Bootstrapping Traits

Diplomarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Adrian Lienhard

2004

Leiter der Arbeit: Prof. Dr. S. Ducasse
Institut für Informatik und angewandte Mathematik

The address of the author:

Adrian Lienhard
Freiburgstr. 76
CH-3008 Bern
lienhard@iam.unibe.ch

Software Composition Group
University of Bern
Institute of Computer Science and Applied Mathematics
Neubrückstrasse 10
CH-3012 Bern

Abstract

Despite the undisputed prominence of inheritance as the fundamental reuse mechanism in object-oriented programming languages, the variants – single inheritance, multiple inheritance, and mixin inheritance – all suffer from conceptual and practical problems.

Traits overcome the problems arising with the different variants of inheritance. Traits are essentially groups of methods that serve as building blocks for classes and are primitive units of code reuse. In this model, classes are composed from a set of traits by specifying glue code that connects the traits together and accesses the necessary state.

This thesis discusses the implementation of traits. The result it presents is a new Smalltalk kernel bootstrapped with traits. The implementation is fully done in Squeak [INGA 97], an open-source dialect of Smalltalk. It is planned that the next generation of Squeak will include traits.

Because traits are simple and completely backward compatible with single inheritance, implementing traits in a reflective single inheritance language like Squeak is unproblematic.

However, an implementation with a sophisticated and clean design, with the robustness to be used in production and the flexibility to be used as a vehicle for future research, is not trivial. Furthermore our work is aimed at serving as a reference implementation for the introduction of traits in other languages. Hence, we focused on building a simple but powerful system for the future.

Consequently following the fundamental idea of a reflective language – using the features of the language to define the behavior of the language itself – we bootstrapped the new kernel which, eventually, allowed us to fully express the system itself with traits.

The refactoring of the core of the Smalltalk language as a composition of traits not only improved its quality but also enhanced its understandability. This has the advantage that it is easier maintainable and it facilitates experimentation with the language because the different aspects of the kernel are now available as traits and can therefore be recomposed to create new kernel classes with different properties.

Acknowledgements

I'd like to thank all the people who were, directly or indirectly, involved in this work.

First I wish to thank my supervisor Prof. Dr. Stéphane Ducasse for his guidance and that he motivated me to learn Smalltalk and join the SCG.

I'd also like to thank Prof. Dr. Oscar Nierstrasz, head of the SCG, for giving me the opportunity to work in his group and for the careful reading of this thesis and the constructive comments that helped me to improve it.

Special thanks to Nathanael Schärli for many fruitful discussions, pair programming sessions and taking the time to review my code.

Thanks also to all the other members of the SCG and to the fellow students from the lab, especially Markus Kobel, Thomas Bühler and Michael Meer, for providing a pleasant working atmosphere. Special thanks go to Christoph Wysseier, my friend and business partner of netstyle.ch, for his extra commitment during the time-consuming periods of my studies.

Finally I would like to thank my family and friends, and especially my love Felicia Flicker, for being in my life and for their appreciation for what I do.

Thank you all!

Adrian Lienhard
November 2004

Contents

List of Figures

Chapter 1

Introduction

"Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away." Antoine de Saint Exupery

Traits are essentially groups of methods that serve as building blocks for classes and are primitive units of code reuse. In this model, classes are composed of a set of traits by specifying glue code that connects the traits together and accesses the necessary state. Traits overcome the problems arising with the different variants of inheritance, such as single inheritance, multiple inheritance, and mixin inheritance.

During the research on traits, a prototype had been implemented in Squeak. As an instrument for experiments and as proof-of-concept it was not integrated with the host language. As a consequence, the design of this prototype is brittle and it cannot run on newer versions of Squeak, drastically limiting the accessibility of traits for external users.

The goal of the new work is to build up from ground a stable implementation with a clean architecture. As a consequence we go a step further and build the new language kernel with traits itself.

The new implementation has the following purposes: on one hand, it is targeted on being employed in productive systems and integrated in the standard distribution of Squeak, and at the same time, it will be used as a research vehicle for further development of the traits model. Furthermore it serves as a reference implementation for the introduction of traits in other languages.

1.1 Bootstrapping the Traits Kernel

When we analyzed the old prototype implementation and started designing the new class hierarchy – for which we decided to follow the structure of the traditional kernel – it became clear that there is no entirely satisfactory architecture possible: The class hierarchy cannot be implemented with single inheritance without code duplication. This is because some of the corresponding classes in the trait-branch

and the class-branch of the hierarchy 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 traits are designed for, and hence it is natural to use traits to implement the new kernel.

The problem we face is starting a certain system without the system already functioning. To implement a reflective traits kernel which is modeled by itself, we need to do a *bootstrapping*. The bootstrapping of traits in Squeak mainly consists of the following two parts.

First, we need to extend the traditional kernel so that it is able to represent traits and trait composition (*i.e.*, classes and traits that are composed of traits) and we need to make sure that instances of classes composed of traits exhibit the new runtime behavior.

In the second stage of the bootstrapping process traits allow us to decompose the new kernel classes into separate traits that can, if necessary, be shared between the different branches of the hierarchy.

This decision follows the fundamental idea of reflective programming: to use the available features of a language to define and control the behavior of the language itself [KICZ 91]. In this sense, an implementation of traits in a reflective language like Smalltalk, should be itself built using traits.

1.2 Our Approach

Our approach follows these rationales:

Sophisticated design and stability. An important criterion when designing the new kernel was its *simplicity*. The newly implemented code aims at being short, understandable and documented – code already present from the traditional kernel was refactored when necessary.

What helped us in achieving a robust and stable implementation was *test-driven development* [ASTE 03]. All the critical code is covered by unit tests. This led us to first do "the simplest things that can possibly work" and then evolve step by step the additionally needed behavior while bootstrapping traits and integrating them in the development environment.

Like this, we aimed at achieving a sophisticated design without any unnecessary complexity.

Following the structure of the traditional kernel. For reasons of backward compatibility and understandability, we decided that the new language kernel should follow the structure of the traditional kernel [GOLD 83]. On one hand, this means that we wanted to preserve the purpose, relationships and responsibilities of the

existing kernel classes but it also means that the new classes that are necessary to represent traits should be structured in a similar way.

Avoiding changes to the Squeak virtual machine. To ease deployment, portability and maintenance we decided not to introduce any changes to the virtual machine.

Avoiding runtime overhead of traits. A program using traits should not be slower than the corresponding single inheritance program in which all methods provided by traits are implemented locally. There may be a small performance penalty because instance variables cannot be accessed directly from traits and local accessor methods have to be used instead. But we think that this is entirely justifiable because accessor methods are used wildly in any case since they improve maintainability.

Simple deployment. In contrast to the old prototype implementation that could not be loaded into a fresh Squeak image, our new implementation is packaged in a self-installable file. It can be loaded with one click and automatically bootstraps the new traits kernel. As discussed above, our implementation runs on the unmodified virtual machine.

To avoid changes to the virtual machine and to avoid runtime overhead of traits our implementation flattens the trait structure at composition time. This means that when a class is composed of traits, its method dictionary is changed to incorporate all the relevant trait methods (*i.e.,* the methods that are not overridden or excluded). Since compiled methods in traits do not usually depend on the location where they are used, the actual method objects (*i.e.,* the bytecode) can be shared between the trait that defines the method and all the classes and traits that use it. Thus, this process requires only a small fraction of the bytecode to be duplicated. Duplication of source code is never necessary.

Since single inheritance alone is not expressive enough to achieve an architecture that mimics the structure of the original kernel without code duplication, we decided to bootstrap our new kernel with traits.

The process of bootstrapping was not only followed during development, it is present in the self-installable package that provides the facilities to load traits into a traditional Squeak image. This supports a simple installation of the new kernel. What we could not avoid is the fact that each class in the system has to be recompiled during installation[1] because we modified the format of the kernel classes.

The result of our approach is a new reflective Smalltalk kernel that is built from traits.

[1]The recompilation takes 20 minutes on a machine with 1.25 GHz processor.

1.3 Contributions

The thesis presents the new architecture of the Smalltalk-80 kernel bootstrapped with traits. It discusses the rationales and considerations that led to the design of our implementation.

We reveal different *implementation strategies*: the static design approach which we choose for our implementation and the dynamic design approach which is based on a modified method lookup. The thesis also provides an overview of the traditional Smalltalk-80 kernel which is the starting point and context of our implementation.

We present the complete *reengineering* of the kernel with traits. By decomposing the kernel classes into traits we show how the new kernel can be expressed by traits themselves.

We develop a general *iterative process* to decompose a class into traits. It depends on basic heuristics derived from our analysis to identify potential problems caused by dependencies between traits and classes.

The thesis discusses how traits are implemented to provide the facility to fit Smalltalk's parallel class hierarchy paradigm (classes and metaclasses). We show how traits can be used to *compose metaclasses* in a uniform and safe way. Those traits, applied to metaclasses, form class properties. As an example we show how we refactored the Boolean hierarchy in Squeak using the class properties *Abstract* and *Singleton*.

Another part of our work is the integration of traits in the *development environment*. This includes development tools such as a new code browser or the adaptation of the debugger. Furthermore the different facilities and tools for source code management were enhanced to support traits. The integration was crucial for our work because at a point in the bootstrapping process we were forced to have a source code management supporting traits and allowing the team members to develop concurrently. This work, though, is out of scope of this thesis and is not further discussed.

1.4 Thesis Outline

- Chapter 2 briefly introduces the traits model and discusses different implementation strategies.

- In the beginning of Chapter 3 we analyze the important parts of the traditional Smalltalk kernel and then show the new traits kernel architecture. It follows a section explaining the composition of traits on which the flattening algorithm depends. The implementation of the flattening algorithm, the most important part of the kernel, is presented next. This chapter ends with a discussion about the usage of traits on the meta-level.

- Chapter 4 discusses the process we developed to refactor an existing system into traits. We introduce basic design heuristics which help identifying traits.
- Chapter 5 presents the approach of using traits to implement class properties.
- In Chapter 6 we present the new, bootstrapped trait kernel. The classes defining class and trait behavior are now decomposed into traits.

Chapter 2

The Traits Model

2.1 Overview

Traits are an extension of single inheritance with a purpose similar to that of mixins but avoiding their problems. Traits are essentially groups of methods that serve as building blocks for classes and are primitive units of code reuse. As such, they allow one to factor out common behavior and form an intermediate level of abstraction between single methods and complete classes. A trait consists of provided methods that implement its behavior, and of required methods that parameterize the provided behavior. Traits cannot specify any instance variables, and the methods provided by traits never directly access instance variables. Instead, required methods can be mapped to state when the trait is used by a class.

With traits, the behavior of a class is specified as the composition of traits and some glue methods that are implemented at the level of the class. These glue methods connect the traits together and can serve as accessor for the necessary state. The semantics of such a class is defined by the following three rules:

- *Class methods take precedence over trait methods.* This allows the glue methods defined in the class to override equally named methods provided by the traits.
- *Flattening property.* A non-overridden method in a trait has the same semantics as the same method implemented in the class.
- *Composition order is irrelevant.* All the traits have the same precedence, and hence conflicting trait methods must be explicitly disambiguated.

Because the composition order is irrelevant, a conflict arises if we combine two or more traits that provide identically named methods that do not originate from the same trait. Traits enforce explicit resolution of conflicts by implementing a glue method at the level of the class that overrides the conflicting methods, or by method exclusion, which allows one to exclude the conflicting method from all but one trait. In addition traits allow method aliasing. The programmer can introduce

an additional name for a method provided by a trait to obtain access to a method that would otherwise be unreachable, for example, because it has been overridden. Traits can be composed from sub-traits. The composition semantics is the same as explained above with the only difference being that the composite trait plays the role of the class.

Traits are proposed by Schärli et al. [SCHÄ 03a] and [SCHÄ 02]. Latter discusses the formal model. Furthermore Black et al. [BLAC 04] and [SCHÄ 03b] discuss methodologies and tools in relation to traits. Traits were also applied [BLAC 03] to refactor the Smalltalk collection hierarchy [COOK 92].

2.2 Conventions

In our implementation and throughout this thesis, trait names start with the letter T, and class names do not. Because the traits are implemented in Squeak, we present the code in Smalltalk. The notation ClassName≫methodName indicates that the method methodName is defined in the class ClassName.

The graphical representation of a trait is illustrated by Figure 2.1.

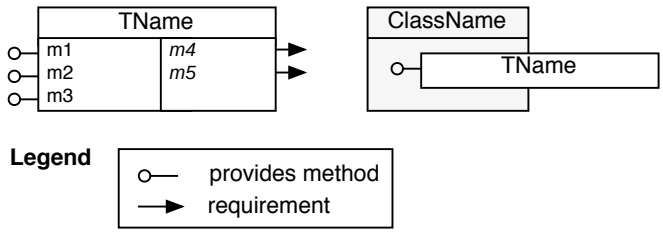

Figure 2.1: Left: trait with provided and required methods. Right: collapsed trait applied to a class.

A trait contains a set of methods that implement the behavior that it *provides*. These methods are shown in the left column of the trait. In general, a trait may require a set of methods that serve as parameters for the provided behavior. These so called *requirements* are shown in the right column. Traits cannot specify any state, and never access state directly. Trait methods can access state indirectly, using required methods that are ultimately satisfied by accessors (getter and setter methods) of a class.

The right side of Figure 2.1 shows TName in a collapsed representation applied to a class.

2.3 Analysis of Different Implementation Strategies

We now discuss two different approaches we considered to implement traits in Smalltalk.

2.3.1 The Dynamic Design Approach

The idea of the dynamic design approach is to adapt the traditional method lookup of the virtual machine to look up trait methods at runtime. Below we briefly outline the modified algorithm for a self-send of a method with selector s:

- As usual, the method lookup starts at the class of the receiving object. If the method is defined in this class (*i.e.,* s matches a key in the method dictionary), the method is returned (*class methods take precedence over trait methods*).
- If the method is not found in the class, the method lookup does not continue at the superclass as usual but rather follows the composition of traits (*trait methods take precedence over superclass methods*). An identity-set of *provided* methods of the composition is constructed as follows: for each trait t in the composition, collect the methods that satisfy one of the following two rules:
  - t *provides* a method with the name s and the composition clause does not declare an exclusion for this method
  - the composition clause declares an alias a→m with a = s and the aliased method m is *provided* by the trait t

  The trait t *provides* a method m if
  - m is defined locally in t, or if
  - the trait composition of *t* provides m

  If the above constructed set of provided methods includes more than one method, a conflict is signalled. If the set contains exactly one method it is returned and if the set is empty the lookup continues recursively at the superclass.
- Finally, if no method is found and there is no superclass (if the class is Object), the method doesNotUnderstand: is sent to the receiver with the representation of the initial message.

Note that trait composition is not necessarily a tree but can also be a graph without cycles like in multiple inheritance class hierarchies. This means that there may be several paths to the same trait. The method lookup algorithm might thus be optimized to not search the same path more than once.

Another difficulty is to handle the special semantics of *super-sends* defined in trait methods. Opposed to the above discussed method lookup, the method lookup of a message-send to super starts at the superclass *of the class of the method* containing the super expression (note that this is not necessarily the superclass of the receiver's class). For a method that is defined in a trait the *class of the method* is the

class the trait is (indirectly) applied to. For a trait method this class is not known at compile-time. Thus, the superclass of the class in which a trait method is executed has to be computed dynamically at runtime and cannot be statically stored when compiling as it is done with locally defined methods in classes.

In Smalltalk-80 a compiled method stores the superclass of the class in which it is defined in its literal frame at the last position if the method includes a super-send. When the compiler encounters a message to super, it uses the bytecode that pushes self for the receiver, but it uses an extended super bytecode to indicate the selector instead of a regular send bytecode [GOLD 83].

The literal frame of a method cannot be used to store the class in which the method is executed because the trait can be applied in different classes at the same time. Therefore the virtual machine and interpreter implementation has to be changed not only to look up methods in traits but also to dynamically compute the superclass of trait methods with super-sends.

2.3.2 Our Approach: The Static Design Approach

The idea of our approach is to make use of the fact that the trait composition does not affect the semantics of a class. The *flattening property* says that the meaning of a class would remain the same if the methods obtained from traits have been defined directly in the class [SCHÄ 03a].

By flattening the trait structure at composition-time we achieve the correct runtime behavior. This allows us to ignore the traits structure at runtime and use the ordinary method lookup algorithm that only takes the (flattened) method dictionaries of classes into account. This has the advantage that we do not have to introduce any changes to the virtual machine.

In Smalltalk, extending the method dictionary with methods from traits does not generally mean to duplicate the methods (which are first class objects in the Smalltalk system). In most cases – when the method does not use the keyword *super* – the identical bytecode can be used and thus the method dictionary just references the original method which is defined in the trait. Methods with super-sends store in their literal frame a reference to the superclass of the class in which they are defined. Therefore, methods defined in traits which send a message to super have to be copied and adjusted[1] to reference the actual superclass when they are applied to a class.

The method dictionaries not only have to be modified when a trait composition is applied but also when a method is (re-)compiled or removed from a trait which is (indirectly) used. In addition, each class has to keep track of the set of methods that are defined locally to appropriately update its method dictionary. For example, if a local method is deleted from a class the trait composition has to be queried for a method with the same name because this method, if it exists, replaces the local one. Aliasing, exclusions and conflicts are also handled by the flattening algorithm

[1] This is simply done by re-compiling the method in the class it is applied to.

and do not need any adaption to the runtime environment.

For optimization reasons each trait knows where it is used to facilitate the propagation of changes. This is the same idea as is used for classes: each class in the system keeps track of its subclasses, although this is technically redundant information. It avoids the very costly operation of iterating over all classes in the system to collect the subclasses of a class.

Flattening method dictionaries. The flattening algorithm affects the method dictionary of a class C that is composed from at least one trait, as follows:

- The method dictionary of C is extended with an entry for each provided trait method that (i) is not excluded, (ii) is not overridden in C, and (iii) 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.

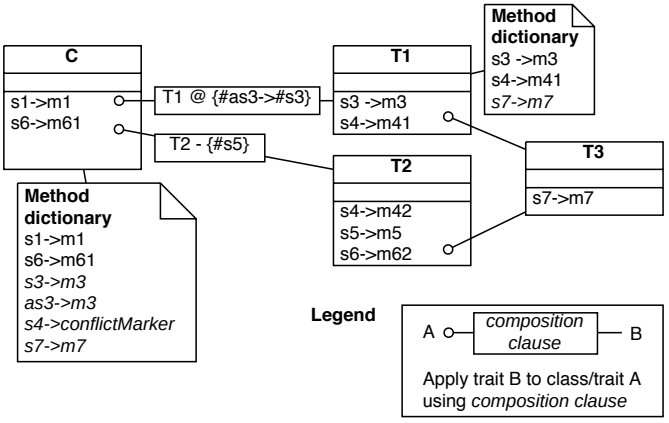

Figure 2.2: Static approach: flattened method dictionaries.

Figure 2.2 shows an example with a class named C that uses the two traits T1 and T2 by aliasing a method of the former and excluding one from the latter. Both traits use a sub-trait T3. Figure 2.2 shows the mapping from the selectors to the methods that are defined locally (*e.g.*, s3→m3 means that the selector named s3 points to the method m3 in the method dictionary).

Furthermore Figure 2.2 illustrates the flattened method dictionaries. The non-local methods are printed in italic. Flattening the method dictionaries has the following effect: T1 and T2 both get the method m7 with the name s7 from T3. Looking at the method dictionary of the class C shows the following different cases:

- s3→m3 is added without change because T2 does not provide a method named s3 and C does not define one locally.
- Since the composition clause of C defines the alias T1 @ {#as3→#s3}, an additional entry is added: the method m3 (T1≫s3) under the alias selector as3.
- T1 and T2 both provide a method with selector s4 and C does not define a s4 locally. Since the two methods m41 and m42 are not identical a conflict marker method[2] is generated and added under the name s4.
- Because of the exclusion T2 - {#s5} the method m5 from T2 is not added.
- T1 and T2 both provide a method named s7 which they obtain from the same trait T3. Thus this is the identical method (m7) and it is added without generating a conflict.

With this approach we can additionally make use of the fact that traits have the flattening property the same way as classes do. In many respects traits are very similar to classes: they are first-class objects having a name, holding and managing a method dictionary, compiling methods etc.. With this design, traits can use the identical flattening mechanism as that of classes for their sub-traits.

2.3.3 Discussion

The dynamic and static approaches are different in many aspects: the latter approach (which we have eventually chosen) solely takes action at composition- and compile-time by making use of the flattening property whereas the former approach modifies the runtime behavior of the language (*i.e.,* the virtual machine) to achieve the right semantics.

Comparing execution speed, the static approach does not have any effect because the method dictionary directly references the methods as it would if the methods had been defined in the class. The dynamic approach, however, probably has a negative effect on the performance because an extended method lookup is used. In addition the superclass has to be computed at runtime. Caching might be effective to regain performance.

The static approach additionally has the advantage that the virtual machine does not have to be modified to support traits – it is very straightforward to implement. The dynamic approach with its incompatibilities with the current architecture makes a change of the virtual machine necessary.

[2] A conflict marker method is a automatically created method with self traitConflict as its sole statement which throws an exception

The static approach duplicates method dictionary entries by mixing trait methods into method dictionaries of classes. But in most cases those entries are only references to the original methods. Thus, except for methods including super-sends (which is a relatively rare case), the identical bytecodes are used. The source code of a method always stays unchanged.

These observations led us to the decision to choose the static approach. The most important arguments are that our approach does not need a change of the virtual machine and that it is very straightforward to implement. The next chapter discusses the implementation of our approach in Squeak.

Chapter 3

The New Kernel: Design and Implementation

This chapter discusses the first step of bootstrapping the new kernel: the enhancement of the language to support traits. Chapter 4 shows how we refactored the intermediate result by using traits themselves and Chapter 6 presents the final implementation.

This chapter is related to the Smalltalk-80 kernel since the implementation is done in Squeak [INGA 97]. But the ideas and considerations are general enough that they are valuable in the context of other languages. Since Smalltalk is a dynamically typed language we do not deal with problems concerning static types that have to be considered when bringing traits to languages such as C# or Java [FISH 03].

3.1 Smalltalk-80 Kernel Architecture Overview

Because we integrate the traditional kernel into the new trait kernel, we present some important implementation details of the Smalltalk-80 system, especially the part which implements classes. In Squeak and other pure object-oriented languages, classes are first class objects, instances of other classes called metaclasses. Thus, the behavior of classes is described by classes the same way as other objects in the system. In this section we make a brief analysis of the responsibilities of those classes.

This eventually leads us – by comparing the responsibilities of classes with the responsibilities of traits (Section 3.1.4) – to the new class hierarchy of the trait kernel (Section 3.2).

3.1.1 Classes and Metaclasses

Classes. In class-based object oriented languages, classes play two roles: Creating new instances and defining the behavior of those instances. In Smalltalk as

well as for example in CLOS classes themselves are first-class objects. This reflects Smalltalk's guiding principles of "everything is an object" and "each object is an instance of one class". Consequently, each class is an instance of another class, its so-called *metaclass* [INGA 76] [COIN 87] [FORM 99]. There is exactly one metaclass for each class.

Metaclasses. In the same way as classes define behavior for their instances, metaclasses define behavior for classes. The main responsibility of metaclasses is the implementation of instance creation, inheritance, instance variables and method compilation. Specific metaclass-properties are for example *Singleton*, *Final* or *Abstract* [LEDO 96] [DUCA 05]. Furthermore metaclasses provide the possibility to define user specific behavior such as custom instance creation and initialization of newly created objects or the implementation of behavior and state that are in common for all instances of a specific class.

Metaclasses in Smalltalk are *implicit* which means that the programmer cannot specify the metaclass for a class manually [GOLD 89] and that the metaclasses cannot be shared. Smalltalk automatically generates a metaclass which is anonymous (it does not have an own name and is not referenced directly by the environment) when a new class is generated. It can be accessed by sending the message class to a class.

3.1.2 Parallel Class Hierarchy

The class hierarchy in Smalltalk-80 follows the rules:

- Metaclasses: the class of the class A is the metaclass named A class. Each metaclass has exactly one instance.
- Parallel hierarchy: if the class B is a subclass of A then the metaclass of B, B class, is the subclass of the metaclass A class. This ensures *upward* and *downward compatibility* [BOUR 98] between the classes and metaclasses.

Figure 3.1 illustrates the inheritance and instantiation relationships of the core classes of the Smalltalk-80 language. At the top of the inheritance hierarchy stands Object which is the ultimate superclass of all other classes in the system. The kernel classes are Behavior, ClassDescription, Class and Metaclass. Figure 3.1 illustrates the corresponding metaclasses and their inheritance relationship.

There is one exception to the rules stated above, namely the singularity at Object[1]: the class hierarchy terminates at Object (Object's superclass is nil), but the metaclass hierarchy wraps around to Class, since all metaclasses are (indirect) subclasses of Class. This means that the superclass of the metaclass of Object is Class.

[1] In Squeak there exists a superclass of Object named ProtoObject. In this case the singularity is at ProtoObject. But for our observations this does not matter.

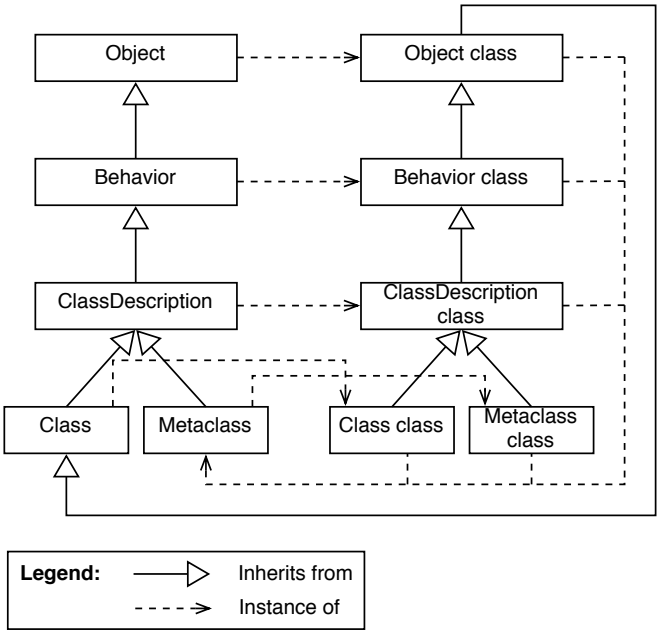

Figure 3.1: The parallel class hierarchy of Smalltalk.

This layout of the inheritance and instantiation relationships has the following consequences: the classes which implement classes, such as Class and Metaclass, ultimately inherit from Object. The class of Class, for example, ultimately inherits from Object class, the metaclass of Object. And this class inherits from Class. Hence the class Class defines its own behavior.

3.1.3 Identifying Responsibilities

To start with the new implementation we first analyzed the responsibilities of the core kernel classes because we intended to embed the original kernel into the new one to maintain backward compatibility on one hand and to share behavior that is in common for classes and traits on the other hand. The class hierarchy is illustrated in Figure 3.1. In contrast to Goldberg et al. [GOLD 89] which list the protocol divided into the categories *creating*, *accessing*, *testing* and *enumerating*, we identify groups of methods by their responsibilities.

Behavior. Behavior provides the minimum structure necessary for classes but neither provides a representation for instance variable names and class variable names, nor for a class name or a comment about the class. It's responsibilities and the corresponding methods are:

- Creating and allocating instances: new, basicNew etc.

- Managing the method dictionary (a dictionary object that stores compiled methods): addSelector:withMethod:, compiledMethodAt:, selectors, allSelectors, canUnderstand: etc.
- Compiling methods: compile: etc.
- Maintaining the class hierarchy (link to the superclass): superclass, subclasses etc.
- Managing and storing information concerning (i) the format of the class (encodes the kind and number of instance variables as an integer): format, (ii) instances of the class: allInstances and (iii) instance variables: instVarNames (only faked names because Behavior does not have named instance variables) and classVarNames

ClassDescription. ClassDescription provides additional facilities to its superclass Behavior. It is an abstract class intended to define the common behavior of Class and Metaclass and provides:

- Named instance variables: addInstVarName:, removeInstVarName:
- Category organization for methods: compile:classified:
- The notion of a name of this class (implemented as subclass responsibility)
- The maintenance of a change set, and logging changes on a file
- Most of the mechanism for filing code in and out: fileOutOn: etc.
- Class comment

Metaclass. The primary role of a metaclass in the Smalltalk-80 system is to provide the interface for initializing class variables and for creating initialized instances of the metaclass' sole instance. Metaclass delegates much of the functionality to ClassBuilder. Its responsibilities are:

- Creating and managing its sole instance
- Declaring and initializing class variables of its sole instance: instanceVariableNames:
- Initializing subclasses of itself, that is, a method for creating a metaclass for a new class

Class. Class describes the representation and behavior of objects. It adds comprehensive programming support facilities to the basic attributes of Behavior and the descriptive facilities of ClassDescription. Its responsibilities are:

- Naming of classes (by which they can be accessed in the environment) and renaming: name, rename:
- Creating subclasses: subclass:instanceVariableNames: etc.
- Providing a set of all subclasses
- Maintaining a classPool for class variables shared between this class and its metaclass and a list of sharedPools
- Maintaining a collection of subclasses which is a redundant structure. It is never used during execution, but is used by the development environment to simplify or speed certain operations.

3.1.4 Identifying Shared Behavior Between Classes and Traits

Traits in some respects are very similar to classes because they define behavior (*i.e.,* they manage a method dictionary). On the other side, classes are instance creators and traits are not. A trait, an instance of the new class Trait, is not a class. In our new system, classes and traits have additional common behavior: both can apply a trait composition and have to manage it.

Hence, an important observation is that classes and traits have common behavior and should be designed to optimally share code and implement a common interface. Making traits and classes having a similar interface greatly improves the ease of integration with tools of the development environment like the code browser.

Shared behavior. In the previous section we have identified the responsibilities of the classes Behavior, ClassDescription, Class and Metaclass. *Which functionality do classes implementing traits have in common with the classes of the original kernel?*

- Behavior: Since a trait is essentially a group of methods it also implements behavior for accessing and managing a *method dictionary* as well as compiling methods.
- ClassDescription: As with classes, traits should also be able to categorize their methods. Another functionality implemented on ClassDescription is the code exchange mechanism and the class comment.
- Class/Metaclass: The system dictionary which references all the classes in the system is also used for traits. Traits are added to the system dictionary

by their names the same way that classes are. Furthermore we implement traits so that they fit the paradigm of the class - metaclass pairs. Section 3.6 discusses in more detail how traits are implemented and used at the meta-level.

Not in common behavior.

- Behavior: At the level of Behavior we find the implementation for instance creation, handling the inheritance hierarchy, instance variables and format of a class which are not used for traits.
- ClassDescription: Named instance variables, since traits do not define state.
- Class/Metaclass: Creation and management of subclasses and class variables shared between the class and metaclass.

Class behavior	**Used by traits**
Behavior	
instance creation	no
method dictionary	yes
compiling	yes
class hierarchy	no
instances, variables, format	no
ClassDescription	
named instance variables	no
category organization for methods	yes
name	yes
change set, logging changes to file	yes
file-out mechanism	yes
comment	yes
Class	
subclass creation	no
pool and class variables	no

Table 3.1: Responsibilities of classes that traits have in common

The above analysis reveals that there is behavior that classes and traits have in common and not in common at all levels (see Table 3.1.4 for a brief summary). The top part of the table corresponds to the functionality defined at the level of Behavior, the middle part to the level of ClassDescription and the bottom part to the level of Class.

Our current possibilities with class-based single inheritance are limited to a suboptimal solution: Making the class Trait a subclass of Behavior or ClassDescription introduces the problem that our traits would have implemented too much

behavior, *e.g.*, instance creation or instance variables. We would be forced to disable those methods on the class Traits. The other possibility is to implement Trait at a higher level in the hierarchy and do without sharing all common behavior. The drawback of this solution is that we have to duplicate code, especially at the level of ClassDescription.

Looking at this architectural problem, we can clearly see that traits would be very useful here. Having traits we would be able to implement the identified groups of behavior with traits and then use them where needed at the right place in the inheritance hierarchy. This is the plan we follow. In the next section we present the class hierarchy of the new kernel which derives from the analysis we discussed above.

3.2 New Trait Kernel Class Hierarchy

The responsibilities of the new 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. This section gives an overview of the responsibilities and structure of the new kernel classes, while their decomposition into traits is presented in the next chapter. Figure 3.2 illustrates the inheritance hierarchy. The newly added classes are printed in bold.

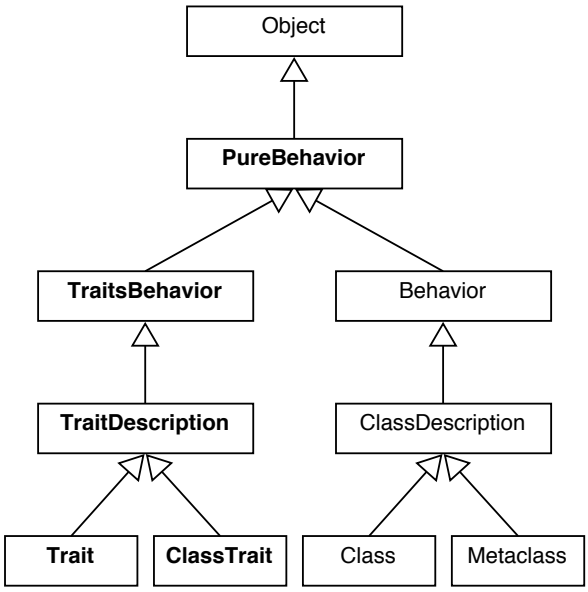

Figure 3.2: The new trait kernel class hierarchy.

PureBehavior. The class PureBehavior is the abstract root class of the new kernel hierarchy, and it describes the common facilities to represent behavior (but not state) in classes and traits. It provides the basic functionality for compiling methods, managing a method dictionary and trait composition. Note that the facilities for inheritance are only provided by the class Behavior as inheritance is only used for classes and not for traits.

To fulfill these responsibilities, the class PureBehavior uses three instance variables to store the method dictionary, the composition clause, and the local selectors (*i.e.*, the selectors in the method dictionary that are actually implemented locally rather than being obtained from sub-traits as a result of the flattening). The names of those instance variables are: methodDict, traitComposition and localSelectors. However, because the order of class instance variables is hard-coded in the virtual machine, and the instance variable holding the superclass must be first, the declaration of instance variables is deferred to the subclasses and they are accessed through abstract accessors from within PureBehavior.

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

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 the trait is used. This means that it declares an instance variable, named users, to hold all the classes and traits that are composed of the current trait. Like Behavior it also declares the three instance variables methodDict, traitComposition and localSelectors deferred by the superclass PureBehavior.

ClassDescription. As in the traditional Squeak kernel, ClassDescription adds a number of facilities to Behavior: named instance variables, a category organization for methods, the notion of a class name, logging of changes, and a mechanism for filing-out the class. As in the traditional Squeak kernel, this is done by declaring two instance variables for holding an array of instance variable names and the category organization of the methods, respectively. Note that the class ClassDescription defers the declaration of the instance variable which holds the name of the class to its subclass Class.

TraitDescription. The purpose and most of the implementation of the class TraitDescription is nearly identical to ClassDescription, with the main difference being

that TraitDescription is used for traits and therefore does not provide the facilities for named instance variables. This means that this class adds categorization for methods, the notion of a trait name, logging of changes and a file-out mechanism to TraitBehavior.

Class. The purpose of the class Class is the same as in the traditional kernel; *i.e.,* the instances of Class describe the representation and behavior of objects. Also the responsibilities and most of the implementation of the class Class are the same as before. In particular, the class Class concretizes its superclass by providing instance variables and accessors for the class name, the set of subclasses, the environment where the class is stored, and pool variables.

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 for the name and the environment. Since traits do not define variables, the class Trait does not provide facilities for pool variables. However, Trait declares an instance variable to hold the associated classtrait, which is an instance of the class ClassTrait described below.

Metaclass. Since the metaclass hierarchy is generally parallel to the class hierarchy and metaclasses are anonymous, the class Metaclass does not provide instance variables for name, environment and pools. Instead, this functionality is derived from the associated class. Thus, the class Metaclass declares only a single instance variable to hold the base class (*i.e.,* the Singleton instance) that is associated to the metaclass.

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

The distinction between traits and classtraits is necessary because a classtrait has slightly less and different behavior compared to a trait used at the instance side: for example because a classtrait is anonymous it does not need to store a name. A classtrait derives its name from the accompanying instance side trait the same way that metaclasses do.

Despite the conceptual similarity between metaclasses and classtraits, there are two crucial differences. The first difference is that each class is a Singleton instance of its associated metaclass, but each trait is an instance of the class Trait rather than an instance of its associated classtrait. This difference is also the reason why we chose the name ClassTrait rather than MetaTrait: a classtrait 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 a classtrait. If the methods provided by a trait do not rely on any methods on the class side, there is no need for this trait to have an associated classtrait.

Section 3.6 discusses in more detail how traits are supported at the meta-level and related to that, Chapter 5 discusses how traits are used to implement metaclass properties.

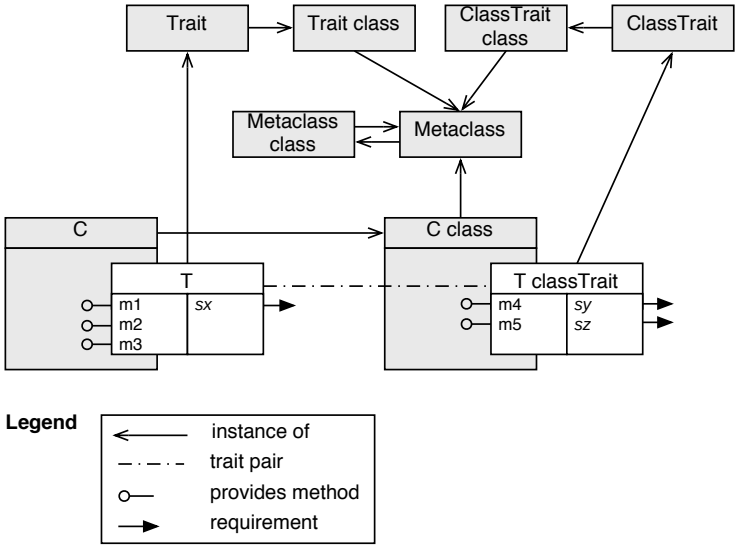

Figure 3.3: Instantiation relationships between classes, metaclasses and traits.

Figure 3.3 illustrates the instantiation relationships between a class C, a trait T and the classes Trait and ClassTrait. Additionally, Figure 3.3 shows all the metaclasses. The class C is an instance of its metaclass C class and the class of C class is Metaclass.

The classes Trait and ClassTrait work the same way. The trait T is an instance of the class Trait and its corresponding trait pair T classTrait is an instance of the class ClassTrait. Note that T is not an instance of T classTrait. Details concerning trait pairs, which facilitate the use of traits at the base and the meta-level in a similar way to classes and metaclasses, are discussed in Section 3.6.

The next sections discuss how traits are composed (Section 3.3 and Section 3.4) and how the flattening algorithm is implemented to achieve the right semantics of traits (Section 3.5).

3.3 The Trait Composition Clause

How traits are applied to a class or to another trait is expressed by means of a trait composition clause. It specifies which traits a trait composition contains and additionally which aliases should be created and which methods should be excluded. This is done in the definition of a class or trait. The following code shows the definition of a class named C:

```
Object subclass: #C
   uses: TA + TB @ {#a:→#x:} - {#x} + TC
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'traits'
```

The original class definition template is extended with the keyword uses:. As argument it expects the trait composition clause (printed in bold in the above code). By default this composition clause is empty which is expressed by an empty collection as the code below illustrates. The trait composition clause for trait definitions works in exactly the same way as for class definitions:

```
Trait named: #TA
   uses: {}
   category: 'traits'
```

In the same way as class and trait definitions are specified in Smalltalk, the trait composition clause is specified by means of normal Smalltalk code. This uniform approach makes a special parser for trait composition clauses needless. As we discuss in the subsequent section, the operators + for sums, @ for aliases and - for exclusions build up a *trait composition*, an instance of the class TraitComposition. Behaviors store the trait composition in their instance variable named traitComposition as discussed above in Section 3.2.

Sum. To combine traits, the operator + is used between each trait as shown in the class definition above. In the composition clause the operators @ and - bind stronger than the operator +. This means, for example in the clause TA + TB @ {#a:→#x:}, that the alias is only applied to the trait TB and not to the sum of the traits TA and TB. Trait composition is symmetric: the order in which the traits appear in the composition clause is irrelevant.

Aliases. Aliasing is achieved by the operator @ and an array[2] of associations. An association #a:→#x: introduces a method with selector a: that aliases the existing method named x:. Obviously, both method names need to have the same number of arguments. Our implementation allows one to mix binary selectors and keyword selectors with one argument, *e.g.*, the selector @ can be aliased by at:.

[2]The curly braces are special in Squeak. They dynamically create an array of elements separated by points.

Exclusions. Exclusions are specified by the operator - with argument an array of selectors as shown in the above example. Each method provided by the trait whose name matches one of the selectors in the array is excluded from the composition.

Aliases and exclusions are often used to resolve a conflict: In this case a common pattern is to first create an alias to the conflicting method and then exclude it. Like that the conflict is resolved and the method is still accessible under the new alias selector. If we first have excluded the method it would not be possible to add an alias to it afterwards. Because of this and for conformity reasons we enforce that aliases must be specified before the exclusions.

There is mainly one restriction that has to be checked when applying a trait composition: the directed graph which is built by traits must not have cycles. Each trait is represented by a node. If a trait TA uses a trait TB then there is a vertex from TB to TA. For example, it is forbidden that TA uses TB if TB directly or indirectly uses TA.

Note that the graph does not necessarily have to be a tree. E.g., it is not forbidden to create a sum of two traits which both use the identical sub-trait. When flattening the trait structure, the algorithm has to identify identical methods which are propagated through different paths. This case is not a conflict in contrast to non-identical methods with identical names.

3.4 Implementation of Trait Compositions

In the previous section we discuss the possibilities and restrictions to specify how a set of traits is applied to a behavior. In this section we show how trait compositions, which are built up from the composition clause by means of normal message sends, are implemented. This way of composing a trait composition is elegant and powerful. It follows the spirit of Smalltalk which uses message sends, *e.g.,* to define classes.

Figure 3.4 illustrates the class diagram of the classes representing a *trait composition*.

Trait composition. A trait composition is an instance of the class TraitComposition. It holds an ordered collection of *transformations*. For each trait in the composition clause there exists exactly one transformation in the collection. Each transformation can itself contain another transformation as discussed below. The order in which the traits are added to the collection by the sum operator is semantically not relevant but has to be maintained because it is needed when a trait composition clause is evaluated: the operators for aliasing and exclusions always only effect the last trait in the composition clause.

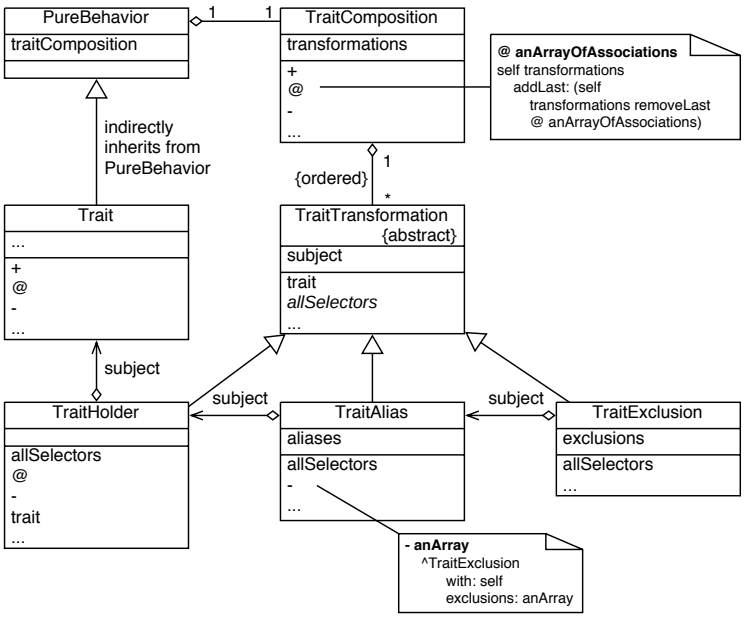

Figure 3.4: Class hierarchy implementing trait compositions.

Trait transformation. A trait transformation is an instance of the class TraitHolder, TraitAlias or TraitExclusion. Those three classes are subclasses of the abstract class TraitTransformation. The latter two represent a transformation of a trait, specified by the alias and exclusion operators. A trait holder is the identity transformation which does not modify the trait. The instance variable subject defined in TraitTransformation holds the object that is transformed. Thus, a trait holder always only holds a trait as its subject, an alias transformation always holds a trait holder, and a trait exclusion either holds a trait alias or a trait holder (latter reference is omitted from Figure 3.4). Each of the three concrete transformation classes implement the method allSelectors according to the transformation it represents.

Remembering the trait composition. Traits and classes which have applied a trait composition, store it in their instance variable named traitComposition. On one hand it is used to update the behavior's method dictionary with its provided and conflicting methods and on the other hand it is used to recreate the composition clause when displaying the class or trait definition.

Figure 3.4 shows the instance variable traitComposition being defined in the class PureBehavior, the root of the class and trait kernel. Note that as discussed in Section 3.2 the actual definition of the instance variable in our implementation is deferred to the subclasses of PureBehavior due to restrictions caused by the virtual machine. But conceptually its declaration belongs to PureBehavior.

Example. Let's have a closer look at how the trait composition is built by evaluating the following composition clause:

```
TA + TB @ {#a:→#x:} - {#x}
```

The messages +, @, and - are all binary selectors. They have the same priority and hence are sent one after the other from left to right. Note that messages have the same priority when evaluated in Smalltalk whereas we implement the composition clause so that aliases and exclusions have a higher priority than the sum operator: they always only transform the last trait in a sum of traits.

Sending + to the trait TA with the argument TB creates a trait composition (using double dispatch to avoid nested compositions if parenthesis are used). This composition holds two instances of TraitHolder which reference the two traits via subject. Next, the message @ is sent to this composition. Because the aliasing should only effect the trait TB the trait holder of TB is replaced by an alias transformation (Figure 3.4 shows the implementation of TraitComposition≫@). This alias transformation now holds as its subject the trait holder. Additionally it adds to the collection of aliases, stored in its instance variable aliases, the association #a:→#x:. In the same way as aliasing works the exclusion of x.

Restricting the order of aliasing and exclusions. Implementing the appropriate methods for aliasing and exclusion in the subclasses of TraitTransformation makes it possible to control the order in which they can be applied: TraitExclusion, for example, does not define aliasing and exclusion and TraitAlias only implements exclusion as Figure 3.4 illustrates. This enforces that aliases are specified before exclusions.

Trait composition used by the flattening algorithm. The trait composition not only holds the traits, aliases and exclusions but it also provides crucial behavior used by the flattening algorithm (it is discussed in the subsequent section): TraitComposition implements the API which *e.g.,* facilities to query the set of provided and conflicting methods or the comparison of two trait compositions. Hence, the logic which is not purely concerning the process of flattening the method dictionaries is defined by the classes representing the trait composition.

3.5 Flattening Algorithm

In Section 3.2 we discuss how the new class hierarchy of the kernel is organized and what the responsibilities of those classes are. We show how trait compositions are implemented in the previous section. In the following part the flattening algorithm, a crucial part of the kernel, is described. Its responsibility is to update the method dictionary of a behavior (a class or trait) whenever a relevant change occurs, *e.g.,* when a new trait composition clause is set. An overview, pros and cons of our approach compared to an alternative approach is discussed in Section 2.3.

Two important criteria we had in mind when writing this part of the new trait kernel was *robustness* and *simplicity*. After some iterations, our solution became elegant because on one hand we managed to unify the processing of the two different kinds of updates that have to be handled, and on the other hand, we factored out the logic that is responsible for computing the actual provided methods. Hence, this critical part of the kernel can be implemented in only a few lines of code which is, with only a few exceptions, presented in the remaining part of this section.

3.5.1 Overview

There are two cases where the flattening algorithm is used: (i) when a trait composition is applied to a class or trait (*i.e.*, when the class definition is compiled and evaluated) and (ii) when a method is compiled or removed from a trait. The former case can effect many methods, for example if a new trait is added, whereas the latter case most of the time only effects one single method (if aliases are involved there can be more than one).

The strategy. To achieve the semantics of traits the method dictionaries are *flattened*. This means that for each provided method of the trait composition we put an entry into the local method dictionary referencing this method. Generally, those compiled methods do not have to be copied; only if they include *super-sends* does the method have to be recompiled[3] to correctly reference the superclass of the class it is in. In contrast to bytecode duplication in the infrequent latter case, source code never has to be copied. To distinguish the local from non-local methods in the method dictionary a set containing all the non-local selectors is added to behaviors.

This approach ensures that nothing needs to be changed in the virtual machine and that there is no loss in runtime performance. Section 2.3.2 discusses in detail our approach.

The process of updating all the involved behaviors for a specific change is done in two steps which are repeated for each involved behavior: flattening the method dictionary and then, if the behavior is a trait, propagating the changes to all users of the trait. These two steps are implemented by the following method which is the entry point of the algorithm:

```
PureBehavior≫noteChangedSelectors: aCollection
  | affectedSelectors |
  affectedSelectors := IdentitySet new.
  aCollection do: [:selector |
    affectedSelectors addAll: (self updateMethodDictionaryFor: selector)].
  self notifyUsersOfChangedSelectors: affectedSelectors
```

[3] Recompiling the method in the class has the same effect as copying the method and accordingly changing the superclass reference in its literal frame.

Propagation of changes. During flattening the method dictionary (which is discussed below in Section 3.5.3), we track the selectors of all methods that are added or removed. This information is passed forward when notifying the users so that they only have to check a limited number of methods.

The method notifyUsersOfChangedSelectors: is a hook method (Template Method Pattern). The traits branch of the kernel class hierarchy specializes this method in the following way: It iterates over the set of users and sends to each of them the message noteChangedSelectors: with the argument affectedSelectors.

The updating is triggered whenever a method is:

- added to a trait,
- removed from a trait or a class

When a method is compiled in a class it simply replaces the old, maybe non-local, method if one is present. This is the desired behavior because local methods take precedence over trait methods. The event of removing a method has to be handled, though, because the trait composition may provide a method with this selector. In this case the local method is replaced with the non-local method.

3.5.2 Applying a Trait Composition

One of the two kinds of updates is the application or change of a trait composition clause. After the old trait composition is replaced with the new one, the following method is eventually invoked:

```
PureBehavior≫applyChangesOfNewTraitCompositionReplacing: oldComposition
  | changedSelectors |
  changedSelectors := self traitComposition
    changedSelectorsComparedTo: oldComposition.
  changedSelectors ifNotEmpty: [
    self noteChangedSelectors: changedSelectors].
  ...
  ↑ changedSelectors
```

The first statement queries the new trait composition for the *delta* between the old composition. Only if the returned set of selectors is not empty further steps are processed (see above for the implementation of noteChangedSelectors:).

Applying a trait composition with possibly a lot of methods is not more complex than what has to be done when only one method is changed. The above printed method unifies those two different kinds of possible updates. Most logic, *e.g.*, the computation of the *delta* between two compositions, is implemented in the classes representing a trait composition.

3.5.3 Updating the Method Dictionary

We now analyze the main part of the flattening process. The method updateMethodDictionaryFor: aSymbol, which is shown below, updates the method dictionary for a potential change of the method with the selector aSymbol. It is invoked by the method noteChangedSelectors: which we discussed above.

```
PureBehavior≫updateMethodDictionaryFor: aSymbol
  | effectiveMethod modifiedSelectors descriptions selector |
  modifiedSelectors := IdentitySet new.
  descriptions := self traitComposition methodDescriptionsForSelector: aSymbol.
  descriptions do: [:methodDescription |
    selector := methodDescription selector.
    (self includesLocalSelector: selector) ifFalse: [
      methodDescription isEmpty
        ifTrue: [
          self removeTraitSelector: selector.
          modifiedSelectors add: selector]
        ifFalse: [
          effectiveMethod := methodDescription effectiveMethod.
          (self compiledMethodAt: selector ifAbsent: [nil]) ∼∼ effectiveMethod
            ifTrue: [
              self addTraitSelector: selector withMethod: effectiveMethod.
              modifiedSelectors add: selector]]]].
  ↑ modifiedSelectors
```

In the second statement of the method, the trait composition is queried for a set of method descriptions. A method description provides meta information about provided and conflicting methods and is discussed below in Section 3.5.4. The set contains a method description object for the selector passed as argument plus one for each alias in the trait composition that points to the selector. Aliases have to be taken into account because if an aliased method was modified or deleted, also any alias is effected by this change.

The method now adds or removes trait methods depending on the method descriptions. For each method description the following situations can occur:

- The method description is empty. This means that there does not exist a provided method with this selector. In this case the method is removed from the method dictionary if it exists.
- The method description is not empty and there exists a local method with the same name as the one of the method description. In this case there are no changes needed because local methods take precedence over trait methods.
- The method description is not empty and there does not exist a local method with the same name. The effectively provided method is added to the method dictionary if it is not already there. The effective method is the actual method provided by a trait or is a conflict marker method.

The two methods addTraitSelector:withMethod: and removeTraitSelector: simply add respectively remove the method from the method dictionary. When adding a method to a class it is additionally checked if the method includes a *super-send*. In this case it is recompiled so that the entry in the literal frame of the compiled method correctly references the superclass. This behavior is added by Behavior which specializes the method. It is not directly implemented by PureBehavior because it is not necessary for traits since they do not have instances and thus their methods are never executed.

3.5.4 Meta Information for a Composition of Traits

To keep the flattening algorithm as simple as possible the logic concerning the calculation of provided or conflicting methods is delegated to the trait composition, the object representing the current composition clause. The class TraitComposition provides the relevant API for the flattening algorithm.

The previous sections show the use of two of those methods implemented by TraitComposition: changedSelectorsComparedTo: aTraitComposition in Section 3.5.2 and methodDescriptionsForSelector: aSymbol in Section 3.5.3. The former method returns the set of names of methods that have changed compared to the argument aTraitComposition.

The latter method is the most important method of this interface. It provides information about all methods in the trait composition which are relevant when the method with selector aSymbol has changed. This meta information is used by the flattening algorithm to decide which actions to take: E.g., when a method from a trait was recompiled the flattening algorithm of each user of the trait has to update the method dictionary. If there is an alias to this method, it has to be updated as well.

The method changedSelectorsComparedTo: provides the relevant information by returning a set of *method descriptions*, instances of the class TraitMethodDescription. Each method description encapsulates a collection of methods for one particular selector.

According to the number and kind of those methods a provided method exists, there is a conflict or there are no provided nor conflicting methods at all. TraitMethodDescription provides the interface to query for those situations, *e.g.*, effectiveMethod returns the provided method or the conflict marker method. How conflicts are handled is discussed in the following section.

3.5.5 Conflict Handling

Conflicts can occur if two or more traits in a composition, taking the operators of the composition clause into account, provide methods with the same name. If the methods are identical this is not a conflict. Identical methods are provided by traits if they directly or indirectly use the same trait as a sub-trait. Another case where

there is no conflict is when all but one method are required methods[4].

When a conflict occurs a *conflict marker method* is created and returned instead of the provided method. A conflict marker method has self traitConflict as its sole statement. Its name is the same as the conflicting method's name. If the parameters of the original method names are not identical, a generic name is chosen.

Example. The traits TA and TB implement two conflicting methods with the following signatures:

TA≫from: start to: end do: aBlock

TB≫from: start to: stop do: aBlock

Since the first and third argument name, *start* and *aBlock*, are identical, they are used in the conflict method. On the other hand the second argument names do not match and thus a generic name is utilized:

```
from: start to: anObject do: aBlock
    self traitConflict
```

Instantiating a class which has conflicts is possible. Only calls to a conflict method will eventually cause an exception to be thrown. This supports an iterative way of programming as it is often practiced in Smalltalk. Running code that is not yet fully implemented is also supported with conflicting methods in traits: the programmer can, when a trait conflict occurs, directly implement a new local method in the debugger. The method resolves the conflict because it is defined locally and takes precedence over the conflicting methods. The programmer can implement this method in the debugger and directly proceed the execution of the new code.

Apart from implementing a local method that overrides the conflict method, conflicts are often resolved by choosing one of the conflicting implementations and excluding the other one from the trait composition clause. If both implementations are required, an alias can be used to make the conflicting method accessible after the exclusion.

3.6 Traits at the Meta-level

An important concept of Smalltalk-80 is its parallel class hierarchy as discussed in Section 3.1.2. Each class has its own metaclass which defines instance creation, reflection etc. and which the programmer can use to define behavior such as custom instantiation methods. The Smalltalk development environment reflects this facility by providing views on the *instance side* and on the *class side* of a class. The instance side shows the protocol which is defined in the method dictionary of the class (*i.e.*, messages which objects of the class can understand), the class

[4] A required method is a method with the sole statement self requirement and is the traits analogy to self subclassResponsibility

side shows the methods that are understood by the class, *e.g.,* Number class≫@ which is used to instantiate a point initialized by two numbers, the receiver and the argument of the binary message @.

Apart from Ducasse et al. [DUCA 05] who propose to use traits to implement class properties to resolve metaclass composition problems (see Chapter 5), there has not been a discussion on how exactly traits can be applied and used at the meta-level. We think that it is important to extend traits so that they fit the paradigm of the parallel hierarchy of class - metaclass pairs.

3.6.1 Two Different Kinds of Usages

We identify two different possibilities how traits can be used at the meta-level:

Trait pair. Similar to classes a trait can have an accompanying class side trait that is applied to the class' metaclass. This accompanying trait should be generated automatically (like the metaclass of a class) and the tools should allow the developer to define methods in the code browser on the class side the same way as for classes. In addition, declaring aliases and exclusions also has to be possible for the class side trait. Section 3.2 already illustrated a trait pair applied to a class and its metaclass (see Figure 3.3).

A simple example illustrating a trait pair is the following:

```
T≫foo
  self class bar

T classTrait≫bar
  ↑ 17
```

The method foo in the instance side trait T sends bar to the class side. To be able to implement the method bar with the method foo the accompanying class side trait T classTrait is used.

Single trait applied directly to a metaclass. To implement class-properties such as Abstract or Final, often one does not need a trait to be applied to the instance side of a class. Although it is possible to implement those metaclass-properties or other class-specific behavior with the above discussed trait pair, it is more convenient and understandable to directly apply the trait to the class side.

3.6.2 Trait Pair Implementation

Although traits have many properties in common with classes, the trait pair and the class-metaclass pair are different in the sense that a trait is not an instance of its accompanying classtrait whereas the class is the (sole) instance of its metaclass. This is why the class side trait is not called metatrait but *classtrait*.

The classtrait is not the *class* of the base trait because it would cause the following problem: the method dictionaries of traits are used to store the methods they provide. When a classtrait is the class of the base trait then the provided methods in its method dictionary also define the behavior of the base trait. For example implementing the trait TSingleton would unexpectedly affect the behavior of the base trait: it would be possible to create singleton instances of the base trait.

To accomplish the first kind of usage of class side traits, traits declare an instance variable classTrait (and baseTrait resp.) and implement the navigation protocol to move between the class and the instance side. This navigation protocol (*i.e.*, the methods instanceSide, classSide, isInstanceSide etc.) is also understood by classes. ClassDescription≫instanceSide, for example, returns self theNonMetaClass.

3.6.3 Special Composition Clause Rules

Class side compositions are defined the same way as instance side compositions. This is important for two reasons: first, to be able to add additional traits directly to the class side and second to be able to manipulate the composition clause by alias and exclusion operators.

When using trait pairs to form a trait composition, constraints have to be introduced to guarantee consistency.

First of all, if a trait is added to the instance side and the trait has an accompanying classtrait, the class side composition has to include its classtrait as well. This happens automatically when applying a trait on the instance side and it is not possible to remove the class side trait from the composition clause. But, of course, it is possible to specify aliases and exclusions.

Second, a trait can only be directly applied to the class side if this trait does not provide any methods on the class side. Consequently, as long as a trait is applied to the class side of a behavior, the system prevents the developer from defining methods on the trait's class side.

Chapter 4

Identifying Traits

This chapter discusses a general *process* and a set of *heuristics* which stem from the experiences we obtained while applying our approach to the reengineering of the kernel classes. The heuristics are intended to help someone analyzing the structure of a decomposition of a class into traits. For illustration purposes we discuss the refactoring of the class PureBehavior.

In this sense this discussion can be of a more general interest because it shows how to identify weaknesses of a decomposition and it proposes possible improvements.

This chapter focuses on refactoring a class using traits. Another interesting question is how traits in class hierarchies can be identified. Black et al. report on the refactoring of the Smalltalk Collection hierarchy using traits [BLAC 03]. The refactoring of classes and *class hierarchies* has been widely discussed, *e.g.,* to refactor big libraries by reorganizing classes [CASA 95]. Traits are interesting in this context because they provide a fine-grained mechanism to improve reusability and extensibility of object-oriented systems.

4.1 Overview

We now present an incremental process to identify traits which proved to be successful when bootstrapping the new trait kernel. Our discussions are illustrated on the example of decomposing PureBehavior, the root class of our new hierarchy. We discuss the key aspects, problems and solutions to decompose a class into traits.

Problem. When reengineering an existing system to use traits we face the problem of identifying good traits in classes. Blindly moving similar methods into traits might cause useless and problematic structures because of unintended dependencies. Those dependencies are requirements that are satisfied in a suboptimal manner: often required methods in one trait are fulfilled by another trait rather than by the user of the trait. As we discuss below there are several possible solutions that can cure this problem.

Process. To identify traits we propose a simple process. The starting point is a class. The first step is to create an initial, probably suboptimal, solution by grouping methods to traits according to their responsibility (Section 4.2). We look closer at the intermediate solution by identifying dependencies. (Section 4.3). This analysis helps us to identify problems which we tackle in the following steps. The next step (Section 4.4) tries to cure possible problems by reorganizing traits whereas Section 4.5 improves the implementation at the level of methods.

The process can then be applied iteratively to the identified traits and to other classes in the system.

4.2 An Initial Solution

Our first step is to identify the main responsibilities of the class, create a trait for each of them and then move as many methods as possible into the traits without producing too many requirements at the same time.

In our running example of PureBehavior the identified responsibilities – suggested by the method protocol names – are:

- Method dictionary
- Tracking local/non-local methods
- Compiling/decompiling
- Update-mechanism for traits
- Managing the trait composition

Figure 4.1 illustrates the corresponding traits which we further refine in the next step of the process.

At the current state almost all methods are moved from the class to the traits. The behavior could be quite clearly separated so that there were only a moderate number of methods required by most of the traits.

Still it is only the first step and a starting point for further improvements. In the following section we show how a structure such as our present one can be evaluated. We provide a set of simple heuristics and discuss problems and possible solutions.

4.3 Design Heuristics

In this section we identify different kinds of dependencies based on self-sends which facilitate the reasoning about quality aspects of a particular solution. We do not take super-sends or method sends to other objects than self into account.

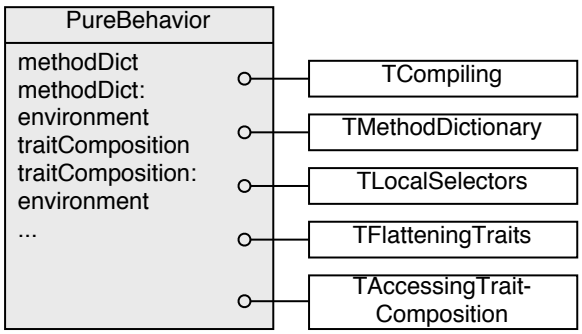

Figure 4.1: First step: decomposing PureBehavior into five traits.

Based on the different types of dependencies we present simple heuristics to reason about defects. In the subsequent sections we apply the heuristics in our process of refactoring the class PureBehavior.

Analyzing the internal structure of a class that is composed of several traits, and the traits themselves composed of other traits, we can identify the following kinds of dependencies at the level of methods:

- (1) *Glue method*: A local method (*i.e.,* defined in a class) invoking methods that are defined in applied traits. Such methods are commonly used to bring the functionality from different traits together. If there are method name clashes, aliasing can be used to access methods with identical names in different traits.

- (2) *Satisfying required method*: A method in a trait sends a message to self which is defined neither in the trait itself nor in a sub-trait of the trait. When using this trait in a class the class has to provide a method with this name to satisfy the requirement. It can do this by implementing the method locally or by using a trait which provides such a method:

  - (2.1) *Locally satisfying a required method*: The class (or one of its superclass) provides a method with the same name like the one of the required method.
  - (2.2) *Required method satisfied by another trait*: A trait's required method is not provided by the class but by another trait (not the trait which poses the requirement of course).
  - (2.3) *Required method indirectly satisfied by another trait*: As in (2.2), but the required method originates in a sub-trait of one of the traits of the class. If we use the flattening property at the trait this situation is exactly like (2.2) because the trait itself would declare the required method.

- (3) *Method calls of methods defined in the same trait*: Self-sends of methods that are defined in the trait itself.

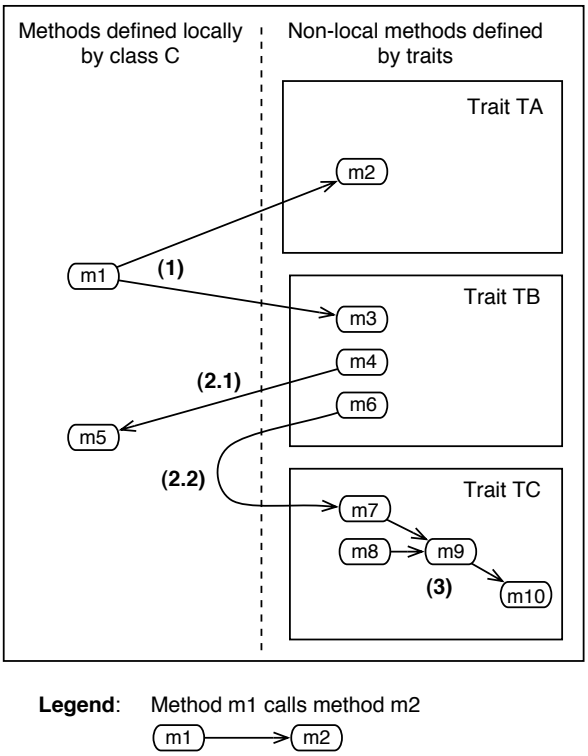

Figure 4.2: Dependencies at the level of message sends to self.

Figure 4.2 illustrates these dependencies. The surrounding box is the context of our investigation: a class named C with the trait composition TA + TB + TC. On the left hand side the figure shows the methods m1 and m5 which are defined locally by C. On the right side of the vertical line the non-local methods of C are shown. An arrow between two methods, m1 and m2 for example, indicates that m1 invokes the method m2. Furthermore the arrow discloses which method is called for a specific message send. In the example of m1 and m2 the method m2 which is defined in the trait TA is invoked. Note that we consider the trait composition not having any conflicts.

A *glue method* (1) is illustrated by the method C≫m1 which sends messages with selectors m2 and m3 to self. There are no local methods with identical selectors that would take precedence over trait methods. Hence, the actually called methods are TA≫m2 and TB≫m3.

C≫m5 locally satisfies the requirement that originates in C≫m4 (2.1) whereas there is no local method that satisfies the requirement of TB≫m6 to provide a

method with selector m7. This requirement is satisfied by TC by providing the method TC≫m7. This means that the requirement is satisfied by another trait in the composition (2.2). The figure doesn't show the dependency (2.3).

Last but not least Figure 4.2 shows methods that send messages to self which are directly satisfied by other methods within the same trait (3). Those methods form a cohesive group by calling each other.

Identifying problems. Using the dependencies (1) to (3), we are able to identify problems and argue about the fitness of a concrete solution.

Ideally a class and its used traits are configured in a way which minimizes dependencies between them. The class is responsible for weaving the different parts provided by the traits which is done by glue methods (1). A class should not define a lot of extra behavior itself and a trait should only be responsible for one concrete purpose. Often methods within a trait call each other (3) but there should not be many calls to the outside. Those outside calls become requirements that have to be satisfied later on by the user of the trait (2). Thus, within a trait we aim at high cohesion whereas between traits and the user we aim at minimizing dependencies.

Reorganizing traits. An interesting criterion when designing traits is the *number of required methods* of a trait. If this number is small – in our experience about 5 – it is easier to use the trait as when the number is big. This number can be put into relation with the number of provided methods: one might argue that the number of required methods can be bigger than, let's say 5, if the size of the trait is big as well, let's say 20. On the other side a big size can be a sign that the trait defines too much behavior and could be split into several traits. The new traits can be organized in two different ways depending on the situation: the traits are made fully independent of each other (not having any dependencies). This is the ideal and simplest case which enhances reusability. The second possibility is to split the original trait into more than two traits by moving groups of behavior into sub-traits of the original trait. Latter now defines glue methods to implement the weaving behavior.

Note that the argumentation on how to structure classes and traits is the same at the level of traits and sub-traits.

Reorganizing methods. If we are not able to reduce the number of required methods any more by reorganizing traits it is valuable to investigate into *how the requirements are satisfied* (2.1) - (2.3). The normal case is that the required method is defined locally by the user (2.1) *i.e.*, by the class which uses the trait. This is often the case for accessor methods because traits cannot access instance variables directly. The case where the requirement is satisfied by another trait (2.2) or (2.3) can be critical and should be addressed. The problem here is that the traits are coupled because one depends on functionality provided by another.

We identified two possible solutions that can improve the quality of the design in this case: (i) the trait which satisfies the required method is made a sub-trait of the trait posing the requirement. Or (ii) the method which poses the requirement should not be defined by the trait but rather defined locally by the user. Moving the method from the trait to its user has the effect that the method becomes a glue method (1).

Both solutions are discussed in detail in the following sections (Section 4.4 and Section 4.5) on the example of improving the kernel class PureBehavior.

4.4 Improvements at the Trait-Level

We now come back to the process of decomposing the class PureBehavior. The current structure is illustrated in Figure 4.1. Looking in more detail at the set of requirements of the trait TCompiling shows that most of the methods fulfilling those requirements are not defined in the class PureBehavior but in the other trait TMethodDictionary.

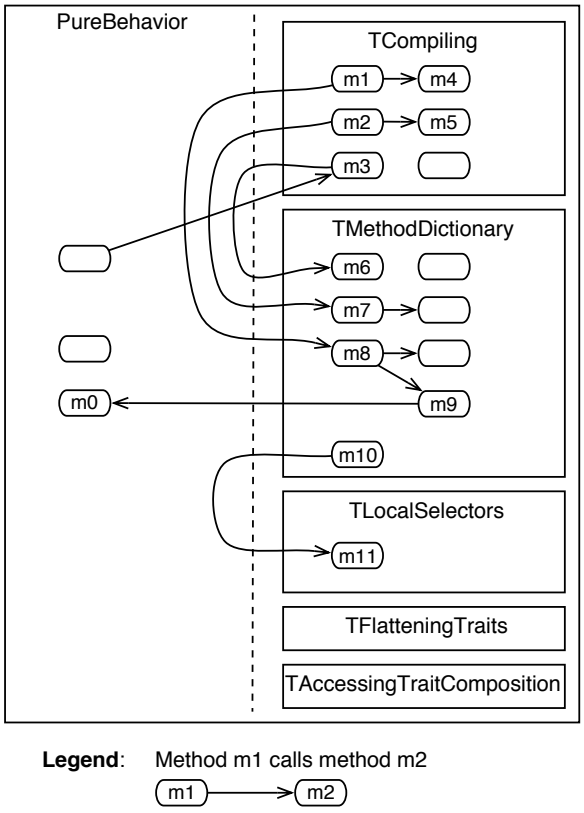

Figure 4.3: Dependencies between TCompiling and TMethodDictionary

This situation is illustrated by Figure 4.3 in the same manner as discussed in the previous section. The critical requirements are addSelector:withMethod:notifying:, basicAddSelector:withMethod: and selectorsDo: (denoted in the figure with m6, m7 and m8 which are called by the methods m1, m2 and m3 in TCompiling).

Problem. This is problematic not because of the number of requirements (compared to the number of providing methods in the trait, this number is even lower) but because of the fact that the required methods are not locally defined by the user (in this case the class PureBehavior) but by one of the other traits of the user. This corresponds to the dependency (2.2) discussed in the previous section. The problem is that this structure introduces dependencies between the traits which makes them more difficult to reuse because they cannot be applied alone.

Solution. To cure this problem a possibility to consider is moving the methods from TMethodDictionary back into the class so that these required methods get satisfied by locally defined methods. But then we get additional requirements from TMethodDictionary because the moved methods conceptually belong into that trait: they call methods from TMethodDictionary. The same arguments apply if we move the methods which cause the requirements from TCompiling back into the class.

The situation is due to the fact that the responsibility of compiling is related to the method dictionary and its behavior: compiling a method in a class or in a trait implicitly means to add the compiled method to the method dictionary. Only few basic methods of the compiling protocol do not directly or indirectly depend on a method dictionary.

From this point of view the compiling behavior depends on the method dictionary behavior: Hence TMethodDictionary has to be a sub-trait of TCompiling and it can be removed from the composition of PureBehavior. The previously required methods are now directly satisfied by the sub-trait. The effect is that TCompiling has less required methods of its own although it obtains additional required methods from the sub-trait *e.g.,* methodDict sent by the method sourceCodeAt: in TMethodDictionary.

Additional refinement. To additionally refine our composition we identify the locally defined methods in TCompiling which do not depend directly or indirectly on the sub-trait TMethodDictionary. The most important ones are: basicCompile:notifying:trailer:ifFail:, decompile: and bindingOf:. The other methods are constant methods that return the compiler, parser and decompiler classes.

We can now move these methods to a new trait which we name TBasicCompiling and which we also make a sub-trait of TCompiling. Our trait TCompiling now only defines glue methods that use the basic compiling and the method dictionary behavior. An example is the following method.

```
TCompiling≫compile: code notifying: requestor
   | methodAndNode |
   methodAndNode := self
      basicCompile: code
      notifying: requestor
      trailer: self defaultMethodTrailer
      ifFail: [↑ nil].
   ....

   self
      addSelector: methodAndNode selector
      withMethod: methodAndNode method
      notifying: requestor.
   ↑ methodAndNode selector
```

The method compiles the argument *code* in the context of the receiver and installs the result in the receiver's method dictionary. To accomplish this task the method uses the basic compile behavior and the behavior to add a new method to the method dictionary which are both defined in its sub-traits.

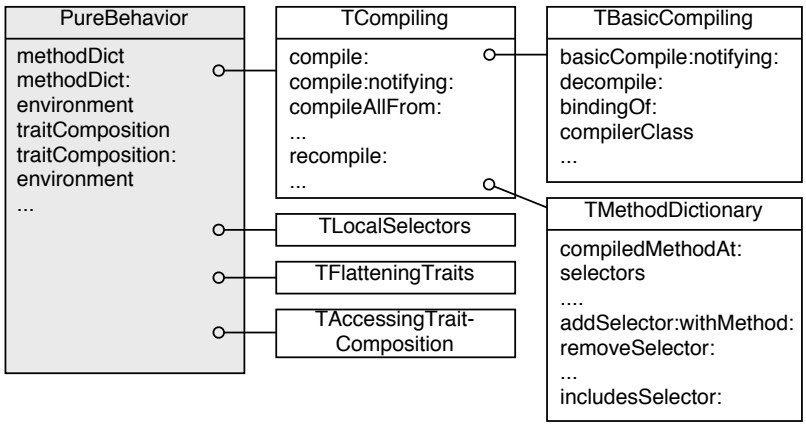

Figure 4.4: Decomposed compiling and method dictionary behavior.

Discussion. Figure 4.4 shows the corresponding class/traits diagram: The class PureBehavior is decomposed into four traits (note that after the first step of our procedure there were five traits). TCompiling is responsible for the compilation and management of methods. It is itself composed of the traits TBasicCompiling and TMethodDictionary. Figure 4.4 shows where the most important methods are defined.

Figure 4.5 illustrates the new call graph after reorganizing the traits as discussed above. Compared to before (Figure 4.3) there are less dependencies between traits but more glue methods like the method TCompiling≫compile:notifying:

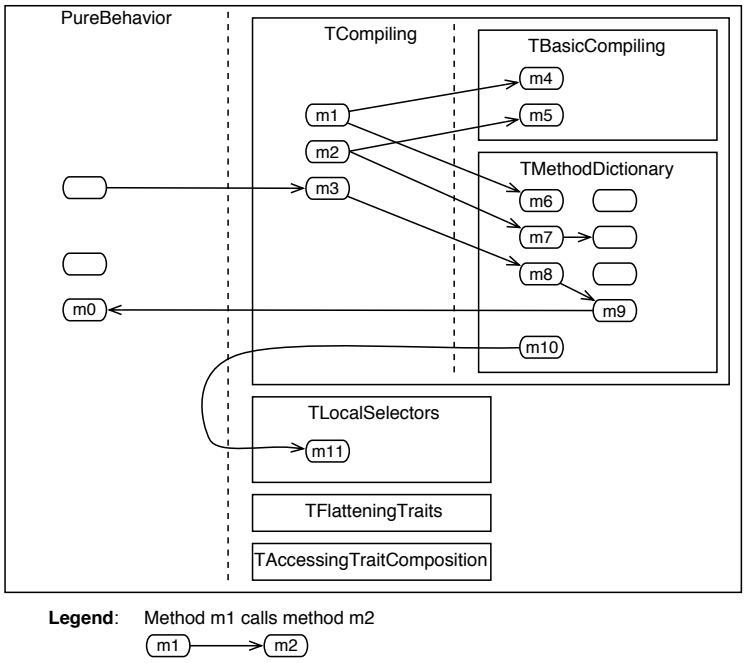

Figure 4.5: Result after second step: reduced dependencies after reorganizing traits.

which we discussed above.

The trait TCompiling only has the following requirements left:

- methodDict
- canUnderstand:
- environment
- Several methods like registerLocalSelector: which are defined in TLocalSelectors and TFlatteningTraits.

The first three requirements are indirectly propagated by the sub-trait TMethodDictionary. Figure 4.5 illustrates this situation by the method sourceCodeAt: (m9) posing the requirement methodDict (m0) which is a instance variable accessor defined in the class.

The remaining required methods of TCompiling are fulfilled by the traits TLocalSelector and TFlatteningTraits. This situation is illustrated in Figure 4.5 by the requirement m11 posed by m10. The problem is tackled in the next step of our process (Section 4.5) in which we improve our solution at the level of methods.

4.5 Improvements at the Method-Level

TMethodDictionary has still some requirements: A required method is methodDict (which is the instance variable accessor defined in the class) but there are also requirements that are satisfied by other traits. These are registerLocalSelector:, includesLocalSelector:, ensureLocalSelectors (defined by TLocalSelectors) and noteChangedSelectors: (TFlatteningTraits).

As in the previous section we have the situation where the required methods are not locally defined by the user but by one of the traits of the user. As discussed in Section 4.3 this is a sign that there might be suboptimal distributed responsibilities.

Now we discuss the problem and a solution to the example of the method addSelectorSilently:withMethod:. It is defined in TMethodDictionary and has a self send to registerLocalSelector: which poses one of the above requirements. This situation is illustrated in Figure 4.5 by the requirement m11 posed by m10.

Analysis. Ideally, TMethodDictionary should know neither about managing local or non-local selectors nor about the flattening algorithm. Thus, the solution to reduce the number of required methods by letting TMethodDictionary itself use the trait TLocalSelectors does not make much sense. From its name addSelectorSilently:withMethod: is correctly put in TMethodDictionary. But the implementation reveals that the method delegates the task of adding the method to the method dictionary *and* additionally registers this method as being a local method:

```
TMethodDictionary≫addSelectorSilently: selector withMethod: aMethod
  self basicAddSelector: selector withMethod: aMethod.
  self registerLocalSelector: selector
```

Registering the method is not the task of the method dictionary because it should not know anything about local or non-local methods to be reusable in other contexts. We are faced with the situation where on one hand the method belongs to the trait and on the other hand it should be implemented locally by the user. In the latter case it would be glue code. Still, this is not a perfect solution.

The method acts more as a hook method: it provides the possibility to perform additional actions when a method is added to the method dictionary. This argument additionally is strengthened by the fact that the subclasses override this method *e.g.*, TraitBehavior specializes the method to notify users of the changed selector.

Solution. Our solution makes use of aliasing and overriding: the idea is to let the class do the registering but at the same time use the original implementation in the trait. To accomplish this task we redefine the method in the class (it will take precedence over the trait method) and let it do the registering. Since we do not want to shadow the original method in the trait we create an alias to it. This facilitates the access of the original method. Now this method does not have to know about the registering anymore. The three steps are in detail:

First we remove the second line of the method in TMethodDictionary. Now it only sends basicAddSelector:withMethod:.

```
TMethodDictionary≫addSelectorSilently: selector withMethod: aMethod
  self basicAddSelector: selector withMethod: aMethod
```

Then we add an alias with the name methodDictAddSelectorSilently:withMethod: to this method in our composition clause of PureBehavior to be able to call the original method when we override it. Note that the trait TMethodDictionary which defines our questioned method is not directly used by PureBehavior but by TCompiling (due to our refactoring in Section 4.4). Nevertheless we define the alias in the composition clause of PureBehavior because the method is indirectly provided by TCompiling.

```
Object subclass: #PureBehavior
  uses:
    TCompiling @ {#methodDictAddSelectorSilently:withMethod:→
      #addSelectorSilently:withMethod:} +
    TLocalSelectors +
    TFlatteningTraits +
    TAccessingTraitComposition
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Traits-Kernel'
```

Lastly we override the original method in the class. This method now sends registerLocalSelector:.

```
PureBehavior≫addSelectorSilently: selector withMethod: aMethod
  self methodDictAddSelectorSilently: selector withMethod: aMethod.
  self registerLocalSelector: selector
```

Discussion. Figure 4.6 illustrates our last improvement. Compared to Figure 4.5 the method registerLocalSelector: (m10) doesn't have a requirement anymore. The newly added method in PureBehavior with the same name, m10, shadows the original method. To be able to use its implementation we create an alias methodDictAddSelectorSilently:withMethod: (a10) to it. The newly added local method m10 now acts as a glue method weaving behavior of the traits TMethodDictionary and TLocalSelectors.

This solution removes the dependency between the method dictionary and the local selector trait and so the unwanted requirement vanishes. The new method in the class acts as a glue method which refines the original method and adds the additional registering call to the behavior defined in another trait. The aliasing mechanism enables a specialization of the original method. In contrast to duplication without calling the original one this ensures consistency in the case where the original implementation in the trait changes.

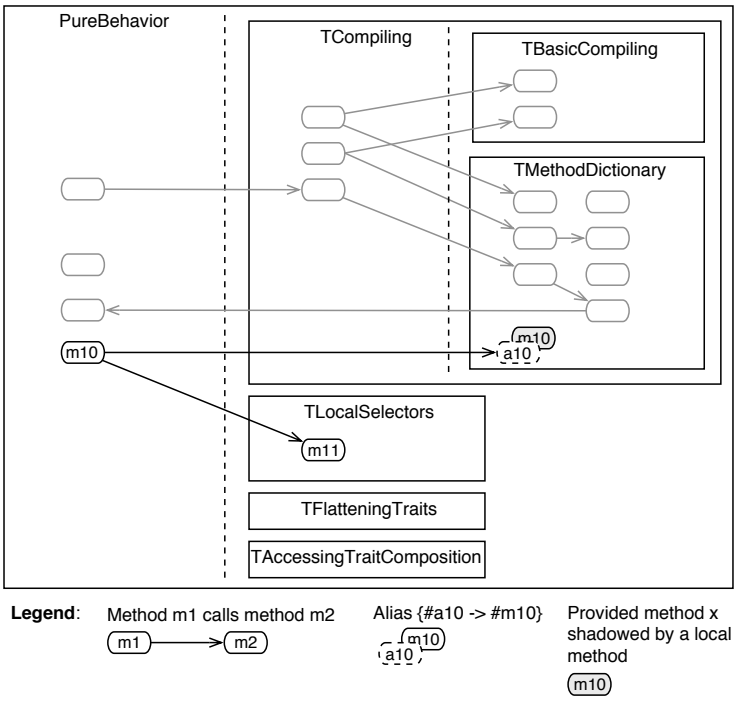

Figure 4.6: Removed dependency by refactoring at the method-level.

Chapter 5

Traits as Class Properties

In pure object-oriented languages, classes are objects, instances of other classes called metaclasses. In the same way that classes define the properties of their instances, metaclasses define the properties of classes. It is therefore very natural to wish to reuse class properties, utilizing them amongst several classes. However this introduces *metaclass composition problems, i.e.,* code fragments applied to one class may break when applied to another class due to the inheritance relationship between their respective metaclasses.

This section discusses how traits are used to compose metaclasses from traits. This approach facilitates the modelling of class properties in a uniform way: like all the other classes in the system, metaclasses are composed out of traits [DUCA 05]. This solution supports the reuse of class properties and their safe and automatic composition based on explicit conflict resolution.

5.1 Metaclass Composition Problems

As discussed in Section 3.1.1 Smalltalk-80 uses the concept of implicit metaclasses which makes reusability of class properties hard.

In contrast to implicit metaclasses, *explicit* metaclasses provide the possibility to reuse metaclass properties across different classes *i.e.,* the *Singleton* design pattern [ALPE 98] does not have to be re-implemented each time it is used. On the other hand implementations using explicit metaclasses open the door for compatibility problems between the class and metaclass level [GRAU 89] [COIN 87]: especially *upward* or *downward compatibility* may not hold. Numerous approaches to solve metaclass composition problems have been attempted, but they always resort to an ad-hoc manner of handling conflicting properties [BOUR 98].

5.2 Using Traits to Reuse and Compose Class Properties

Traits offer the means for a uniform approach that represents class properties as traits to compose classes. This facilitates safe and uniform metaclass composition

[DUCA 05]. In the same way that classes can be composed of traits at the base level, metaclasses can use traits. This solution makes possible the reuse of class properties, and their safe and automatic composition based on explicit conflict resolution.

Because our approach is based on using traits within the traditional parallel inheritance schema proposed by Smalltalk-80 our approach is safe *i.e.*, it supports downward and upward compatibility [BOUR 98]. But on top of that it promotes the reuse of class properties. Composition and application of class properties are based on trait composition, which gives the programmer explicit control in a uniform manner.

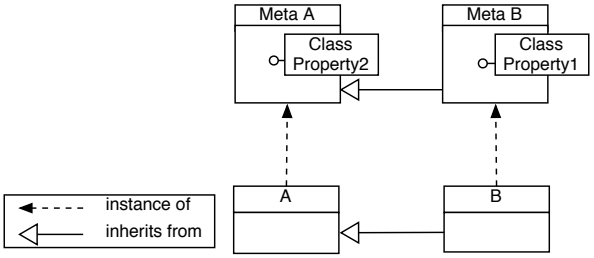

Figure 5.1: Metaclasses are composed of traits representing class properties. Traits support upward and downward compatibility.

We represent class properties as traits, which are then used to compose metaclasses as shown in Figure 5.1.

During our refactoring of Squeak code we identified the following class properties: TInstantiator, TAbstract, TSingleton and TFinal which we explain below. We start with a discussion of our implementation of the class property *Singleton*. Section 5.4 then illustrates its application on the traditional Boolean hierarchy [LEDO 96, BOUR 98] and finally Section 5.5 shows that traits provide a good basis to re-engineer the meta-level.

5.3 Singleton

To represent the fact that a class is a Singleton [GAMM 95], we define the trait TSingleton. Its responsibilities are to control that a class only has one instance and to provide a global point of access to it.

In Smalltalk there are two ways to create new objects: (i) by instantiating a class (by sending the message basicNew to a class) and (ii) by copying an existing object. The two methods which are eventually executed by those calls, basicNew and shallowCopy, each executes a primitive of the virtual machine.

We first clarify the basic instance creation concept of Smalltalk. In Smalltalk, creation of a new instance involves two different methods, namely basicNew and

new[1]. The method basicNew is a low-level primitive which simply allocates a new instance of the receiver class. The method new stands at a conceptually higher level and its purpose is to return a usable, initialized instance of the receiver class. For most classes, new therefore calls basicNew to obtain a new instance and then initializes it with reasonable default values.

To satisfy the responsibility to control the instantiation of new objects, TSingleton has to redefine the corresponding methods that are accessible through inheritance. The behavior of instantiating new instances from a class (i) is defined on the class side (basicNew is defined in Behavior) whereas copying (ii) is defined on the instance side (shallowCopy is defined in Object). Hence, the trait TSingleton has to take effect on the class side as well as on the instance side.

TSingleton is defined as a trait pair: two connected traits that define instance- respectively class-specific behavior (see Section 3.6 for more details on trait pairs). When TSingleton is applied to a class it automatically applies its corresponding classtrait, TSingleton classTrait, to the metaclass of this class.

Figure 5.2 shows the trait TSingleton which is composed of an instance side and a class side trait.

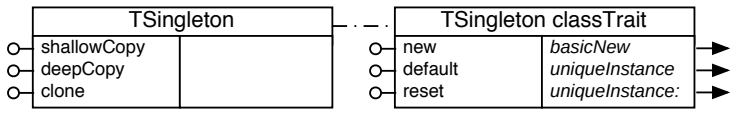

Figure 5.2: The trait TSingleton

TSingleton. On the instance side the following methods are defined: shallowCopy, deepCopy and clone. They are all implemented as empty methods and thus return the object receiving the message[2] without copying it.

```
Trait named: #TSingleton
   uses: {}
   category: 'Traits-Kernel'

TSingleton≫shallowCopy
   "Cannot create copies of an instance of a Singleton."
```

TSingleton classTrait. The class side of the trait defines the following methods: default which returns the default instance, new which raises an error, and

[1]Note that there are also the methods basicNew: and new:, which are used to create objects with indexed fields (*i.e.,* arrays). For the sake of simplicity, we do not take these methods into account here.

[2]A method which does not explicitly return an object implicitly returns self, the receiver of the message.

reset which invalidates the current Singleton instance. It requires basicNew which returns a newly created instance[3], and the methods uniqueInstance and uniqueInstance:. Note that these accessors methods are needed because traits cannot contain instance variables.

```
TSingleton classTrait
  uses: {}
  category: Traits-Kernel

TSingleton classTrait≫default
  self uniqueInstance ifNil: [self uniqueInstance: self basicNew].
  ↑ self uniqueInstance

TSingleton classTrait≫new
  self error: 'Cannot create new instances of a Singleton. Use default instead.'

TSingleton classTrait≫reset
  self uniqueInstance: nil
```

The class side trait has the required methods uniqueInstance and uniqueInstance: as Figure 5.2 shows. They are supposed to be accessors to the corresponding instance variable which stores the unique instance of the class. Furthermore it requires basicNew and error:. When TSingleton is applied to a metaclass basicNew will be provided by the indirect superclass Behavior. The requirement error: is fulfilled in any case since this method is implemented in Object.

Discussion. Using a trait pair to model Singleton stands in contrast to Ducasse et al. [DUCA 05] who propose to use a trait that only controls the class instantiation behavior and thus is directly applied to the class side. We propose to let the Singleton trait also control the instance side behavior since it is crucial to assure the right behavior. In our solution the trait TSingleton is not a pure class property the way TAbstract or TFinal are: it is applied to the instance side of a class which might be confusing at first. But by controlling the copying methods – rather than leaving the task of manually defining these methods to the developer – it enhances understandability and robustness: the definition of those methods in the trait directly communicates what they are used for; it might be less obvious if they are redefined in the class. Furthermore the right behavior is used automatically, which prevents possible bugs when the developer forgets to implement this behavior manually.

The next section demonstrates how we used the class properties TSingleton and TAbstract to refactor the Boolean hierarchy.

[3]Using basicNew is the traditional way to implement Singleton in Smalltalk when we want to forbid the use of the new method [ALPE 98]. basicNew allocates objects without initializing them. It is a Smalltalk idiom to never override methods starting with basic names.

5.4 The Boolean Hierarchy Revisited

The Smalltalk Boolean hierarchy consists of the abstract class Boolean, which has two Singleton subclasses True and False. Using traits the Boolean hierarchy is refactored as shown in Figure 5.3. Note that we designed the refactored solution to be backward compatible with the idioms existing in the current Smalltalk implementation and literature [ALPE 98]. So we assume that a method basicNew is defined on the class Behavior, can always be invoked to allocate instances and should not be overridden.

Boolean. The class Boolean is an abstract class, so we compose its class Boolean class from the trait TAbstract.

```
Trait named: #TAbstract
   uses: {}
   category: 'Traits-Kernel'

TAbstract≫new
   self error: 'Abstract class. You cannot create instances'

TAbstract≫new: size
   self error: 'Abstract class. You cannot create instances'
```

And the class definition of the metaclass Boolean class:

```
Boolean class
   uses: TAbstract
   instanceVariableNames: ''
```

True and False. The classes True and False are Singletons so they are composed of the trait TSingleton.

As mentioned above in Section 5.3, the class side of the trait TSingleton requires the methods basicNew, uniqueInstance, and uniqueInstance:. Therefore the class True class (resp. False class) has to define an instance variable uniqueInstance and the two associate accessors methods uniqueInstance and uniqueInstance:. Note that the method basicNew does not have to be redefined locally in True class or False class as it is inherited ultimately from the class Behavior. This example shows that class properties are reused over different classes and that metaclasses are composed of different properties.

```
Boolean subclass: #True
   uses: TSingleton
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Kernel-Objects'
```

```
True class
   uses: TSingleton classTrait
   instanceVariableNames: 'uniqueInstance'

True class≫uniqueInstance
   ↑ uniqueInstance

True class≫uniqueInstance: anObject
   uniqueInstance := anObject
```

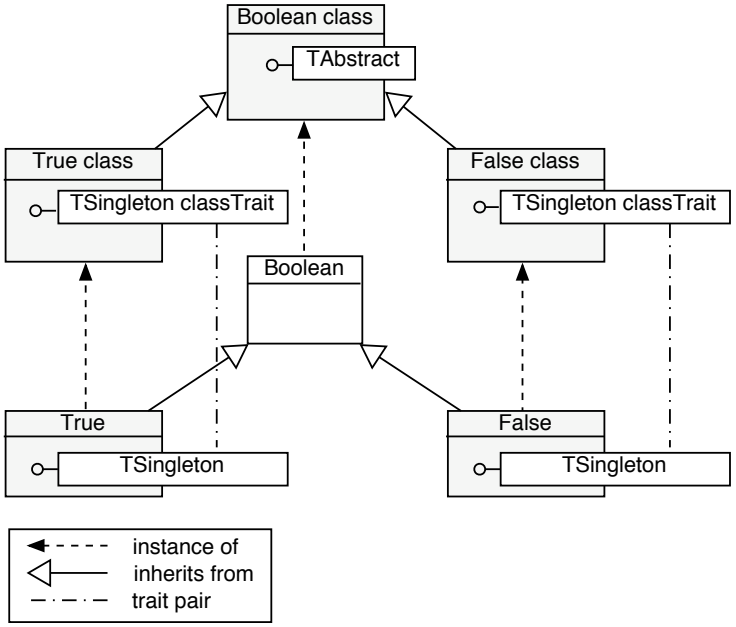

Figure 5.3: Boolean hierarchy refactored with traits.

Figure 5.3 illustrates the refactored Boolean hierarchy. Before our refactoring the *copy behavior* was defined in the class Boolean although it is not a Singleton! Conceptually those methods should be implemented in the two Singleton classes True and False. The reason why these methods were not implemented in those two classes was the need to facilitate code reuse. This example shows that class properties as discussed here (and traits in general) provide the means to model this kind of behavior in an elegant and more uniform way than single inheritance. The refactored Boolean hierarchy better reveals the intentions of the programmer now. Since the copying methods are implemented in the trait TSingleton it is obvious what their purpose is. This was not the case before refactoring.

The next section shows how class properties presented above can further be refined to engineer the meta-level with traits.

5.5 Fine-grained Architecture of Class Properties

So far we have presented simple examples that show how traits are well-suited to modeling class properties, which can then be combined or applied to arbitrary classes. In this section, we show that traits also allow more fine-grained architectures of class properties. We also want to stress that the techniques used here at the meta-level are exactly the same as those used at the base level. As such, traits provide a uniform model.

Figure 5.4 gives an overview of the class properties we identified. Note that all of these properties are traits, and that they are therefore composed using trait composition. The left side of a box representing a trait only shows the additional provided methods to the methods that are indirectly provided by sub-traits. Likewise, the right side only shows the additional requirements.

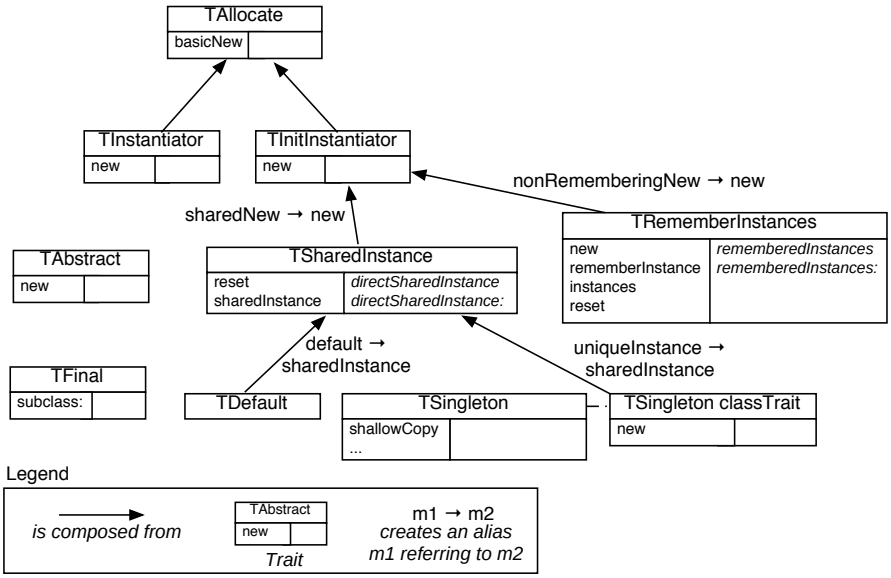

Figure 5.4: A fine-grained architecture of class properties based on traits.

Allocation. As indicated by its name, the trait TAllocator provides the behavior to allocate new instances. In our case, this is the standard Smalltalk basicNew method, but of course we could also create another trait with an alternative allocation strategy.

Instantiation. The traits TInstantiator and TInitInstantiator are two 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 uses the trait TAllocator. However, as suggested by its name, it actually initializes the newly created instance by calling the method initialize before the instance is returned.

```
TInstantiator≫new
  ↑ self basicNew

TInitInstantiator≫new
  ↑ self basicNew initialize
```

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 traits TInitInstantiator which is then available as nonRememberingNew. This aliasing allows one to access the original new method of the trait TInitInstantiator while leaving the option to override the method new in the trait TInitInstantiator. It requires the methods rememberedInstances and rememberedInstances: to access a collection storing the created instances. Then, it implements the methods new, rememberInstance:, instances and reset as follows:

```
TRememberInstances≫new
  ↑ self rememberInstance: self nonRememberingNew

TRememberInstances≫rememberInstance: anObject
  ↑ self instances add: anObject

TRememberInstances≫instances
  self rememberedInstances ifNil: [self reset].
  ↑ self rememberedInstances

TRememberInstances≫reset
   self rememberedInstances: IdentitySet new
```

Note that another implementation could be to define the methods reset and rememberedInstances: as trait requirements. This would leave the class with the option to use 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. Whereas a Singleton can only have one single instance, a class adhering to the Default Instance pattern has one default instance but can also have an arbitrary number of other instances.

Since these two properties are very 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. Then, it implements the methods reset and sharedInstance as follows:

```
TSharedInstance≫reset
  self directSharedInstance: self sharedNew

TSharedInstance≫sharedInstance
  self directSharedInstance ifNil: [self reset].
  ↑ self directSharedInstance
```

The property TDefault is then defined as an extension of the trait TSharedInstance that simply introduces the alias default for the method sharedInstance. Similarly, the property TSingleton classTrait introduces the alias uniqueInstance for the same method. In addition, TSingleton overrides the method new so that it cannot be used to create a new instance. In Section 5.3 we discuss the class property Singleton in more detail and Section 5.4 shows how we applied it to the Boolean hierarchy. Note that for sake of simplicity the trait TSingleton is not composed of other traits in these two sections.

Another useful class property popularized by Java is the class property TFinal which ensures that a class cannot have subclasses. In Smalltalk, this is achieved by overriding the message subclass:[4]. Note that unlike all the other properties presented in this section, TFinal is not concerned with instance creation and therefore is entirely independent of the other properties.

5.6 Explicit Composition Control Power

Having an architecture of class properties has many advantages for a programmer. Whenever a new class needs to be created, a choice can be made regarding the creation of instances, and whether or not the class should be final. Besides having the obvious advantage of avoiding code duplication, it also makes the design much more explicit and therefore facilitates understandability of the class. The level of abstraction of the trait design is at the right level: the traits correspond to the class properties, and the class properties can be combined into metaclasses.

In addition, factoring out the properties in such a fine-grained way still gives the user a lot of control about some crucial parts of the system. Suppose for example that at first we would have decided to use the trait TInitInstantiator as the basis for all the other instance creation properties. If later on, we would decide to comply to the Smalltalk standard to create uninitialized instances by default, then we could make this change without modifying any of the involved methods. We would just need to make sure that the traits TRememberInstances and TSharedInstance use the trait TInstantiator instead of TInitInstantiator.

By providing several different properties that are all related to instance creation behavior, this example also shows why it is so important to have explicit control over composition and application of class properties. In our example, there are many different properties which essentially introduce variants of the method new, and therefore, combining these properties typically leads to conflicts that can only

[4] In reality, the method to create a subclass takes more arguments but this is not relevant here.

be resolved in a *semantically* correct manner if the user has explicit control over the composition. In case of traits, this is ensured by allowing partially ordered compositions, exclusions, and aliases.

As an example, imagine that we want to combine the properties TDefault and TRememberInstances to get a property that allows both a default instance and also remembers all its instances. With our trait-based approach, we do this by creating a new trait TDefaultAndRememberInstances which uses TRememberInstances and TDefault as follows:

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

TDefaultAndRememberInstances≫sharedNew
   ↑ self storeNew

TDefaultAndRememberInstances≫reset
   self storeReset.
   self defaultReset
```

Since both traits provide a method new, we exclude this method from the trait TRememberInstances when it is composed. As a consequence the 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 new method provided by TRememberInstances.

Because the method reset is also provided by both traits, we use aliasing to make sure that we can access the conflicting methods. Then, we 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.

Chapter 6

The Bootstrapped Trait Kernel

Figure 6.1 illustrates the classes of the bootstrapped trait kernel. The original class hierarchy presented in Section 3.2 and illustrated by Figure 3.2 is left unchanged but most of the classes were decomposed into traits.

Figure 6.1 shows classes with gray background color to better distinguish them from traits. The names of all traits start with an uppercase T by convention. The names we chose for the traits used in the kernel at the level of Trait- and ClassDescription end with "Description" in some cases to avoid name clashes or to make their intent more clear. An example is TCompiling and TCompilingDescription: the first one is the basic compiling behavior used by the class PureBehavior whereas the latter is used by the two classes TraitDescription and ClassDescription.

Note that the trait TCompiling actually is composed itself from two sub-traits. The figure does not show these two traits for the sake of simplicity. Details of this decomposition are discussed in Section 4.4 and is illustrated by Figure 4.4.

6.1 Kernel Classes Composed of Traits

PureBehavior. The decomposition of the class PureBehavior is discussed in detail in Section 4.1. PureBehavior is decomposed into the four traits TCompiling, TLocalSelectors, TFlatteningTraits and TAccessingTraitComposition which is illustrated by Figure 4.4. The trait TCompiling itself is composed of two traits: TBasicCompiling and TMethodDictionary. TCompiling implements glue methods like compile: which use provided methods from the two sub-traits, *e.g.*, to compile code and then add the new method to the method dictionary. The trait TLocalSelectors manages the (de-)registering of local and non-local methods; TFlatteningTraits is the core flattening algorithm (see Section 3.5). Last but not least, TAccessingTraitComposition provides convenience methods to access the trait composition, *e.g.*, to query the origin of a method.

Behavior. The class Behavior is composed of only one trait, TInstantiator, which provides behavior for instantiating new objects from classes.

ClasssDescription and TraitDescription. The most fine-grained decomposition occurs in the classes ClassDescription and TraitDescription, which are each composed of ten traits, nine of which are shared between the two classes. The large amount of common behavior is due to the fact that the new kernel mimics the original organization that defined the class-/metaclass behavior on several levels – Behavior, ClassDescription, Class/Metaclass. To keep this kind of organization we didn't move the common behavior between the classes ClassDescription and TraitDescription up into the common superclass PureBehavior but rather share it via traits.

Class. The class Class is composed of the following traits: TRemoveClassFromSystem which provides the behavior to remove a class and make it obsolete. TFileInOut supports filing-in and -out (saving/restoring) source code of classes. TVariables supports adding and removing instance variables of the class.

Metaclass and ClassTrait. TApplyingOnClassSide is a trait that provides the functionality which is used to keep the trait composition on the instance side in sync with the one on the class side (metaclass or classtrait resp.). In addition it supports checking whether a class side composition is valid or not. This trait is used by Metaclass as well as by ClassTrait.

6.2 Analysis of the Gained Reuse and Understandability

The kernel classes are composed of a total of 24 traits (including all sub-traits). Especially in the case of ClassDescription and TraitDescription traits were successfully used to *eliminate code duplication*: after decomposing ClassDescription the class only defines 40 local methods – 98 methods are provided by traits. TraitDescription looks similar: from a total of 113 methods it defines 28 methods locally (thus, 85 methods come from traits). Almost all traits, nine of ten, are shared between the two classes. The reuse of code at this level of the class hierarchy is especially high.

Another example of code reuse between two classes is Metaclass and ClassTrait. In other places of the kernel there were no traits used by more than one class. Still, traits proved to *leverage understandability* of the system. A good example is the class PureBehavior. The explicit decomposition of the class into conceptual groups of methods directly communicates the main responsibilities of the class to the developer.

Furthermore – following the design heuristics that we identified in Section 4.1 – traits promote a clear design with a minimized number of externally required methods and a high cohesion inside a trait. This not only enhances understandability of responsibilities, it also makes it easier to *identify dependencies* between methods. This makes potential impacts of changes more obvious – something that is especially important when refactoring or enhancing existing code.

Code reuse and explicit communication of intents also come in hand to model *class properties* as discussed in Chapter 5. The amount of code that can be shared is relatively small, though. Most of the identified traits do not define more than four or five methods. Nevertheless, traits are successfully applied to model the meta-level because they provide the means for a safe and uniform composition of class properties. The example of refactoring the Boolean hierarchy (Section 5.4) illustrates how the Singleton and Abstract class properties were applied. Looking at the classes in the class browser after refactoring directly reveals the properties they have. Behavior that conceptually belonged to the two subclasses True and False, but was pushed up into the common superclass Boolean for the sake of reusability, now is defined at the right place. The reusability is achieved by traits so that there was no need to duplicate code.

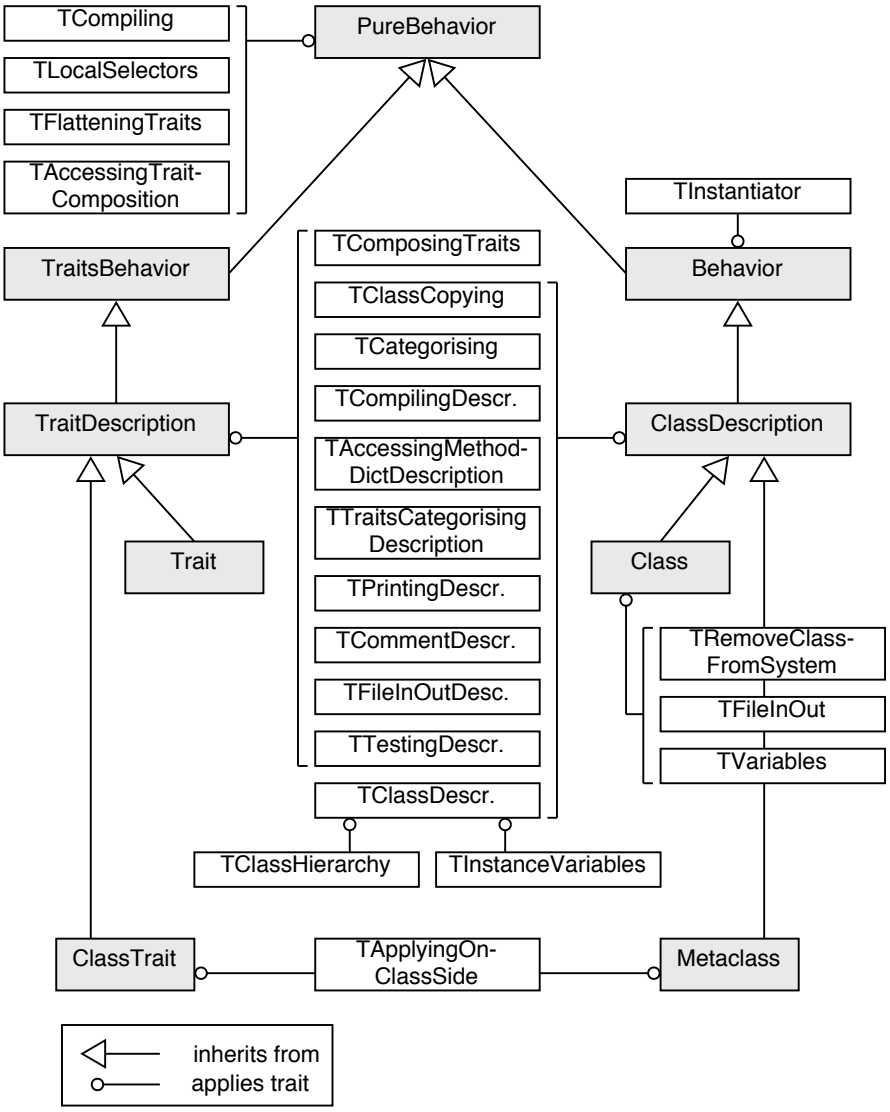

Figure 6.1: The new kernel class hierarchy composed of traits.

Chapter 7

Conclusion and Future Work

7.1 Summary

This thesis presents a new Smalltalk kernel bootstrapped with traits. Its design is aimed at being clean, robust and backward compatible with the traditional kernel. Furthermore by bootstrapping the kernel it follows the fundamental idea of a reflective language: to use the features of the language to define the behavior of the language itself.

The thesis briefly introduces traits and discusses our and an alternative implementation strategy. We discuss the difference, pros and cons of our static design approach and the alternative dynamic design approach. The former approach is based on flattening the method dictionaries whereas the latter modifies the virtual machine's method lookup.

Before presenting the class hierarchy of the new trait kernel we analyze the traditional Smalltalk kernel by identifying the responsibilities of its classes. After introducing the new kernel we present how trait compositions are formed and implemented. It is shown how normal message sends are used to build up the objects representing a trait composition. This leads to the heart of the trait kernel: the flattening algorithm which is responsible for updating the method dictionaries. It facilitates the correct semantics of traits by mixing trait methods into the method dictionary of classes and traits.

By bootstrapping the kernel with traits we gained valuable insight into the process of decomposing an existing program into traits. We identify common patterns at the level of method invocations between traits and their users that help us to emerge heuristics which proved to be valuable guidelines when refactoring the kernel classes.

We furthermore discuss how class properties are implemented and composed in a uniform and safe way. As an example we show how we refactored the Boolean hierarchy of Smalltalk. The thesis eventually presents how the bootstrapped kernel is composed of traits. Those traits implement basic groups of behavior, such as managing the method dictionary, compiling, method categorization etc., and are in

some cases shared between classes of the two branches of the class hierarchy.

7.2 Evaluation

Because traits are simple and completely backward compatible with single inheritance, implementing traits in a reflective single inheritance language like Squeak proved to be unproblematic. 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 bytecode 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 is entirely justifiable.

By applying the new trait kernel to express itself we could directly validate the operability of our implementation. The fact that we were able to cleanly bootstrap this part of the system is an evidence that our implementation is well-factored. Furthermore it 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 also 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.

7.3 Main Contributions

The main contributions of this thesis are the following:

- **Evaluation of different implementation strategies:** We analyze two different implementation strategies of traits and discuss their pros and cons.
- **Clean trait kernel architecture:** We present how traits are implemented in Smalltalk in a simple and clean way without the need to change the virtual machine. With our approach a program with traits has the same performance as being built without traits in the traditional system.

- **Bootstrapping of the new kernel:** We show how the trait kernel is bootstrapped with traits to improve its quality and understandability. Furthermore we present how we refactored the Smalltalk Boolean hierarchy using traits representing the class properties Abstract and Singleton.
- **Heuristics to decompose a class into traits:** From our analysis and experience we develop a process and heuristics to decompose a class into traits. We present and illustrate different types of dependencies between traits and classes.
- **Safe and uniform metaclass composition:** We discuss how traits are used to implement class properties and how they solve metaclass composition problems.

7.4 Future Work

An interesting area for future work is that of tools and methodologies which support the implementation and understandability of a traits system. The decomposition of an existing object-oriented program into traits is not a trivial task as we discuss in Chapter 4. We propose heuristics to identify good traits and detect possible defects. Further research into the development of techniques and tools which support the programmer in the task of identifying traits and understanding an existing traits program is promising. This area will be of increasing interest when legacy systems start to be transformed to use traits.

There are two promising paths:

Formal Concept Analysis. Formal concept analysis (FCA) may successfully be used to identify traits in an object-oriented program. FCA is a mathematical technique for clustering abstract entities that share common attributes [BIRK 40] [GANT 99] [WILL 81]. FCA has been applied in computer science, for example to identify modules [SIFF 97], to build and reengineer class hierarchies [GODI 93] [SNEL 98] [SNEL 00] or to detect software patterns [ARÉV 03] [BUCH 03].

In the context of traits, a tool based on FCA can help the developer to decompose a system by proposing potential traits.

Visualizations. The visualization of a system can give immediate clues to its organization. This can dramatically support the process of reengineering and reverse engineering of software systems.

In Section 4.1 we use a simple visualization of the decomposition of a class into traits. It shows the call graph of the methods grouped by the class and its traits. This helps us detect different types of method dependencies which can indicate potential problems.

Our initial ideas can be further developed to generate visualizations and methodologies for understanding systems with traits. This follows the ideas of *class*

blueprints proposed by Lanza et al. [LANZ 01] which help the programmer quickly grasp the purpose of a class and its inner structure. It would be interesting to enhance class blueprints to analyze classes built from traits. Additional to a flattened view of a class, the class blueprint would provide a view that shows the traits of which a class (or recursively a trait) is composed. Since the composition of traits often is orthogonal to the inheritance relationship of classes, the investigation into using a 3D visualization to master the additional complexity could be an interesting idea.

Bibliography

[ALPE 98] S. R. Alpert, K. Brown, and B. Woolf. The Design Patterns Smalltalk Companion. Addison Wesley, 1998. (pp 49, 52, 53)

[ARÉV 03] G. Arévalo. *Understanding Behavioral Dependencies in Class Hierarchies using Concept Analysis*. In Proceedings of LMO 2003: Langages et Modeles à Objets, pages 47–59. Hermes, Paris, January 2003. (p 65)

[ASTE 03] D. Astels. Test-Driven Development - A Practical Guide. Prentice Hall, 2003. (p 2)

[BIRK 40] G. Birkhoff. *Lattice Theory*. American Mathematical Society, 1940. (p 65)

[BLAC 03] A. P. Black, N. Schärli, and S. Ducasse. *Applying Traits to the Smalltalk Collection Hierarchy*. In Proceedings OOPSLA'03 (International Conference on Object-Oriented Programming Systems, Languages and Applications), volume 38, pages 47–64, October 2003. (pp 8, 37)

[BLAC 04] A. P. Black and N. Schärli. *Traits: Tools and Methodology*. In Proceedings ICSE 2004, pages 676–686, Mai 2004. (p 8)

[BOUR 98] N. M. N. Bouraqadi-Saadani, T. Ledoux, and F. Rivard. *Safe Metaclass Programming*. In Proceedings OOPSLA '98, pages 84–96, 1998. (pp 16, 23, 49, 50)

[BUCH 03] F. Buchli. Detecting Software Patterns using Formal Concept Analysis. Diploma thesis, University of Bern, September 2003. (p 65)

[CASA 95] E. Casais. *Managing Class Evolution in Object-Oriented Systems*. In O. Nierstrasz and D. Tsichritzis, editors, Object-Oriented Software Composition, pages 201–244. Prentice-Hall, 1995. (p 37)

[COIN 87] P. Cointe. *Metaclasses are First Class: the ObjVlisp Model*. In Proceedings OOPSLA '87, ACM SIGPLAN Notices, volume 22, pages 156–167, December 1987. (pp 16, 49)

[COOK 92] W. R. Cook. *Interfaces and Specifications for the Smalltalk-80 Collection Classes*. In Proceedings OOPSLA '92, ACM SIGPLAN Notices, volume 27, pages 1–15, October 1992. (p 8)

[DIXO 89] G. Dixon, G. D. Parrington, S. K. Shrivastava, and S. M. Wheater. *The Treatment of Persistent Objects in Arjuna*. In S. Cook, editor, Proceedings ECOOP '89, pages 169–189, Nottingham, July 1989. Cambridge University Press. (p 64)

[DUCA 05] S. Ducasse, N. Schaerli, and R. Wuyts. *Uniform and Safe Metaclass Composition*. Journal of Computer Languages, Systems and Structures, 2005. (pp 16, 23, 34, 49, 50, 52)

[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 15)

[FORM 99] I. R. Forman and S. Danforth. Putting Metaclasses to Work: A New Dimension in Object-Oriented Programming. Addison-Wesley, 1999. (p 16)

[GAMM 95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, Reading, Mass., 1995. (p 50)

[GANT 99] B. Ganter and R. Wille. Formal Concept Analysis: Mathematical Foundations. Springer Verlag, 1999. (p 65)

[GODI 93] R. Godin and H. Mili. *Building and Maintaining Analysis-Level Class Hierarchies Using Galois Lattices*. In Proceedings OOPSLA '93, ACM SIGPLAN Notices, volume 28, pages 394–410, October 1993. (p 65)

[GOLD 83] A. Goldberg and D. Robson. Smalltalk 80: the Language and its Implementation. Addison Wesley, Reading, Mass., Mai 1983. (pp 2, 10)

[GOLD 89] A. Goldberg and D. Robson. Smalltalk-80: The Language. Addison Wesley, 1989. (pp 16, 17)

[GRAU 89] N. Graube. *Metaclass Compatibility*. In Proceedings OOPSLA '89, ACM SIGPLAN Notices, volume 24, pages 305–316, October 1989. (pp 23, 49)

[INGA 76] D. Ingalls. *The Smalltalk-76 Programming System Design and Implementation*. In POPL'76, Principles of Programming Languages, pages 9–16. ACM Press, 1976. (p 16)

[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 i, 15)

[KICZ 91] G. Kiczales, J. des Rivières, and D. G. Bobrow. The Art of the Metaobject Protocol. MIT Press, 1991. (p 2)

[LANZ 01] M. Lanza and S. Ducasse. *A Categorization of Classes based on the Visualization of their Internal Structure: the Class Blueprint*. In Proceedings of OOPSLA 2001 (International Conference on Object-Oriented Programming Systems, Languages and Applications), pages 300–311, 2001. (p 66)

[LEDO 96] T. Ledoux and P. Cointe. *Explicit Metaclasses as a Tool for Improving the Design of Class Libraries*. In Proceedings of ISOTAS '96, LNCS 1049, pages 38–55. JSSST-JAIST, March 1996. (pp 16, 50)

[SCHÄ 02] N. Schärli, O. Nierstrasz, S. Ducasse, R. Wuyts, and A. Black. *Traits: The Formal Model*. Technical Report IAM-02-006, Institut für Informatik, Universität Bern, Switzerland, November 2002. (p 8)

[SCHÄ 03a] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. *Traits: Composable Units of Behavior*. In Proceedings ECOOP 2003 (European Conference on Object-Oriented Programming), volume 2743 of *LNCS*, pages 248–274. Springer Verlag, July 2003. (pp 8, 10)

[SCHÄ 03b] N. Schärli and A. P. Black. *A Browser for Incremental Programming*. Technical Report CSE-03-008, OGI School of Science & Engineering, Beaverton, Oregon, USA, April 2003. (p 8)

[SIFF 97] M. Siff and T. Reps. *Identifying Modules via Concept Analysis*. In Proc. of. the Internation Conference on Software Maintenance, pages 170–179. IEEE Computer Society Press, 1997. (p 65)

[SNEL 98] G. Snelting and F. Tip. *Reengineering Class Hierarchies Using Concept Analysis*. In ACM Trans. Programming Languages and Systems, 1998. (p 65)

[SNEL 00] G. Snelting and F. Tip. *Understanding Class Hierarchies Using Concept Analysis*. ACM Trans. on Programming Languages and Systems, pages 540–582, Mai 2000. (p 65)

[WILL 81] R. Wille. *Restructuring lattice theory: An approach based on hierarchies of concepts*. Ordered Sets, Ivan Rival Ed., NATO Advanced Study Institute, vol. 83, pages 445–470, September 1981. (p 65)