Perl 5 as well as firm plans for using a trait-like feature called “roles” as a built-in language element of the upcoming Perl 6.

##### Implementation of Traits in Perl 5

Inspired by our initial publication on traits [SCHÄ 03], Stevan Little ported traits to Perl 5. His implementation closely conforms to the description of traits given in this thesis. In particular, it supports all three trait composition operators (sum, alias, exclusion), it allows one to express required methods, and it requires one to explicitly resolve all method conflicts.

Unlike the purely object-based language Smalltalk, Perl 5 is not fundamentally object-oriented. Instead, Perl models objects as references that know what class they belong to. Classes are expressed as packages, and methods are subroutines that expect an object reference to the receiver as the first argument.

Following these principles, Perl traits are also expressed as packages. As an example consider Figure 8.1, which shows the Perl implementation of the trait `TSyncReadWrite` (cf. Section 2.2.2). On the first line, we begin the trait definition by declaring the package where the trait resides (*i.e.*, the name of the trait). The second line declares this package to be a trait by using the package base from the module `Class::Trait`. This is necessary in order to be able to properly resolve all method calls. After declaring the requirements, the trait implements the synchronized versions of the methods `read` and `write`.

```

package TSyncReadWrite;

use Class::Trait 'base';

our @REQUIRES = qw(read write);

sub read {
    my ($self) = @_;
    $self->acquireLock();
    my $result = $self->SUPER::read();
    $self->releaseLock();
    return $result;
}

sub write {
    my ($self, $n) = @_;
    $self->acquireLock();
    $self->SUPER::write($n);
    $self->releaseLock();
}

sub acquireLock() { ... };
sub releaseLock() { ... };

```

Figure 8.1: The trait TSyncRead in Perl

```

package SyncA;

use base ("A");

use Class::Trait ("TSyncReadWrite");

```

Figure 8.2: The class SyncA defined as a subclass of A using the trait TSyncReadWrite

Using this trait the synchronized class SyncA can be derived from the base class A as shown in Figure 8.2. (See Figure 2.7 for the corresponding implementation using C++ templates.) Again, the first line starts the class definition by declaring a new package that is named after the class. Then, we declare that the new class SyncA inherits from the base class A using the trait TSyncReadWrite.

The fundamental differences between Perl and Smalltalk is reflected in the actual implementation of traits in these two languages. A detailed description of the Perl implementation is outside the scope of this thesis. We note only that like the Smalltalk implementation, the Perl implementation is also based on flattening the trait structure at compile-time: *i.e.*, for each relevant trait method, the class' symbol table is extended with an entry that refers to the original trait method.

### Traits as a Language Element of Perl 6

In his series about the unfolding of the Perl 6 design [WALL 04], the creator of Perl gives a description of *roles*<sup>1</sup>, the variation of traits that is planned as a fundamental language element of Perl 6.

One of the main difference between roles and traits is that roles can also contain state, which means that they can define attributes. Another difference is that a role does not only provide some implementation but also

<sup>1</sup>The name “roles” is used because Perl already uses the term “trait” for a language element that modifies the metadata of the classes it is applied to.

defines an interface, which contains the signatures of the role's provided and required (*i.e.*, abstract) methods. Because roles define interfaces, they are also types.

The composition rules of roles are essentially the same as for traits. Composition is unordered, class methods take precedence over role methods, which in turn take precedence over superclass methods. As with traits, method conflicts must be resolved explicitly. However, roles do not support aliases and exclusions. Instead they allow the programmer to access a hidden role method by explicitly using the role name as a scope qualifier (*e.g.*, `Role::method`). Interestingly, the proposal does not specify how conflicting attributes are handled, for example in a diamond situation.

While the Perl 6 design document states that roles are primarily designed to be used at compile-time, they can also be attached to individual objects at runtime.

### 8.1.2 Traits in VisualWorks Smalltalk

VW Traits is an implementation of traits for VisualWorks [CINC] by Terry Raymond. Although VisualWorks is like Squeak a dialect of Smalltalk, there are essential differences between VW Traits and our Squeak based implementation of traits. A major difference is that VW Traits can include state, which means that they can specify instance variables, class instance variables, and shared variables. When a trait is used in a class, the variables defined in the trait will be added to the ones defined in the class. However, unlike with methods, identically named variables are unified and do not cause conflicts. If a class *C* uses two traits *T1* and *T2* that both specify a variable *x*, only one variable *x* is added to the class, and all references to *x* in methods of *T1* and *T2* are bound to this variable. Similarly, if *C* defines or inherits a variable *x* and uses a trait that defines another variable *x*, these two variables are unified.

While unification of variables makes the use of traits defining variables very straightforward, it is somewhat problematic because it can easily lead to unexpected behavior when a class uses two traits that provide identically named variables that are used for different purposes. Because the variables are unified rather than causing a conflict, this problem may not be detected at composition time.

Another distinctive feature of VW Traits is that they support “policy objects” that determine what action to take when particular composition situations occur. By defining additional policies, the programmer can for example specify how to resolve certain conflicts. A drawback of VW Traits is that they do not support the alias operator, which makes it had to implement certain glue methods without code duplication.

Regarding the implementation, the biggest difference between VW Traits and traits in Squeak is that VW Traits do not reuse CompiledMethod objects. Instead, the source code of a trait method is copied to each client class and recompiled in the class when the trait is installed. This means that both the source code and the byte-code of trait methods are duplicated when the trait is applied to a class. The reason for this implementation strategy is twofold. First, the VisualWorks virtual machine does not permit one to execute a method that has not been compiled specifically for the class of the receiver. Second, because methods in a VW Trait can contain instance variable references, they need to be recompiled to update the instance variable offsets.

VW Traits are merged into client classes using a “trait specification”. Unlike our composition clause, which is part of the class definition, this trait specification is a “pragma method” that identifies the trait to be merged and the package that is to contain the merged methods. The trait specification allows the programmer to exclude certain trait methods by adding them to the list of excluded methods.

### 8.1.3 Traits in Scala

Traits are a built-in language mechanism of the language Scala [ODER 04], a modern multi-paradigm programming language designed to express common programming patterns in a concise, elegant, and type-safe way. The traits adaptation of Scala is particularly interesting as Scala is a statically typed language with a type system similar to the ones of Java and C#.

### Declaring and Composing Traits

Scala traits are modeled as abstract classes that do not encapsulate state, neither in the form of variable definitions nor by providing a constructor with parameters. Consequently, trait declarations have the same form as class declarations except that the keyword `class` is replaced by the keyword `trait`. As an example, consider the definition of a trait `Emptiness` providing a method `isEmpty`, which is defined in terms of an abstract method `size`:

```
trait Emptiness {
  def isEmpty: Boolean = size == 0;
  def size: Int;
}
```

In order to apply a number of traits to a class, Scala offers the optional `with` declaration that follows the class name and an optional `extends` declaration specifying the class' superclass. As an illustration, consider the definition of a class `IntSet` that inherits from `ScalaObject` and uses the two traits `Emptiness` and `Testing`:

```
class IntSet extends ScalaObject with Emptiness with Testing {
  ...
}
```

The semantics of `with T1 with T2` is the same as that of `uses T1 + T2` defined in Chapter 3: the composition is symmetric (*i.e.*, the order of the traits does not matter) and conflicts need to be resolved explicitly. Note, however, that conflict resolution is less flexible in Scala because it does not feature exclusion and aliasing.

An interesting feature of Scala is that traits cannot only be composed but can also be inherited, which is a consequence of the fact that Scala traits are just special classes. This means that both classes and traits can be defined as an extension of other traits. For example, Scala allows one to define a trait `B` that inherits from a trait `A` and uses the two traits `U` and `V`:

```
trait B extends A with U with V {
  ...
}
```

The semantics of this construct is the same as if `A` and `B` were classes: local methods in `B` override methods in `U` and `V`, which in turn override methods inherited from `A`. This form of trait inheritance allows the programmer to establish partially ordered compositions of traits: features of trait `B` override all equally named features of trait `A`. It also allows the programmer to use the keyword `super` in the trait `B` to access (overridden) methods defined in `A`, which somewhat compensates for the missing alias operator.

In this context, it is important to note that in Scala, a super-send `super.foo()` that occurs in a trait `B` is only valid if `B` inherits from another trait `A` that implements (or inherits) the method `foo`. If the method `foo` inherited from `A` is abstract, the super-send `super.foo()` in `B` has the semantics known from super-sends in our traits: it will refer to the method `foo` in the superclass of the class to which `B` will eventually be applied.

### Integration into the Type System

The most interesting aspect of the Scala adaptation of traits is the fact that they are fully integrated into Scala's static type system. Because Scala traits are modeled as abstract classes, each trait, like each class, also defines a type. This is important because it means that in Scala, traits without any concrete methods play the roles of interfaces, and Scala therefore does not have a separate notion of interfaces.

The Scala type system supports generics in a very similar but even more expressive way than Java 1.5. This is important for the integration of traits, because it allows the programmer to express generic traits without having to introduce any trait-specific additions to the type system. Instead, generic traits are written in exactly the same way as are generic classes. As an example, consider the fully abstract trait `Set`, which is parameterized with a type parameter `T` that corresponds to the type of the set's elements:

```

trait Set[T] {
  def includes(x: T): Boolean;
  def add(x: T): Set[T];
  ...
}

```

## 8.2 Traits and Static Types

We are currently working on a port of traits to C#/Rotor, the shared source implementation of the .NET Common Language Infrastructure (CLI). Given our background of implementing traits in dynamically typed languages, the biggest challenge is dealing with C#'s static type system.

Although the integration of traits in Scala, the static type system for traits developed by Fisher and Reppy [FISH 03], and the extension of Featherweight Java [IGAR 99] with traits developed by Liquori and Spivack [LIQU 04] provide valuable insights, we want to explore in a more fundamental way the conceptual design decisions that occur when traits are integrated into a statically typed language. In particular, we want to investigate whether and how the language and its type system need to be extended in order to achieve (most of) the expressiveness known from dynamically typed traits.

In this section, we give an overview of important conceptual design decisions and trade-offs related to traits and static types, and we briefly outline how they can be addressed.

### 8.2.1 Relationship between Traits and Types

When integrating traits into a static type system, we must ask ourselves what is the relationship between traits and types. In particular, should each trait define a type? At first glance, this sounds reasonable, especially because in most statically typed languages, classes also define types and the inheritance hierarchy determines the corresponding subtype relation. This corresponds to the design decision taken by Scala (cf. Section 8.1.3), where each trait (and each class) defines a type, and the subtype relation is defined based on both inheritance and trait composition.

However, making each trait be a type also blurs the important conceptual distinction between implementation and interfaces, which leads to two kinds of problem. First, it does not address the fact that in the same way as subclassing does not necessarily imply subtyping [COOK 90], a trait may be composed from another trait without really being a subtype of it. Second, in a nominal type system, we end up with multiple identical types if there are multiple traits (or classes) providing different implementations of the same conceptual interface (e.g., `TRectangle` and `TOptimizedRectangle`).

With traits, such problems appear more frequently than in purely inheritance-based systems because the availability of traits naturally leads to more components and finer-grained compositions. Furthermore, because traits feature the aliasing and exclusion operators, such trait-based types can make no guarantee of signature conformance [QUIT 04]. One solution to this problem would be to restrict the subtype relation so that it does not apply to trait compositions that involve exclusions. Alternatively, one could adopt the solution of Scala, which simply does not allow exclusions and aliasing and therefore avoids the problem in the first place. However, this also means that the programmer's ability to resolve conflicts is limited.

### 8.2.2 Typing Trait Methods

Another issue that arises when traits are implemented in a statically typed language is that it may not always be clear how their methods should be typed. This is especially the case if the type system does not feature parametric polymorphism (e.g., Java prior to version 1.5).

As an illustration, consider the trait `TLinkable` that bundles together the behavior of an element in a linked list and is used in classes such as `Link` and `Process`<sup>2</sup>. Besides others, this trait provides the methods `includes`;

<sup>2</sup>Smalltalk processes are links so that the scheduler can keep them in linked lists.

```

Trait named: #TLinkable

includes: other
  | tail |
  self == other ifTrue: [↑ true].
  tail := self getNext.
  tail ifNil: [↑ false].
  ↑ tail includes: other

checkAndSetNext: other
  (other includes: self) ifTrue: [↑ false].
  self setNext: other.
  ↑ true.

reverse
  | result list temp |
  result := nil.
  list := self.
  [list notNil] whileTrue: [
    temp := list getNext.
    list setNext: result.
    result := list.
    list := temp.
  ].
  ↑ result.

getNext
  self requirement

setNext: other
  self requirement

```

Figure 8.3: The trait TLinkable in the dynamically typed language Smalltalk

checkAndSetNext: and reverse. While includes: checks whether the argument link is included in the linked list starting at the receiver, the method checkAndSetNext: sets the next field of the receiver to the link that is passed as an argument, but only if this does not cause a cycle. Finally, the method reverse reverses the linked list. Figure 8.3 shows the implementation of this trait in Smalltalk.

Because Smalltalk is dynamically typed, the trait TLinkable can be immediately used in the class Link, the class Process, and any other class that is linkable. The only condition is that these classes provide the two required methods getNext and setNext:, which get and set the next element of the list.

Now assume that we would like to write the same trait in a statically typed language such as Java. This immediately raises the question of what type should be used for the arguments, the return values, and the temporary variables of the methods defined in the trait TLinkable so that this trait can be used for both Link and Process as well as any other class that is linkable.

**Using interfaces.** Regarding the argument type of includes:, a possible answer could be that the chosen type should allow every linkable object to be passed as an argument. In a language like Scala, where every trait also defines its own type and trait composition establishes a subtype relationship, one might for example use the type TLinkable as the argument type.

Alternatively, in a language where traits do not define types, one could define a separate interface ILinkable, that would then need to be implemented by all linkable classes. In case of a nominal type system, this requires a certain amount of coding discipline and foresight, but it has the advantage that the typing concern is completely

```

trait TLinkable<T> implements ILinkable {

    public boolean includes(ILinkable other) {
        if (this == other) return true;
        T tail = list.getNext();
        if (tail == null) return false;
        return tail.includes(other);
    }

    public boolean checkAndSet(T other) {
        if (other.includes(this)) return false;
        setNext(other);
        return true;
    }

    public T reverse() {
        T result = null;
        T list = this;
        while (list != null) {
            T temp = list.getNext();
            list.setNext(result);
            result = list;
            list = temp;
        }
        return result;
    }

    public abstract T getNext();
    public abstract void setNext(T other);
}

class Link uses TLinkable<Link> {
    ...
}

class Process uses TLinkable<Process> {
    ...
}

```

Figure 8.4: The generic trait TLinkable used in the classes Link and Process

separated from the implementation concern. As a consequence, classes that are not built from the trait TLinkable because they follow a different implementation strategy can still be passed as an argument to this method as long as they implement the ILinkable interface.

**Using generics.** When it comes to the other types, things are more problematic. As an example, consider the return type of the methods reverse and getNext. Assuming that TLinkable is used only for homogeneous lists, the methods reverse and getNext should return an instance of whatever class they are called on. In particular, this means that an instance of Link (Process) should be returned when these methods are called on a Link (Process).

What makes this situation difficult is that the return types of these methods are in fact parametric; *i.e.*, they depend on the class to which the trait TLinkable is finally applied. Therefore, using an interface such as ILinkable as the return type does not solve our problem because it would only allow a common subset of all the methods in Link and Process to be called on the return values.

```

trait TLinkable implements ILinkable {

    public boolean includes(ILinkable other) {
        if (this == other) return true;
        if (getNext() != null) return getNext().includes(other);
        return false;
    }

    public boolean checkAndSet(ThisType other) {
        if (other.includes(this)) return false;
        setNext(other);
        return true;
    }

    public ThisType reverse() {
        ThisType result = null;
        ThisType list = this;
        while (list != null) {
            ThisType temp = list.getNext();
            list.setNext(result);
            result = list;
            list = temp;
        }
        return result;
    }

    public abstract ThisType getNext();
    public abstract void setNext(ThisType other);
}

```

Figure 8.5: The trait TLinkable implemented using ThisType

The same problem also applies to the argument of the methods `setNext` and `checkAndSetNext`: as well as to the temporary variables used in the method `reverse`. If we for example declared the type of these temporary variables to be `ILinkable`, the type of the list elements would be changed when the list is reversed.

The problem can be addressed using parametric polymorphism as provided by the generics mechanism introduced in Java 1.5 (or the generics mechanisms available in C# and Scala), because it allows us to write the trait `TLinkable` with a type parameter that is then used for the return values, the arguments, and the temporary variables of these methods. Whenever the trait `TLinkable` is applied to a class such as `Link` and `Process`, we can then use the corresponding type as the concrete parameter. The code for this approach is shown in Figure 8.4. Note that it uses the Java 1.5 syntax extended with the keywords `trait` and `uses`, which are used to declare a trait and to apply a trait, respectively.

**Reifying the class that uses a trait.** While such a parameterized implementation of `TLinkable` solves the problem of correctly typing the trait methods, it also adds quite a bit of complexity. In `traits-mini-java` (TMJ) [QUIT 04], an implementation of traits based on a subset of Java, Quitslund therefore proposes to address this problem by reifying the class that actually uses the trait. This means that TMJ features a new keyword `ThisType`, which can be used in a trait to refer to the class where the trait will eventually be used. Using this feature, TMJ allows one to write the trait `TLinkable` without the explicit use of generics (see Figure 8.5).

Compared to the version based on Java generics, this has the advantage that both the trait `TLinkable` and the classes (and traits) using this trait can be written in a way that is simpler and more concise because it does not require an explicit parameter for the class type. Furthermore, the keyword `ThisType` is, unlike type parameters, fully equivalent to the (name of the) class it refers to; in particular, it can be used to construct new instances of the current class, which is for example useful for traits that need to construct a new object of the same class as

the receiver. (In Java and in Scala, it is not permitted to instantiate type parameters since the actual type may correspond to an interface or an abstract class, which can not have instances.)

At the same time, the approach based on `ThisType` is less expressive than generic types and can therefore not in general replace them. As an example, generics are still needed if one would like to write a trait (or a class) that implements a collection or an iterator that can deal with different element types without having to use explicit down-casts.

### 8.2.3 Conclusions

The considerations in this sections have shown that integrating traits into a statically typed language requires one to take design decisions that do not arise in a dynamically typed environment. This is mainly because traits are designed to be applied to a variety of different classes that are usually not known when the traits are written, while static type systems tend to require the programmer to make early assumptions about the types of classes to which a trait can be applied.

This dilemma is not really new. In fact, the same problems arise when mixins are used in a statically typed environment. As shown by Scala, most of these problems are successfully addressed by a notion of generic types. This is particularly encouraging because it means that given a type system with parametric polymorphism, an adequately expressive integration of traits can be naturally achieved without adding any traits-specific features to the type system. Using the Scala approach, it should therefore also be possible to add traits to statically typed languages such as Java 1.5 or C#.

Nevertheless, there remain some trait constructs that can be expressed straightforwardly in a dynamically typed language, but not in a statically typed language featuring Java-like generics. Future research has to show whether it is worth introducing trait-specific features to a static type system to address at least some of these issues.

## 8.3 Other Related Work

In Chapter 2, we have shown how multiple inheritance and different forms of mixins attempt to promote code reuse, and the problems that beset these generalizations of inheritance. In this section we compare traits to some other approaches for structuring complex artifacts.

### Other Reuse Constructs Called “Traits”

Several other systems have used entities called “traits” to share and reuse implementation in ways that are related to the composition mechanism introduced in this dissertation. One of them is the prototype-based language Self [UNGA 87]. In Self, there is no notion of class; each object conceptually defines its own format, methods, and inheritance relations. Objects are derived from other objects by cloning and modification. Objects can have one or more parent objects; messages that are not found in the object are looked for and delegated to a parent object. The order in which these parent objects are searched is not defined<sup>3</sup>, and it is an error for a selector to be found in more than one parent. In Self, explicit sends to such parent objects are called *resends* and there are two different kinds of them: *directed resends* look in a specific parent object, while regular *resends* traverse all parent objects.

While Self does not feature classes, it uses so-called *trait objects* to factor out common features [UNGA 91]. Similar to the notion of traits presented in this thesis, these trait objects are essentially groups of methods. However, unlike our traits, Self’s trait objects do not support specific composition operators; instead, they are used as ordinary parent objects.

<sup>3</sup>In some older versions, Self featured sophisticated mechanisms to influence the search order.

Also the software for the Xerox Star workstation was implemented using entities called traits [CURR 82]. Traits were primitive entities used to build up more complex objects. They were implemented as coding conventions in the Mesa programming language. This approach has more in common with other multiple inheritance approaches than with traits as presented in this thesis. In particular, the Star traits differ from ours in their semantics for inheritance, their provision for conflict resolution capabilities, their ability to carry state, and their multiple implementations for a single method.

The Larch Shared Language [GUTT 85] is also based on a construct called a trait; the relationship turns out to be more than name deep. Larch traits are fragments of specifications that can be freely reused at fine granularity. For example, it is possible to define a Larch trait such as `IsEmpty` that adds a single operation to an existing container datatype specification. But there are also significant differences between Larch traits and the traits presented in this thesis. For example, our traits are not intended to be used to prove properties of programs, and adding a trait to a class does not formally constrain the behavior of existing methods.

### PIE

The Personal Information Environment (PIE) is a programming environment that supports the design, development, and documentation of Smalltalk programs [GOLD 80a, GOLD 80b]. The PIE environment is based on a network of nodes that describe different types of entities — from small pieces such as a single procedure to much larger conceptual entities such as categories of classes or configurations of the system — in a uniform way. PIE features a wide variety of innovations such as context-sensitive descriptions (*i.e.*, properties with different associated values depending on the current context), meta-nodes containing a meta-description of the associated node, and a mechanism for unique identification of objects across an entire computing community [BOBR 80, GOLD 80a, GOLD 80b].

In addition, PIE features a form of multiple inheritance based on a notion of multiple perspectives, which reflects the idea that a certain node may have different characteristics depending on the point of view from which it is considered. Hence, PIE allows the programmer to assign an arbitrary number of different perspective (with independent superclasses) to a certain node. In a first version, the state of the object was represented entirely in the node, and the perspectives carried no state: they supplied method definitions only. Although this early form of perspectives bears a certain resemblance to traits, there are important conceptual differences. The most critical one is that unlike with traits, the methods provided by perspectives are not merged into the node where the perspectives are applied. This means that a node does not directly understand the messages implemented by its perspectives, and that a programmer therefore has to use a message pattern that explicitly states the class of the perspective providing the sent message.

While this design has the advantage that equally named methods of different perspectives never conflict, it also means that external clients depend on the structure of a node, and that perspectives are heavyweight entities that do not provide for fine-grained modularization of a node's methods. These conceptual differences between traits and perspectives are even more significant in the second and most recent version of PIE, where each perspective also carries its own state. Note that the notion of perspectives in PIE is based on very similar notions in FRL [GOLD 77] and KRL [BOBR 77], and that it is related to the approach employed by ThingLab [BORN 81], a multiple inheritance constraint satisfaction system.

### Template-based Approaches

There are several C++ template libraries such as the Standard Template Library (STL) [MUSS 96] and the Boost Lambda Library [JÄRV 03], which implement a variety of parameterized data structures and functions such as collections and iterators. Whereas these parameterized data structures facilitate reuse because they are applicable in the context of different types, they are not directly related to the kind of feature composition traits are aiming for.

Indeed, VanHilst [VANH 96b] and Notkin noted that C++ templates can be used for two conceptually different kinds of parametrization: for *genericity* (*e.g.*, a generic class `Set` with a parameterized element type)

and for *composition* (e.g., a class `Color` with a parameterized superclass)<sup>4</sup>. The conceptual difference between these two purposes of templates become especially apparent in a language like Smalltalk. Because Smalltalk is dynamically typed, templates are not necessary for writing the kind of generic data structures implemented in the STL, and because Smalltalk’s built-in blocks are anonymous functions (i.e., lambda abstractions), there is no need for the abstractions proposed by the Boost Lambda Library. But at the same time, Smalltalk lacks a flexible composition mechanism, which is why we extended the language with traits.

The RESOLVE discipline [SITA 94, OGDE 94, EDWA 94, HOLL 94] for component-based software engineering is a set of software engineering design principles introduced by Hollingsworth in his doctoral dissertation [HOLL 92]. While the RESOLVE discipline is language independent, Sitaraman and Weide also developed specialized versions such as a version for C++ known as RESOLVE/C++ [BUCC 94]. Although the RESOLVE discipline covers many different kinds of software engineering principles such as avoiding aliasing problems by consistently using swapping and not assignment as the basic data movement mechanism, its main focus lies on component-based design, i.e., the RESOLVE framework.

A key concept of the RESOLVE framework is the distinction between *abstract components* (specifications) and *concrete components* (implementations). The distinction between these two kinds of components allows each abstract component to be realized using any of several concrete components that correctly achieve the intended functionality but may for example differ in performance characteristics. RESOLVE components are also parameterized. In Resolve/C++, parametrization is achieved by making each component a C++ template with two kind of parameters: *conceptual parameters* for generic components such as a set that deals with elements of a parameterized type, and *realization parameters* that avoid concrete-to-concrete component coupling.

These realization parameters make the RESOLVE components similar to traits (and mixins), as they allow a programmer to apply and compose RESOLVE components in very flexible ways. Apart from this similarity, there are also significant differences between traits and RESOLVE. The traits mechanism is designed as a simple and lightweight extension of single inheritance that allows one to build classes from a fine-grained composition of traits rather than an unstructured collection of individual methods. This is reflected by the fact that traits are implicitly parameterized and are composed in a way that is quite limited but in return guarantees certain properties, such as the flattening property, which are important for the understandability of such fine-grained structures.

In contrast, RESOLVE components are explicitly parameterized and are then composed and applied by (full or partial) instantiation rather than inheritance. Together with the distinction between abstract and concrete components, this makes the RESOLVE approach more complex and heavyweight — Edwards et al. describe the RESOLVE specification language as “rich and fairly complex” [EDWA 94] — but in return offers a wide variety of different kinds of compositions and conformance guarantees.

### Mixin-related Approaches

GenVoca is a design methodology for creating application families and architecturally extensible software, i.e., software that is customizable via module additions and removals [BATO 92]. With GenVoca, class refinements are modeled as functions that take a program (i.e., a GenVoca constant) as input and produce a feature-augmented program as output. While traits are purely behavioral, a GenVoca class refinement can not only introduce or override methods, but can also add new data members and constructors to a target class. A more fundamental difference between traits and GenVoca is that the focus of traits lies on its composition operators that guarantee important properties for making the resulting classes easy to understand (e.g., the flattening property) and robust with respect to changes (e.g., commutativity and explicit conflict resolution). In contrast, GenVoca’s main innovations are the layering and scaling of refinements that allow one to generate high-performance systems for a target domain. Indeed, the actual implementation of GenVoca refinements is based on existing mechanisms such as mixins [BATO 03].

<sup>4</sup>Note that generics in most other languages such as C# and Java cannot express classes with parameterized superclasses. Therefore, they cannot be used to express mixin-like feature composition.

Mixin layers [SMAR 98] are a technique for implementing layered object-oriented designs (*e.g.*, collaboration-based designs). Mixin layers are similar to mixins but scaled to a multiple-class granularity. Mixin layers address the scalability problems that can appear in role-based designs [VANH 96a, VANH 96b], but they still suffer from the fragility problems we identified for mixins (such as fragility with respect to changes) because they are based on mixins as the fundamental composition mechanism.

Smaragdakis also showed how one can develop layered software using common Unix (Linux and Solaris) dynamic libraries [SMAR 02]. The idea is that, from an object-oriented design standpoint, dynamic libraries are analogous to components in a mixin-based object system. This enables one to use libraries in a layered fashion, mixing and matching different libraries, while ensuring that the result remains consistent. However, as for all the other mixin-based approaches, composition is linear and the composition order is crucial for the semantics of the composition. As a consequence, this form of dynamic library composition also suffers from the mixin-related problems that we have addressed with our work on traits (cf. Chapter 2)

### Aspect-oriented Programming (AOP)

Aspect-oriented programming [KICZ 97] allows the programmer to encapsulate concerns that cross-cut class boundaries in a construct called an aspect. Both aspects and traits can add new methods to existing classes. Aspects can also weave code before or after the execution of a method, an effect traits achieve using method overriding and explicit calls to `super`. In addition, most implementations of aspect-oriented programming such as AspectJ [KICZ 01] support weaving code at more fine-grained join points such as field accesses, which is not supported by traits.

Despite the fact that traits and aspects can be used for similar purposes, there are fundamental differences between the two approaches. By definition, aspects are concerns that cannot be cleanly encapsulated in a generalized procedure (*i.e.*, object, method, mixin). This means that in contrast to traits, aspects are neither designed nor used to build classes and components from scratch, but rather to alter the performance or semantics of the components in systemic ways. A single aspect can also be designed to modify the behavior of methods spread across the object-oriented decomposition, *i.e.*, in many classes. A trait cannot do this.

### Other Modularity and Composition Models

Delegation (also known as “object-based inheritance”) is another form of composition that side-steps many of the problems related to class-based inheritance [KNIE 99]. In contrast to traits, delegation is designed to support *dynamic* component adaptation.

The Jigsaw modularity framework, developed by Bracha in his doctoral dissertation [BRAC 92], defines module composition operators such as merge, rename and restrict that are strikingly similar to our trait sum, alias and exclusion operators. For example, Bracha’s merge, like our sum operator, is commutative. Although there are differences in the details of the definitions (for example, in how conflicts are handled), the more significant differences are in motivation and setting. Jigsaw is intended as a complete framework for module manipulation in the large, and makes assumptions appropriate to that setting: namespaces, declared types and requirements, full renaming, and semantically meaningful nesting. Traits are intended to supplement existing languages by promoting reuse in the small, and consequently do not define namespaces, do not declare types, infer their requirements, do not allow renaming, and do not give a meaning to nesting. The Jigsaw operation set also aims for completeness, whereas in the design of traits we explicitly gave up completeness for simplicity. Nevertheless, the similarity of the core operation sets is encouraging, given that they were defined independently.

Caesar’s collaboration interfaces are similar to traits in that they include the declaration of *expected* methods, *i.e.*, those that classes must provide when bound to an interface [MEZI 02]. Thus, Caesar’s interface concept can simulate traits by binding an interface to a class and then combining it with a specific implementation. However, Caesar has no special compositional construct for dealing with conflicts. Instead, Caesar is designed to use one of the conflict resolution strategies known from multiple inheritance languages such as

C++, leading to problems similar to those described in Chapter 2. Moreover, Caesar is based on explicit wrappers, which can be costly at runtime, while the semantics of traits is compatible with single inheritance and does not cause a run-time penalty.

Mezini also proposed an approach to behavior composition in a class-based environment that is based on the encapsulated object model of class-based inheritance, but introduces an explicit combination layer between objects and classes [MEZI 97]. The definition of the behavior of an evolving object is divided between a class that provides the standard behavior of the object and a set of mixin-like software modules, called adjustments. One of the main differences from traits is that Mezini's approach is more dynamic and complex. In fact, a combiner-metaobject is associated with each evolving object, and is responsible for the compositional aspects of the object's behavior. This means that the combiner-metaobject uses the adjustments to define the environment where the messages sent to the object are evaluated.

Logtalk [MOUR 03] is an open source object-oriented extension to the Prolog programming language. It supports both prototypes and classes. In addition, it supports component-based programming using a mechanism called categories that is designed to share code between classes. Despite a superficial resemblance between Logtalk categories and traits, there are many differences between the two mechanisms. Logtalk does not support aliasing and exclusions, it uses a depth-first lookup to *implicitly* resolve any conflicts, and it suffers from scalability problems as categories cannot be composed from other categories.

## Chapter 9

# Conclusions

In the introduction to this dissertation (Section 1.2), we made the claim that traits provide a better solution to the reuse problems in object-oriented languages than multiple inheritance and mixins. In this final chapter, we first summarize how the contributions presented in this thesis support this statement (Section 9.1). Then, we mention open issues and limitations of traits, which lead us to directions for future work (Section 9.2).

### 9.1 Contributions

In Chapter 2 of this dissertation, we have given a detailed description of composition and decomposition problems that arise with existing forms of inheritance — single inheritance, multiple inheritance and mixin inheritance — in object oriented programming languages.

To overcome these problems, we presented traits (Chapter 3), a simple compositional mechanism for building and structuring object-oriented programs. Based on a formal model, graphical diagrams and a running examples, we showed how traits are composed using a set of composition operators — symmetric combination, exclusion, and alias — that are carefully designed so that they allow a fair amount of composition flexibility without being subject to the problems and limitations that we have identified. In particular, we illustrate and discuss how traits allow us to combine multiple wrappers without the fragility problems that we identified for mixins or the need for duplication of glue methods that arises with most forms of multiple inheritance (Sections 3.2 and 3.3).

The formal model shows that trait composition adheres to the flattening property (Sections 3.1.3 and 3.1.6), which says that a non-overridden method in a trait has the same semantics as if it were implemented directly in the class using the trait. This is an important contribution because it indicates that even extensive use of traits never does any harm: the flattening property guarantees that the resulting code is not less understandable than the original, because it is always possible to both view and edit the code as if it were written using single inheritance alone. Because of this property, we believe that traits are an ideal extension for single inheritance languages.

The description of our traits implementation in Squeak Smalltalk shows that traits can be implemented very efficiently while avoiding many of the difficulties known from multiple inheritance and mixin implementations (Chapter 4). This is because the absence of variables in traits and the flattening property allow us to perform all the additional computations when traits are composed, which means that there is almost no performance penalty at runtime; the only penalty is due to the use of accessor methods, but such methods are widely used even without traits because they improve maintainability. This is especially remarkable since our implementation did not require any changes to the Squeak virtual machine, does not duplicate any source code and duplicates the byte-code of a method only if it contains sends to super.

Our experience has shown that without appropriate tools the programmer cannot really benefit from the

theoretical advantages of traits (Section 5.1). In our Squeak-based implementation, we therefore introduced the trait browser, which addresses the requirements that we identified as critical for programming with traits: it allows the programmer to switch seamlessly between the different views, it supports the programmer in understanding composite entities by emphasizing the glue methods that define how the classes and traits are connected, and it facilitates incremental and iterative programming by keeping track of incomplete compositions and by indicating what needs to be done to make the program or at least a part of it complete (Section 5.2). The trait browser has proven to be crucial for giving the programmer the maximum benefit from traits. For example, the programmer can only benefit from the flattening property if there are tools that actually provide the different views (*i.e.*, the flat class-based view and the structured trait-based views) and allow the programmer to switch between them.

Based on our experience in programming with traits and the trait browser, we developed a methodology around the usage of traits (Section 5.3). This is a valuable contribution because a new language construct is only useful if the programmers actually understand how to (and how not to) use it. If programmers start using traits, it is therefore important to tell them how traits affect the programming process.

Being guided by this methodology, we performed two case studies: the refactoring of the Smalltalk collection classes (Chapter 6) and the development of a trait-based approach to composing class properties (Chapter 7). These case studies and their results not only confirm the practical usefulness of our programming tools and the soundness of our methodology, they are also a strong indication that traits can be used for realistic problems. In particular, the case studies showed that traits are suitable for modularizing classes that are already built, and that they raise the level of abstraction when building new classes. As we worked with the refactored collection hierarchy, we were especially impressed with the practical usefulness of the flattening property, which made understanding classes that are built from composite traits quite a simple matter.

The various approaches for typing traits and the adaptations of traits that were implemented in other programming languages (Chapter 8) show that traits have had significant impact in the programming languages community. In the end, this is another indication for the scientific and practical relevance of the work presented in this dissertation.

## 9.2 Future Work

In this section we present several ideas that have not yet been realized or implemented, largely due to a lack of time and human resources.

**Traits and types.** Although existing implementations of traits in languages such as Scala show that traits can be applied to statically typed languages, there remain several open questions that arise when traits are integrated into a statically typed language. For example:

- should traits, like classes, also define corresponding types?
- if traits define types, what should be the impact of the different composition operators on the subtype relationships?
- what does an ideal generics mechanism for traits look like?

In this thesis, we gave an overview of such conceptual questions, the associated trade-offs, and possible solutions, but more research is necessary to give final answers.

**Traits and encapsulation.** A critical limitation of trait is that they offer no real solution to the problem of unintended name clashes: when two traits are composed, it may be that each requires a semantically different method that happens to have the same name. While this may at first look like a trait-related problem, we believe that it is in fact a reflection of a more general problem: insufficient encapsulation support in most object-oriented programming languages [SCHÄ 04a, SCHÄ 04b]. Indeed, a closer look shows that the same

kind of problem also arises in languages like Smalltalk and Java, where unintended name clashes can occur when classes inherit from each other, or when a class implements two interfaces that each specify a method with precisely the same signature but with different semantics.

We therefore suggest that these problems should be solved with a uniform encapsulation mechanism such as our mechanism based on encapsulation policies [SCHÄ 04a, SCHÄ 04b]. While this mechanism was designed with traits in mind, we have so far only applied it to traditional single inheritance languages like Smalltalk; future work has to show whether it leads to the expected results when applied to traits.

**Traits and classboxes.** Bergel *et al.* developed classboxes, a module system for object-oriented programming languages that supports method addition and replacement [BERG 03]. In addition, classboxes feature local rebinding, which means that changes made within a classbox are local and do not affect other classboxes. This is achieved using class extensions. A class extension is defined in a class box and it specifies methods that are added to the definition of a class when it is imported from another classbox.

Although traits and classboxes address different kind of modularity problems, we believe that combining them could lead to interesting synergies. For example, it would be interesting to see whether and how class extensions could be modeled as traits.

**And more...** Other interesting areas for future work include investigating the consequences of allowing traits to specify state variables, extending trait composition so that it can replace inheritance, and evaluating the use of traits to modify the behavior of individual instances at run-time.

# Bibliography

- [ALPE 98] S. R. Alpert, K. Brown, and B. Woolf. *The Design Patterns Smalltalk Companion*. Addison Wesley, 1998. (pp 73, 95, 97)
- [AMER 90] P. America and F. van der Linden. *A Parallel Object-Oriented Language with Inheritance and Subtyping*. In Proceedings OOPSLA/ECOOP '90, pages 161–168. ACM Press, October 1990. (p 5)
- [ANCO 00] D. Ancona, G. Lagorio, and E. Zucca. *Jam — a Smooth Extension of Java with Mixins*. In Proceedings ECOOP 2000, pages 145–178. Springer Verlag, 2000. (pp 1, 2, 9, 34)
- [BAK 02] L. Bak, G. Bracha, S. Grarup, R. Griesemer, D. Griswold, and U. Hölzle. *Mixins in Strongtalk*. In Proceedings ECOOP 2002 Workshop on Inheritance, June 2002. (pp 9, 34)
- [BATO 92] D. Batory and S. O'Malley. *The Design and Implementation of Hierarchical Software Systems With Reusable Components*. ACM Transactions on Software Engineering and Methodology, October 1992. (p 116)
- [BATO 03] D. Batory, J. N. Sarvela, and A. Rauschmayer. *Scaling Step-wise Refinement*. In Proceedings ICSE 2003, pages 187–197. IEEE Computer Society Press, 2003. (p 116)
- [BECK 00] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison Wesley, 2000. (pp 65, 66)
- [BERG 03] A. Bergel, S. Ducasse, and R. Wuyts. *Classboxes: A Minimal Module Model Supporting Local Rebinding*. In Proceedings JMLC 2003 (Joint Modular Languages Conference), pages 122–131. Springer Verlag, 2003. (p 121)
- [BLAC 03] A. P. Black, N. Schärli, and S. Ducasse. *Applying Traits to the Smalltalk Collection Hierarchy*. In Proceedings OOPSLA 2003, pages 47–64. ACM Press, October 2003. (p 31)
- [BLAC 04] A. P. Black and N. Schärli. *Traits: Tools and Methodology*. In Proceedings ICSE 2004, pages 676–686. ACM Press, Mai 2004. (p 48)
- [BOBR 77] D. G. Bobrow and T. Winograd. *An Overview of KRL, a Knowledge Representation Language*. Cognitive Science, vol. 1, no. 1, pages 3–46, 1977. (p 115)
- [BOBR 80] D. G. Bobrow and I. P. Goldstein. *Representing Design Alternatives*. In Proceedings of the Conference on Artificial Intelligence and the Simulation of Behavior, July 1980. (pp 48, 49, 115)
- [BORN 81] A. Borning. *The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory*. ACM Transactions on Programming Languages and Systems, vol. 3, no. 4, pages 353–387, October 1981. (p 115)

- [BOUR 98] N. M. N. Bouraqadi-Saadani, T. Ledoux, and F. Rivard. *Safe Metaclass Programming*. In Proceedings OOPSLA '98, pages 84–96. ACM Press, 1998. (pp 44, 90, 99, 100, 103)
- [BOUR 04] N. Bouraqadi. *Safe Metaclass Composition Using Mixin-Based Inheritance*. Elsevier Journal of Computer Languages, Systems and Structures, vol. 30, no. 1–2, pages 49–61, April 2004. (pp 100, 104)
- [BRAC 90] G. Bracha and W. Cook. *Mixin-based Inheritance*. In Proceedings OOPSLA/ECOOP '90, pages 303–311, October 1990. (pp 1, 2, 7, 37)
- [BRAC 92] G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, Dept. of Computer Science, University of Utah, March 1992. (pp 8, 60, 87, 117)
- [BRAC 93] G. Bracha and D. Griswold. *Strongtalk: Typechecking Smalltalk in a Production Environment*. In Proceedings OOPSLA '93, pages 215–230. ACM Press, October 1993. (p 87)
- [BRUC 95] K. B. Bruce, L. Cardelli, G. Castagna, T. H. O. Group, G. T. Leavens, and B. Pierce. *On Binary Methods*. Theory and Practice of Object Systems, vol. 1, no. 3, pages 221–242, 1995. (p 5)
- [BUCC 94] P. Bucci, J. E. Hollingsworth, J. Krone, and B. W. Weide. *Part III: Implementing Components in RESOLVE*. ACM SIGSOFT Software Engineering Notes, vol. 19, no. 4, pages 40–51, 1994. (p 116)
- [CAST 95] G. Castagna. *Covariance and Contravariance: Conflict Without a Cause*. ACM Transactions on Programming Languages and Systems, vol. 17, no. 3, pages 431–447, 1995. (p 5)
- [CINC] Cincom. *VisualWorks Smalltalk*. <http://www.cincom.com/scripts/smalltalk.dll/>. (pp 106, 108)
- [COIN 87] P. Cointe. *Metaclasses are First Class: the ObjVlisp Model*. In Proceedings OOPSLA '87, pages 156–167. ACM Press, December 1987. (pp 39, 90, 91)
- [COOK 87] S. Cook. *OOPSLA '87 Panel P2: Varieties of Inheritance*. In Addendum to the Proceedings OOPSLA '87, pages 35–40. ACM Press, October 1987. (p 1)
- [COOK 89] W. Cook and J. Palsberg. *A Denotational Semantics of Inheritance and its Correctness*. In Proceedings OOPSLA '89, pages 433–443. ACM Press, October 1989. (p 5)
- [COOK 90] W. Cook, W. Hill, and P. Canning. *Inheritance is not Subtyping*. In Proceedings POPL '90, pages 125–135. ACM Press, January 1990. (pp 5, 110)
- [COOK 92] W. R. Cook. *Interfaces and Specifications for the Smalltalk-80 Collection Classes*. In Proceedings OOPSLA '92, pages 1–15. ACM Press, October 1992. (pp 60, 67, 72, 85)
- [CURR 82] G. Curry, L. Baer, D. Lipkie, and B. Lee. *TRAITS: an Approach to Multiple Inheritance Subclassing*. In Proceedings of the ACM SIGOA conference on Office Information Systems, pages 1–9, June 1982. (p 115)
- [DANF 94] S. Danforth and I. R. Forman. *Derived Metaclass in SOM*. In Proceedings of TOOLS EUROPE '94, pages 63–73. Prentice Hall Press, 1994. (pp 39, 90, 100)
- [DIXO 89] R. Dixon, T. McKee, M. Vaughan, and P. Schweizer. *A Fast Method Dispatcher for Compiled Languages with Multiple Inheritance*. In Proceedings OOPSLA '89, pages 211–214. ACM Press, October 1989. (pp 2, 5, 47)
- [DUCA 99] S. Ducasse. *Evaluating Message Passing Control Techniques in Smalltalk*. Journal of Object-Oriented Programming (JOOP), vol. 12, no. 6, pages 39–44, June 1999. (p 103)

- [DUCA 05] S. Ducasse, N. Schaerli, and R. Wuyts. *Uniform and Safe Metaclass Composition*. Elsevier Journal of Computer Languages, Systems and Structures, 2005. (pp 44, 90)
- [DUCO 92] R. Ducournau, M. Habib, M. Huchard, and M. Mugnier. *Monotonic Conflict Resolution Mechanisms for Inheritance*. In Proceedings OOPSLA '92, pages 16–24. ACM Press, October 1992. (p 15)
- [DUGG 01] D. Duggan and C.-C. Techaubol. *Modular Mixin-Based Inheritance for Application Frameworks*. In Proceedings OOPSLA 2001, pages 223–240. ACM Press, October 2001. (p 7)
- [EDWA 94] S. H. Edwards, W. D. Heym, T. J. Long, M. Sitaraman, and B. W. Weide. *Part II: specifying components in RESOLVE*. ACM SIGSOFT Software Engineering Notes, vol. 19, no. 4, pages 29–39, 1994. (p 116)
- [FISH 03] K. Fisher and J. Reppy. *Statically Typed Traits*. Technical Report TR-2003-13, University of Chicago, Department of Computer Science, December 2003. (p 110)
- [FLAT 98] M. Flatt, S. Krishnamurthi, and M. Felleisen. *Classes and Mixins*. In Proceedings POPL '98, pages 171–183. ACM Press, 1998. (pp 1, 2, 7)
- [FORM 99] I. R. Forman and S. Danforth. *Putting Metaclasses to Work: A New Dimension in Object-Oriented Programming*. Addison-Wesley, 1999. (pp 39, 90, 103)
- [GAMM 95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995. (p 71)
- [GOLD 77] I. P. Goldstein and R. B. Roberts. *NUDGE, a Knowledge-Based Scheduling Program*. In Proceedings of the Fifth International Joint Conference on Artificial Intelligence, pages 257–263, 1977. (p 115)
- [GOLD 80a] I. P. Goldstein and D. G. Bobrow. *Extending Object-Oriented Programming in Smalltalk*. In Proceedings of the Lisp Conference, pages 75–81, August 1980. (p 115)
- [GOLD 80b] I. P. Goldstein and D. G. Bobrow. *Descriptions for a Programming Environment*. In Proceedings of the First Annual Conference of the National Association for Artificial Intelligence, August 1980. (p 115)
- [GOLD 83] A. Goldberg and D. Robson. *Smalltalk 80: the Language and its Implementation*. Addison Wesley, Mai 1983. (pp 39, 42, 65, 67)
- [GOLD 89] A. Goldberg and D. Robson. *Smalltalk-80: the Language*. Addison Wesley, 1989. (pp 90, 91, 101)
- [GRAU 89] N. Graube. *Metaclass Compatibility*. In Proceedings OOPSLA '89, pages 305–316, October 1989. (pp 44, 91, 99)
- [GUTT 85] J. V. Guttag, J. J. Horning, and J. M. Wing. *The Larch Family of Specification Languages*. IEEE Transactions on Software Engineering, vol. 2, no. 5, pages 24–36, September 1985. (p 115)
- [HARR 93] W. Harrison and H. Ossher. *Subject-Oriented Programming (A Critique of Pure Objects)*. In Proceedings OOPSLA '93, pages 411–428, October 1993. (p 5)
- [HOLL 92] J. E. Hollingsworth. *Software Component Design-for-Reuse: A Language Independent Discipline Applied to Ada*. PhD thesis, Dept. of Computer & Information Science, The Ohio State University, Columbus, OH, 1992. (p 116)

- [HOLL 94] J. E. Hollingsworth, S. Sreerama, B. W. Weide, and S. Zhupanov. *Part IV: RESOLVE components in Ada and C++*. ACM SIGSOFT Software Engineering Notes, vol. 19, no. 4, pages 52–63, 1994. (p 116)
- [HOLZ 04] S. Holzner. Eclipse. O'Reilly, Mai 2004. (p 48)
- [IGAR 99] A. Igarashi, B. C. Pierce, and P. Wadler. *Featherweight Java: a Minimal Core Calculus for Java and GJ*. In Proceedings OOPSLA '99, pages 132–146. ACM Press, November 1999. (p 110)
- [INGA 76] D. Ingalls. *The Smalltalk-76 Programming System Design and Implementation*. In Proceedings POPL '76, pages 9–16. ACM Press, 1976. (pp 39, 90)
- [INGA 97] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. *Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself*. In Proceedings OOPSLA '97, pages 318–326. ACM Press, November 1997. (pp 3, 21, 39)
- [JÄRV 03] J. Järvi, G. Powell, and A. Lumsdaine. *The Lambda Library: Unnamed Functions in C++*. Software – Practice and Experience, vol. 33, no. 3, pages 259–291, 2003. (p 115)
- [JETL 99] N. Jetly. *VisualAge for Java 2.0*. Java Developer's Journal, vol. 4, no. 4, pages 48–49, April 1999. (p 48)
- [KEEN 89] S. E. Keene. Object-Oriented Programming in Common-Lisp. Addison Wesley, 1989. (p 1)
- [KICZ 91] G. Kiczales, J. des Rivières, and D. G. Bobrow. The Art of the Metaobject Protocol. MIT Press, 1991. (pp 39, 90, 100, 102)
- [KICZ 97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. *Aspect-Oriented Programming*. In Proceedings ECOOP '97, pages 220–242. Springer Verlag, June 1997. (p 117)
- [KICZ 01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. *An Overview of AspectJ*. In Proceeding ECOOP 2001. Springer Verlag, 2001. (p 117)
- [KNIE 99] G. Kniesel. *Type-Safe Delegation for Run-Time Component Adaptation*. In Proceedings ECOOP '99, pages 351–366. Springer Verlag, June 1999. (p 117)
- [LALO 89] W. R. LaLonde. *Designing Families of Data Types Using Exemplars*. ACM Transactions on Programming Languages and Systems, vol. 11, no. 2, pages 212–248, April 1989. (p 5)
- [LALO 90] W. LaLonde and J. Pugh. Inside Smalltalk: Volume 1. Prentice Hall, 1990. (pp 65, 71)
- [LALO 91] W. LaLonde and J. Pugh. *Subclassing  $\neq$  Subtyping  $\neq$  Is-a*. Journal of Object-Oriented Programming (JOOP), vol. 3, no. 5, pages 57–62, January 1991. (p 5)
- [LAWL 89] J. A. Lawless and M. M. Milner. Understanding Clos the Common Lisp Object System. Digital Press, 1989. (p 15)
- [LEDO 96] T. Ledoux and P. Cointe. *Explicit Metaclasses as a Tool for Improving the Design of Class Libraries*. In Proceedings ISOTAS '96, pages 38–55, March 1996. (p 90)
- [LIEN 04] A. Lienhard. Bootstrapping Traits. Master's thesis, University of Bern, 2004. (p 39)
- [LIQU 04] L. Liquori and A. Spiwack. Adding Multiple Inheritance to Featherweight Java. 2004. (p 110)
- [MADS 90] O. L. Madsen, B. Magnusson, and B. Moller-Pedersen. *Strong Typing of Object-Oriented Languages Revisited*. In Proceedings OOPSLA/ECOOP '90, pages 140–150, October 1990. (p 5)

- [MENS 96] T. Mens and M. van Limberghen. *Encapsulation and Composition as Orthogonal Operators on Mixins: a Solution to Multiple Inheritance Problems*. Object Oriented Systems, vol. 3, no. 1, pages 1–30, 1996. (pp 1, 2, 7)
- [MEYE 88] B. Meyer. *Object-oriented Software Construction*. Prentice-Hall, 1988. (pp 1, 7)
- [MEYE 92] B. Meyer. *Eiffel: The Language*. Prentice-Hall, 1992. (p 9)
- [MEYE 97] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Second edition, 1997. (pp 9, 33)
- [MEZI 97] M. Mezini. *Dynamic Object Evolution without Name Collisions*. In Proceedings ECOOP '97, pages 190–219. Springer Verlag, June 1997. (p 118)
- [MEZI 02] M. Mezini and K. Ostermann. *Integrating Independent Components with On-Demand Remodularization*. In Proceedings OOPSLA 2002, pages 52–67. ACM Press, November 2002. (p 117)
- [MOON 86] D. A. Moon. *Object-Oriented Programming with Flavors*. In Proceedings OOPSLA '86, pages 1–8. ACM Press, November 1986. (pp 1, 2, 7, 37)
- [MOUR 03] P. Moura. *Logtalk*. PhD thesis, Universidade da Beira Interior, 2003. (p 118)
- [MULE 95] P. Mulet, J. Malenfant, and P. Cointe. *Towards a Methodology for Explicit Composition of MetaObjects*. In Proceedings OOPSLA '95, pages 316–330. ACM Press, October 1995. (p 100)
- [MUSS 96] D. R. Musser and A. Saini. *STL Tutorial and Reference Guide*. Addison Wesley, 1996. (pp 37, 115)
- [ODER 04] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. M. and Nikolay Mihaylov, M. Schinz, E. Stenman, and M. Zenger. *An Overview of the Scala Programming Language*. Technical Report 64, École Polytechnique Fédérale de Lausanne, 1015 Lausanne, Switzerland, 2004. (pp 106, 108)
- [OGDE 94] W. F. Ogden, M. Sitaraman, B. W. Weide, and S. H. Zweben. *Part I: the RESOLVE Framework and Discipline: a Research Synopsis*. ACM SIGSOFT Software Engineering Notes, vol. 19, no. 4, pages 23–28, 1994. (p 116)
- [PAEP 93] A. Paepcke. *User-Level Language Crafting*. In *Object-Oriented Programming: the CLOS perspective*, pages 66–99. MIT Press, 1993. (p 15)
- [QUIT 04] P. J. Quitslund. *Java Traits — Improving Opportunities for Reuse*. Technical Report CSE-04-005, OGI School of Science & Engineering, Beaverton, Oregon, USA, September 2004. (pp 110, 113)
- [RIEL 96] A. J. Riel. *Object-Oriented Design Heuristics*. Addison Wesley, 1996. (p 18)
- [RIVA 97] F. Rivard. *Évolution du comportement des objets dans les langages à classes réflexifs*. PhD thesis, Ecole des Mines de Nantes, Université de Nantes, France, 1997. (p 103)
- [SAKK 89] M. Sakkinen. *Disciplined Inheritance*. In Proceedings ECOOP '89, pages 39–56. Cambridge University Press, July 1989. (pp 2, 7)
- [SAKK 92] M. Sakkinen. *The Darker Side of C++ Revisited*. Structured Programming, vol. 13, no. 4, pages 155–177, 1992. (p 7)
- [SCHÄ 03] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. *Traits: Composable Units of Behavior*. In Proceedings ECOOP 2003, pages 248–274. Springer Verlag, July 2003. (pp 16, 106)

- [SCHÄ 04a] N. Schärli, S. Ducasse, O. Nierstrasz, and R. Wuyts. *Composable Encapsulation Policies*. In Proceedings ECOOP 2004, pages 26–50. Springer Verlag, June 2004. (pp 34, 120, 121)
- [SCHÄ 04b] N. Schärli, A. P. Black, and S. Ducasse. *Object-oriented Encapsulation for Dynamically Typed Languages*. In Proceedings OOPSLA 2004, pages 130–149. ACM Press, October 2004.
- [SCHÄ 04c] N. Schärli and A. P. Black. *A Browser for Incremental Programming*. Elsevier Computer Languages, Systems and Structures, vol. 30, pages 79–95, 2004. (p 50)
- [SITA 94] M. Sitaraman and B. Weide. *Component-based software using RESOLVE*. ACM SIGSOFT Software Engineering Notes, vol. 19, no. 4, pages 21–22, 1994. (p 116)
- [SMAR 98] Y. Smaragdakis and D. Batory. *Implementing Layered Design with Mixin Layers*. In Proceedings ECOOP '98, pages 550–570. Springer Verlag, July 1998. (pp 13, 35, 37, 117)
- [SMAR 00] Y. Smaragdakis and D. Batory. *Mixin-Based Programming in C++*. In 2nd Symposium on Generative and Component-Based Software Engineering (GCSE 2000), 2000. (pp 13, 35, 37)
- [SMAR 02] Y. Smaragdakis. *Layered Development with (Unix) Dynamic Libraries*. In Proceedings ICSR 2002, pages 33–45. Springer Verlag, 2002. (p 117)
- [SNYD 86] A. Snyder. *Encapsulation and Inheritance in Object-Oriented Programming Languages*. In Proceedings OOPSLA '86, pages 38–45, November 1986. (pp 15, 34, 35, 51)
- [SNYD 87] A. Snyder. *Inheritance and the Development of Encapsulated Software Systems*. In Research Directions in Object-Oriented Programming, pages 165–188. MIT Press, 1987. (p 15)
- [STRO 86] B. Stroustrup. *The C++ Programming Language*. Addison Wesley, 1986. (p 1)
- [STRO 97] B. Stroustrup. *The C++ Programming Language*. Addison Wesley, Third edition, 1997. (pp 9, 11, 33, 37)
- [SWEE 99] P. F. Sweeney and J. Y. Gil. *Space and Time-efficient Memory Layout for Multiple Inheritance*. In Proceedings OOPSLA '99, pages 256–275. ACM Press, 1999. (pp 2, 5)
- [TAIV 96] A. Taivalsaari. *On the Notion of Inheritance*. ACM Computing Surveys, vol. 28, no. 3, pages 438–479, September 1996. (pp 1, 5)
- [TARR 99] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr. *N Degrees of Separation: Multi-dimensional Separation of Concerns*. In Proceedings of ICSE '99, pages 107–119, 1999. (p 5)
- [UNGA 87] D. Ungar and R. B. Smith. *Self: The Power of Simplicity*. In Proceedings OOPSLA '87, pages 227–242. ACM Press, December 1987. (p 114)
- [UNGA 91] D. Ungar, C. Chambers, B.-W. Chang, and U. Holzle. *Organizing Programs without Classes*. LISP and SYMBOLIC COMPUTATION: An international journal, vol. 4, no. 3, 1991. (p 114)
- [VANH 96a] M. VanHilst and D. Notkin. *Using C++ Templates to Implement Role-Based Designs*. In JSSST International Symposium on Object Technologies for Advanced Software, pages 22–37. Springer Verlag, 1996. (pp 13, 35, 37, 117)
- [VANH 96b] M. VanHilst and D. Notkin. *Using Role Components to Implement Collaboration-Based Designs*. In Proceedings OOPSLA '96, pages 359–369. ACM Press, 1996. (pp 13, 35, 37, 115, 117)
- [WALL 00] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl*. O'Reilly & Associates, Inc., 3rd edition, 2000. (p 106)

- [WALL 04] L. Wall. *Apocalypse 12*, April 2004. <http://www.perl.com/pub/a/2004/04/16/a12.html>. (p 107)
- [WEGN 88] P. Wegner and S. B. Zdonik. *Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like*. In Proceedings ECOOP '88, pages 55–77. Springer Verlag, August 1988. (p 1)

# Curriculum Vitae

## Personal Information

|                       |                        |
|-----------------------|------------------------|
| <i>Name</i>           | Nathanael Schärli      |
| <i>Citizenship</i>    | Zell/LU, Switzerland   |
| <i>Date of Birth</i>  | August 10, 1976        |
| <i>Place of Birth</i> | Solothurn, Switzerland |

## Education

|                            |                                                                                                                     |
|----------------------------|---------------------------------------------------------------------------------------------------------------------|
| <i>Nov 2001 – Feb 2005</i> | Ph.D. in Computer Science at the Software Composition Group, University of Berne, Switzerland                       |
| <i>Oct 1999 – Oct 2001</i> | Master of Computer Science at the Software Composition Group, University of Berne, Switzerland                      |
| <i>Oct 1996 – Sep 1999</i> | Undergraduate Degree in Computer Science at the University of Berne, Switzerland. Minors in Mathematics and Physics |
| <i>Aug 1991 – Feb 1996</i> | Gymnasium (Typus C) at the Kantonsschule Solothurn, Switzerland                                                     |

## Professional Experience

|                            |                                                               |
|----------------------------|---------------------------------------------------------------|
| <i>Jun 2001 – Aug 2001</i> | Intern at Walt Disney Imagineering, Los Angeles, California   |
| <i>May 2000 – Sep 2000</i> | Intern at Walt Disney Imagineering, Los Angeles, California   |
| <i>Feb 1999 – Jul 1999</i> | Software Engineer at Sherpa’x AG, Solothurn, Switzerland      |
| <i>Aug 1997 – Aug 1998</i> | Software Engineer at Ascom Infrasy AG, Solothurn, Switzerland |
| <i>Mar 1996 – Aug 1996</i> | Intern at Ascom Infrasy AG, Solothurn, Switzerland            |