1

PM-1 3 1/2" x 4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

National Library
of Canada

Bibliothèque nationale
du Canada

Acquisitions and
Bibliographic Services Branch

Direction des acquisitions et
des services bibliographiques

395 Wellington Street
Ottawa, Ontario
K1A 0N4

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Not to be used for reference

Not to be used for reference

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

FEATURE INTERACTION IN OBJECT-ORIENTED LANGUAGES

by

Danny Epstein, B.C.S.

**A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of
Master of Computer Science**

Ottawa-Carleton Institute for Computer Science

School of Computer Science

Carleton University

Ottawa Canada

March 30, 1993

© copyright 1993 Danny Epstein

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Author's Acknowledgements

Author's Acknowledgements

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-84101-X

Canada

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

U·M·I

SUBJECT CODE

THE HUMANITIES AND SOCIAL SCIENCES

Psychology	0525
Reading	0535
Religious	0527
Sciences	0714
Secondary	0533
Social Sciences	0534
Sociology of	0340
Special	0529
Teacher Training	0530
Technology	0710
Tests and Measurements	0288
Vocational	0747

COMMUNICATIONS AND THE ARTS	
Architectural	0729
Art History	0377
Cinema	0900
Dance	0378
Fine Arts	0357
Information Science	0723
Journalism	0391
Library Science	0399
Mass Communication	0708
Music	0413
Speech Communication	0459
Theater	0465

LANGUAGE, LITERATURE AND LINGUISTICS

- Education
- General
- Administration
- Adult and Continuing
- Agricultural
- Art
- Bilingual and Multicultural
- Business
- Community College
- Curriculum and Instruction
- Early Childhood
- Elementary
- Finance
- Guidance and Counseling
- Health
- Higher
- History of
- Home Economics
- Industrial
- Language and Literature
- Mathematics
- Music
- Philosophy of
- Physical

LINGUISTICS	
Language	
General	0679
Ancient	0289
Linguistics	0290
Modern	0291
Literature	
General	0401
Classical	0294
Comparative	0295
Medieval	0297
Modern	0298
African	0316
American	0591
Asian	0305
Canadian (English)	0352
Canadian (French)	0355
English	0593
Germanic	0311
Latin American	0312
Middle Eastern	0315
Romance	0313
Slavic and East European	0314

Philosophy	0422
Religion	
General	0318
Biblical Studies	0321
Clergy	0319
History of	0320
Philosophy of	0322
Theology	0469

SOCIAL SCIENCES	
American Studies	0323
Anthropology	
Archaeology	0324
Cultural	0326
Physical	0327
Business Administration	
General	0310
Accounting	0272
Banking	0770
Management	0454
Marketing	0338
Canadian Studies	0385
Economics	
General	0501
Agricultural	0503
Commerce Business	0505
Finance	0508
History	0509
Labor	0510
Theor.	0511
Folklore	0358
Geography	0366
Gerontology	0351
History	
General	0578

Ancient	0579
Medieval	0581
Modern	0582
Black	0328
African	0331
Asia, Australia and Oceania	0332
Canadian	0334
European	0335
Latin American	0336
Middle Eastern	0333
United States	0337
History of Science	0585
Law	0398
Political Science	
General	0615
International Law and	
Relations	0616
Public Administration	0617
Recreation	0814
Social Work	0452
Sociology	
General	0626
Criminology and Penology	0627
Demography	0938
Ethnic and Racial Studies	0631
Individual and Family	
Studies	0628
Industrial and Labor	
Relations	0629
Public and Social Welfare	0630
Social Structure and	
Development	0700
Theory and Methods	0344
Transportation	0709
Urban and Regional Planning	0999
Women's Studies	0453

BIOLOGICAL SCIENCES

Agriculture
General
Agronomy
Animal Culture and Nutrition
Animal Pathology
Food Science and Technology
Forestry and Wildlife
Plant Culture
Plant Pathology
Plant Physiology
Range Management
Wood Technology

Biology	0.24
General	0.306
Anatomy	0.287
Biosciatistics	0.308
Botany	0.309
Cell	0.379
Ecology	0.329
Entomology	0.353
Genetics	0.469
Limnology	0.523
Microbiology	0.410
Molecular	0.307
Neuroscience	0.317
Chemography	0.416
Physiology	0.433
Radiation	0.821
Veterinary Science	0.778
Zoology	0.472
Biophysics	
General	0.786
Medical	0.260

Geodesy	0370
Geology	0372
Geophysics	0373
Hydrology	0388
Mineralogy	0411
Paleobotany	0345
Paleoecology	0426
Paleontology	0418
Paleozoology	0985
Rainology	0427
Physical Geography	0368
Physical Oceanography	0415

HEALTH AND ENVIRONMENTAL SCIENCES

Environmental Sciences	0768
Health Sciences	
General	0566
Audiology	0300
Chemotherapy	0992
Dentistry	0567
Education	0350
Hospital Management	0769
Human Development	0758
Immunology	0982
Medicine and Surgery	0564
Mental Health	0347
Nursing	0569
Nutrition	0570
Obstetrics and Gynecology	0380
Occupational Health and Therapy	0354
Ophthalmology	0381
Pathology	0571
Pharmacology	0419
Pharmacy	0572
Physical Therapy	0382
Public Health	0573
Radiology	0574
Recreation	0575

Speech Pathology	0460
Toxicology	0383
Home Economics	0386

PHYSICAL SCIENCES

Pure Sciences

Chemistry	
General	0485
Agricultural	0749
Analytical	0486
Biochemistry	0487
Inorganic	0488
Nuclear	0738
Organic	0490
Pharmaceutical	0491
Physical	0494
Polymer	0495
Radiation	0754
Mathematics	0405
Physics	
General	0605
Acoustics	0986
Astronomy and Astrophysics	0606
Atmospheric Science	0608
Atomic	0748
Electronics and Electricity	0607
Elementary Particles and High Energy	0798
Fluid and Plasma	0759
Molecular	0609
Nuclear	0610
Optics	0752
Radiation	0756
Solid State	0611
Statistics	0461

Applied Sciences	
Applied Mechanics	0346
Computer Science	0984

Engineering	
General	0537
Aerospace	0538
Agricultural	0539
Automotive	0540
Biomedical	0541
Chemical	0542
Civil	0543
Electronics and Electrical	0544
Heat and Thermodynamics	0348
Hydraulic	0545
Industrial	0546
Marine	0547
Materials Science	0794
Mechanical	0548
Metallurgy	0743
Mining	0551
Nuclear	0552
Packaging	0549
Petroleum	0765
Sanitary and Municipal	0554
System Science	0790
Geotechnology	0428
Operations Research	0796
Plastics Technology	0795
Textile Technology	0994

PSYCHOLOGY

General	0621
Behavioral	0384
Clinical	0622
Developmental	0620
Experimental	0623
Industrial	0624
Personality	0625
Physiological	0939
Psychobiology	0349
Psychometrics	0632
Social	0451

The undersigned recommended to the Faculty of Graduate Studies
and Research acceptance of the thesis:

“FEATURE INTERACTION IN OBJECT-ORIENTED LANGUAGES”

submitted by Danny Epstein, B.C.S.
in partial fulfillment of the requirements for
the degree of Master of Computer Science.

Thesis Supervisor

Director, School of Computer Science

Carleton University

May 10, 1993

ABSTRACT

This thesis analyzes advanced features of object-oriented languages. We enumerate both common and advanced features and discuss the advantages and disadvantages of each. The various choices a language designer must make, once he has decided to incorporate a particular feature into his language design, are discussed. Several new alternatives are suggested. The interactions between these features are analyzed. We choose a set of mutually compatible features for TMPL, an experimental typed multiply polymorphic language. On implementing this language, two unforeseen feature conflicts are recognized. Multiple polymorphism is reconciled with method aliasing by using type inference to determine which method name is legal. However, separate specification and implementation hierarchies cannot be reconciled with multiple polymorphism and typing. We conclude with suggestions for designers of future object-oriented languages.

ACKNOWLEDGMENTS

I would like to thank Mark van Gulik and my brother Peter Epstein for many helpful discussions and comments. Peter and Alan Knight provided many helpful hints on using Microsoft Word to produce theses. They also participated in many email discussions with Carl Watts and myself. A particularly fruitful discussion at OOPSLA '92 included Alan, Carl, and Jean-François Cloutier.

I would also like to thank my supervisor Wilf LaLonde for introducing me to object-oriented programming and giving me a lot to think about. Proofreading by Wilf, Peter, and Anne Lise Hassenklover was much appreciated. Anne Lise's help with C++ was also appreciated.

I would like to thank Mark Wineberg for several helpful comments. Thanks for the 600 dpi output go to Eric Arseneau. Lastly, I would like to thank Doug Macklem and my family for their encouragement, patience, and support.

TABLE OF CONTENTS

Acceptance Sheet.....	ii
Abstract.....	iii
Acknowledgments	iv
Table of Contents	v
List of Tables	vii
List of Figures.....	viii
 CHAPTER 1. INTRODUCTION.....	 1
 CHAPTER 2. FEATURES.....	 4
2.1 Common Features	4
2.2 Advanced Features	17
2.3 Summary	41
 CHAPTER 3. OBJECT-ORIENTED LANGUAGES.....	 46
3.1 Smalltalk.....	46
3.2 Typed Smalltalk.....	50
3.3 SELF.....	51
3.4 Trellis	53
3.5 Eiffel	55
3.6 C++	57
3.7 Common Lisp Object System (CLOS).....	59
3.8 Dylan.....	60
3.9 POOL-I	61
3.10 Summary	62

CHAPTER 4. TMPL: AN EXPERIMENTAL LANGUAGE	64
4.1 Language Design	64
4.1.1 Information Hiding.....	64
4.1.2 Inheritance.....	65
4.1.3 Programming Environment.....	66
4.1.4 Multiple Inheritance	67
4.1.5 Multiple Polymorphism.....	68
4.1.6 Classes or Prototypes?.....	70
4.1.7 Separating Specification and Implementation	70
4.1.8 Typing	71
4.1.9 Syntax.....	72
4.1.10 Summary	74
4.2 Implementation	75
4.2.1 Parser	75
4.2.2 Method Hierarchies.....	81
4.2.3 Interpreter.....	83
4.2.4 One Hierarchy or Two?.....	86
4.2.5 Static Analysis	87
4.2.6 Method Aliasing	91
4.2.7 Programming Environment.....	93
4.3 Using TMPL	96
4.3.1 Numeric Classes.....	96
4.3.2 Collection Classes	98
4.3.3 Use of Features	103
4.4 Future Extensions	106
CHAPTER 5. CONCLUSIONS	110
INDEX	115
REFERENCES	119

LIST OF TABLES

Table 1.	The instance-class hierarchy of a string.....	10
Table 2.	A 3-way multi-method.....	23
Table 3.	Advanced feature interaction	44
Table 4.	Features supported by object-oriented languages.....	63
Table 5.	TMPL collection hierarchy instance variables.....	103

LIST OF FIGURES

Figure 2.1.	Example of inheritance.....	8
Figure 2.2.	The top of the Smalltalk-80 hierarchies.....	11
Figure 2.3.	The Smalltalk-80 4.1 programming environment on a Macintosh...	14
Figure 2.4.	Linked inspectors in SELF.....	15
Figure 2.5.	Is-a hierarchy for polygons	17
Figure 2.6.	Ambiguity due to multiple inheritance of representation.....	18
Figure 2.7.	Merging multiply inherited representation.....	19
Figure 2.8.	Splitting multiply inherited representation.....	19
Figure 2.9.	Multiple inheritance paths to a single variable.....	19
Figure 2.10.	Prototypes without representation inheritance	22
Figure 2.11.	Using a data parent to inherit representation	22
Figure 2.12.	Dynamic inheritance of polygon traits	24
Figure 2.13.	Multiple implementations of the Point specification.....	34
Figure 2.14.	A valid subtype hierarchy	36
Figure 2.15.	A Tektronix version browser	40
Figure 2.16.	Feature option diagram	43
Figure 4.1.	Inserting into method hierarchy.....	82
Figure 4.2.	The top of the TMPL hierarchies	84
Figure 4.3.	The TMPL browser.....	93
Figure 4.4.	Error list produced by static analysis.....	95
Figure 4.5.	Debugging TMPL code.....	96
Figure 4.6.	The TMPL numeric classes	98
Figure 4.7.	The TMPL collection classes.....	101

CHAPTER 1. INTRODUCTION

Object-oriented programming languages have gained in popularity in recent years. This is partly due to the “popularization” of Smalltalk. Digitalk released versions of Smalltalk/V [Digitalk 89] for several platforms including MS-DOS, 286, Windows, OS/2, and Macintoshes. ParcPlace Systems, the vendor of the original Smalltalk-80 [Goldberg & Robson 83], also released versions of Smalltalk for several platforms including Windows, Macintoshes, and various UNIX machines. More importantly, object-oriented programming has spread to other, more conventional languages. The most popular object-oriented language today is certainly C++. Most conventional languages (compilers) being sold today include some kind of object-oriented extension to the language. With all these so called “object-oriented” languages, you would think that there would be some agreement as to what it means for a language to be object-oriented, but as we will see, this is certainly not the case.

A pragmatic definition of *object-oriented* languages is as follows: an object-oriented language must have the following features:

- encapsulation: objects have a representation that is hidden
- dynamic typing: a variable can store several types of objects so objects have run-time type
- message passing: objects send messages to other objects
- polymorphism: different objects can respond differently to the same message
- inheritance: objects inherit (representation and) behavior from other objects

Actually, message passing and polymorphism are not both needed. We will discuss this later. Other features that are commonly provided with object-oriented languages but are not fundamental include:

- classes: objects are instances of a class which defines its representation and behavior
- garbage collection: objects are automatically deallocated when no longer referenced
- programming environment: for editing, executing, and debugging code

Examples of object-oriented languages are Smalltalk and C++.

A language is said to be *object-based* rather than object-oriented if it is missing some of the features required to be object-oriented but still has encapsulation. Examples of object-based languages are Ada [Ichbiad et al 83], Modula-2 [Wirth 88], and CLU.

In the years since Smalltalk was introduced, several new languages have appeared. Many of these added advanced features. Of particular interest are SELF and Trellis.

It is our thesis that few object-oriented language features are orthogonal and that a better understanding of the interactions between language features is needed before designing a new generation of object-oriented languages. To support this claim, we will review the major features supported by object-oriented systems, in addition to investigating some of the more advanced features. We will highlight the advantages and disadvantages of each.

A language designer cannot choose features arbitrarily. Features interact in complex ways. We will analyze these interactions to determine which features are compatible. We will then choose a set of mutually compatible features which work well together for an experimental language called TMPL (pronounced "temple"). Our design decisions will be based on our analysis of each feature and their interactions. By implementing a prototype of the language we have designed, we gain further insights into the chosen features and their interactions.

We conclude by making several suggestions for future language designers, as well as those considering extensions to existing object-oriented languages to incorporate new features.

We attempt to cover as many object-oriented languages as possible, but there will always be some left out. Although many of today's object-oriented languages can trace their heritage back to Simula 67, we do not discuss this language. Many conventional languages have spawned object-oriented versions or extensions in recent years. We cover only some of these. C++ is discussed in detail but Objective C is not. In the Lisp world, we cover only CLOS and Dylan. We do not discuss object-based languages any further in this thesis. Concurrent and distributed languages are not discussed except for POOL-F which is of interest for other reasons. Object-oriented databases are not discussed.

We divide the features of object-oriented languages into two categories. Those features which are common to most object-oriented languages, we refer to as common features. These include both the fundamental and optional features listed above. Features which are in only one or two languages, we refer to as advanced features. Many of these are extensions to common features. In some cases, these features are quite common outside the object-oriented domain but have only recently been introduced in object-oriented languages.

In chapter 2, we discuss both common and advanced features. We discuss the features in order so that each can be understood in the context of the previous ones. As a starting point for discussion, however, we assume knowledge of Smalltalk. Each feature is defined, along with any related terms. Languages which support each feature are listed. We discuss the advantages and disadvantages of each feature. In many cases, different languages choose different options when including a particular feature. When there are several alternatives, we discuss the advantages and disadvantages of each. We also consider several new options which have not appeared in existing languages. Chapter 2 ends with a discussion of how these features interact with one another.

Chapter 3 takes a different approach. Each object-oriented language is discussed briefly. We summarize the features included in each language, pointing out the choices made when there are several alternatives. Chapter 3 ends with a feature-by-feature comparison of the languages.

Chapter 4 introduces TMPL, our experimental language. We decide which features to include in our design. When there are several options, we choose one. A syntax is designed and formalized into a grammar. We then describe the implementation, discussing how each feature was implemented. There were several difficulties encountered along the way. These are described along with their solutions. We go through the implementation of numeric and collection classes within TMPL to see how the features are actually used. Lastly, we describe some possible extensions to the language.

Finally, we draw our conclusions in chapter 5.

CHAPTER 2. FEATURES

2.1 COMMON FEATURES

2.1.1 ENCAPSULATION

Encapsulation is a feature of all object-oriented and object-based languages. It is the ability to group data and the code that manipulates it into an *object*. For example, a point object would contain both x and y components, as well as methods such as `x +`, `x -`, and `distanceTo`. In object-based languages, the group is known as a *module* or *package* rather than an object. Some of the data and/or code can be declared *public*, meaning that it can be accessed from outside the object (or module). Data and code which is declared *private*, on the other hand, is only accessible from within the object. This is known as *information hiding*. In other words, encapsulation puts data and code into a container and information hiding makes (parts of) that container opaque. Encapsulation does not include information hiding but we will discuss both features in this section since they are related.

By declaring some of the data and/or code as public, we are implicitly defining an interface to the object. This is known as a *specification*. The boundary between users and implementors of an object can act as a contract. An object-oriented design will produce specifications for objects indicating which public operations they should provide. Information hiding hides everything but these public operations. This implies that bugs can be located by determining whether the problem lies with the use of an operation or with its implementation. This “divide and conquer” technique is essential to debugging object-oriented and object-based programs.

Different object-oriented and object-based languages provide different information hiding capabilities. They also use different terms for their data and behavior. (See the sections on each language in chapter 3 for details.) For example, in Smalltalk all *instance variables* (data) are private. Instance variables can never be accessed by users of the object unless there is a method for this purpose. Smalltalk does not support private *methods* (behavior) although there are conventions for their use. SELF supports both public and private *slots*. There are two types of slots: *data slots* and *method slots* (behavior). Private slots are accessible from

within the object and from within any object that inherits from it. Trellis provides *subtree visible fields* (data) and *operations* (behavior) to support this type of visibility as well as truly private visibility which does not allow access from subtypes. A basic requirement of information hiding in any language is the ability to hide representation.

It is worth noting that the definitions of encapsulation and information hiding given above are by no means universal. Many authors equate the two concepts. For example, David Smith equates “data hiding” with encapsulation, defining them as “data and code that manipulate it are defined together” [Smith 91]. Grady Booch also equates the two concepts. Other authors distinguish between the two concepts as we have here, including Rebecca Wirfs-Brock and Ralph Johnson. Johnson describes encapsulation as “drawing a line around a bunch of stuff and giving it a name” and information hiding as “aspects of a component are not discernible outside, decided explicitly by the programmer” [Johnson 92].

2.1.2 DYNAMIC TYPING

Conventional languages such as C, Pascal, Modula-2, and Ada use *static typing*, also known as manifest typing or strong typing, which requires declared variables to have a fixed type specifiable at compile-time. The values stored in these variables need not include any type information since the variable’s type is fixed. The compiler can find all type errors and it can use the declared type information to optimize the code it produces. Lisp, on the other hand, is an example of a language with dynamic typing, also known as latent typing or weak typing. All variables are untyped but the values stored include a type which can be queried at run-time. Dynamic typing is a feature of all object-oriented languages. Dynamically typed languages are often interpreted. The reason for this is that a compiler would be of little use since it could not detect type errors, nor could it use type information to optimize the code it produces. (We will see some exceptions to this rule later.)

Dynamic typing can be simulated in languages which do not support it by using variant records with variants for each possible type. Case statements are needed every time an object is accessed. To add a new type requires that we change all these case statements. This is very tedious and it violates encapsulation since everyone is affected by such a local change. As we

will see below, polymorphism, together with dynamic typing, eliminate the need for these case statements in object-oriented languages.

Some languages, such as Trellis, specify types for variables at compile-time but allow the value's type to be any subtype of the specified type. This is still dynamic typing because objects have types at run-time. It is just as powerful because the declared type of a variable can be the most general type (usually called *Object*) allowing any object to be stored in it. It provides all the advantages of static typing—documentation of variable types, compiler optimization, and compile-time type checking—as well as all the advantages of dynamic typing—rapid prototyping and no case statements violating encapsulation.

2.1.3 MESSAGE PASSING

Message passing is a feature of most object-oriented languages. Procedural and functional languages are based on the idea of calling a procedure or function with some data. In the message passing paradigm, we think instead of sending a *message* to an object. The message may include other objects as arguments. A *method* is the implementation of a message for a particular object. Different objects can respond to the same message in different ways. This is *polymorphism* which will be discussed in the next section. In some object-oriented languages, methods always return something (although it can be ignored by the message sender). In others, methods can be declared as returning nothing; eg., C++'s *void*.

A procedure refers to its arguments only. A method refers to its receiver as well, usually denoted *self*, or *me*. Consider the example of a method to add two points together. As a function, this would look like:

```
(define (+ a b)
  (make-point (+ (x a) (x b)) (+ (y a) (y b))))
```

As a method, this would look like:

Point methods:

```
+ b
^ Point x: (self x + b x) y: (self y + b y)
```

Notice that we refer to the receiver as *self*.

2.1.4 POLYMORPHISM

Polymorphism is a feature of all object-oriented languages and some object-based languages. A language has this feature if you can define more than one method or procedure with the same name. Taken literally, polymorphism means “multiple meanings”. There must be some mechanism for choosing which method or procedure to invoke. Usually, this *method lookup* mechanism resolves the search using only the class of the receiving object, as we will see in the next section, but more general mechanisms can be used. Message passing implies polymorphism since different objects can respond to the same message in different ways; ie., using different methods. For this reason, some prefer to leave message passing out of the definition of object-oriented programming languages. David Smith argues along these lines in his book [Smith 91]:

“Casual definitions of OOP often include the idea of message passing as fundamental. Yet it isn’t. It is a result of the other ideas. Given polymorphism and a hierarchy it is necessary to find the method to run. Clearly two steps are necessary: find where to start looking, and then following some algorithm, look until the method is found (or the search fails). Thus message sending is a fancy name for binding methods to their calls.”

If we want to call languages which support multiple polymorphism¹ “object-oriented”, we must not include message passing in the definition of object-oriented programming languages.

Some procedural languages support polymorphism, although it is rarely fully supported. For example, Ada and CLU support polymorphism for built-in operators only and their precedence cannot be changed. This means that there can be more than one function called +, so long as the compiler can determine which one to use by the types of the arguments or the type of the result.

2.1.5 INHERITANCE

Inheritance is a feature of all object-oriented languages. The idea of inheritance is to be able to specify a class by describing how it differs from some other class. (Classes will be discussed

¹Multiple polymorphism is a generalization of message passing, as we’ll see later.

in the next section.) This always includes behavior and usually includes representation. For example (see Fig. 2.1), a **Student** is like a **Person** except that a student has a list of degrees as well as a name and address. **Student** also defines additional behavior beyond the basic behavior inherited from **Person**. A **Student** would include his degrees when asked to print his name; eg., John Doe, B.A.. On the other hand, the representation of **Employee** includes a title as well as a name and an address inherited from **Person**. An **Employee** would include his title when asked to print his name; eg., John Doe, Director F&D.

Figure 2.1. Example of inheritance.

The class used as a basis for comparison is called the *superclass* (or *base class* in C++). As we shall see later, some languages support multiple inheritance, in which case there can be more than one superclass. The inheritance hierarchy forms a tree (or forest) in a single inheritance system. There is usually a root class called **Object** and all other classes are *subclasses* (*derived classes* in C++) of this root, either directly or indirectly. To determine the representation and behavior of a class, the inheritance hierarchy must be traversed. This is precisely what the method lookup routine does to determine which method to execute when a message is sent to an object. To describe how a class differs from its superclass, we must indicate which instance variables and methods need to be added. We may also be able to indicate that an instance variable or method needs to be changed; eg., the print method in our example above. This is called an *override* (or shadow). Some languages support subtractive inheritance, in which case we can indicate that we do not want to inherit an instance variable or method at all. This is called an *exclude*. Smalltalk does not support overriding or excluding instance variables and it supports overriding but not excluding methods. In some cases, we want to inherit an instance variable or method but we want to call it by a different name: an *alias*. For example, we may want to rename an instance variable locally because we use it

differently than the superclass did. Only basic inheritance with overrides is commonly available. A few languages support excludes and/or aliases as well.

If representation is not inherited, then behavior which accesses this representation will fail. This behavior is known as *primitive* behavior. If a subclass has different representation than its superclass, it will have to provide its own primitive methods to access this representation. Consider a language with optional representation inheritance. Assume that the programmer has independent control over inheritance of each instance variable. Some primitive methods (those methods which access instance variables which are not inherited) should not be inherited. The primitive methods in the superclass must be re-implemented in the subclass. For example, a subclass of **Point** could choose not to inherit its *x* and *y* coordinates, opting instead for a polar coordinate representation. The subclass would then have to re-implement all **Point** methods which directly access the *x* and *y* instance variables since they would not be inherited.

2.1.6 CLASSES

Most object-oriented languages are class-based. In class-based languages, all objects are *instances* of some *class* which defines its representation and behavior. This reduces the size of instances because they can simply refer to their classes for detailed information. For example, instance variables can be stored without their names since the names can be found through the class. A class functions as a template for instantiating new instances. A class can be thought of as a set of all possible instances. For example, the **Integer** class represents the set of all possible integers. Being an element of a set then corresponds to being an instance of a class. However, just because an object behaves like an integer does not make it an instance of the **Integer** class. Some definitions of object-oriented programming languages include classes as a required feature [Wegner 87, Stroustrup 87]. David Smith argues against it:

“Other casual (and not so casual) definitions of OOP include the idea of class as fundamental.... Hierarchies [and therefore inheritance] are fundamental; classes are just one way to implement hierarchies.” [Smith 91]

Not all classes actually have instances. An *abstract class* exists solely to contain common representation and behavior of its subclasses. For example, the class **Magnitude** holds methods common to **Date** and **Number** (another abstract class). Abstract classes often require some

specific behavior of their subclasses. For example, `Magnitude` implements `min:` in terms of `<`. To ensure that subclasses of `Magnitude` implement `<`, it is implemented as an error (subclass responsibility) in `Magnitude`.

Some object-oriented languages treat classes as second-class objects. In these languages, classes are not really objects since they can only be used to instantiate instances. In languages which treat classes as first-class objects, classes are objects like any other. They can be stored in variables, passed as parameters, sent messages, etc. This creates a circular definition which must be resolved somehow. If classes are objects, they must be an instance of some class. In many object-oriented languages, classes are instances of corresponding *metaclasses*. Metaclasses define representation and behavior for classes. But even metaclasses must be an instance of some class. We need some way to terminate the instance-class hierarchy.

Table 1. The instance-class hierarchy of a string.

Type of Object	Smalltalk Expression
object	'hello'
class	<code>String</code>
metaclass	<code>String class</code>
class	<code>Metaclass</code>
metaclass	<code>Metaclass class</code>
class	<code>Metaclass</code>
metaclass	<code>Metaclass class</code>
...	...
...	...

In Smalltalk-80, if we ask the string `'hello'` what its class is, it will be the class `String`¹ (see Table 1). If we ask the class `String` what its class is, it will be the corresponding metaclass `String class`. This metaclass is an instance of the class `Metaclass` which is an instance of the metaclass `Metaclass class`. At this point the instance-class hierarchy loops back on itself. The metaclass `Metaclass class` is an instance of the class `Metaclass`. The superclass hierarchy must also be terminated somehow. In this case, a simpler technique is used. The superclass of the

¹The text shown in a sans-serif font is the `printString` of the object. Classes print as their name. Metaclasses print as their name followed by the word `class`. In both cases, this text would evaluate to the object.

class `Object` is simply `nil`. Since `nil` is not a class, this unambiguously indicates that `Object` is the root of the class hierarchy.

Figure 2.2. The top of the Smalltalk-80 hierarchies.

Fig. 2.2 [Goldberg & Robson 89] shows the top of the Smalltalk-80 instance-class and inheritance hierarchies. Each class has its corresponding metaclass which it is an instance of. These metaclasses form a parallel hierarchy. The superclass of the metaclass `Object class` is the class `Class` so that metaclasses inherit their default behavior from the class `Class`. The class `Class` inherits from the class `ClassDescription` which inherits from the class `Behavior`. This is simply factoring out behavior to improve code sharing. We see that the class `Metaclass` is a sibling of the class `Class` rather than a subclass. This was done to avoid the large number of overrides that would have been required. As we noted above, all metaclasses, including the metaclass `Metaclass class`, are instances of the class `Metaclass`.

These organizational problems can make it difficult to follow simple behavior. For example, try to find the new method for `Object` in Smalltalk-80. It is an *instance* method in the class `Behavior`. However, the existence of metaclasses can safely be ignored by newcomers to the

language. In fact, the use of metaclasses is fairly rare, even for experienced Smalltalk programmers. They are primarily used for *meta-programming*: changing the language from within itself.

One problem with class-based systems appears when classes act as managers for their instances. Eventually the programmer may want to separate out the class behavior into a manager class, because more than one manager is needed. In order to avoid such potential reorganizations, we can eliminate all class behavior except that required to create instances. There is another reason to do this. When instance initialization becomes complex, it is often done by sending messages to an instance created with the new message, rather than sending a message to the class with all the arguments. For example:

```
I listPane listItem I
listPane := ListPane new.
listPane addColumn: 'Name' width: 3.
listPane addColumn: 'Grade' width: 1.
listItem := ListItem new.
listItem atColumn: 'Name' put: 'John Doe'.
listItem atColumn: 'Grade' put: 'A+'.
listPane addItem: listItem.
```

This allows several messages to be sent to initialize the object. Normal methods should then verify that the object has been properly initialized. Often there are several stages to the initialization process, as in our example above [Epstein 93]. The problem with this approach is that normal methods will often forget to check for proper initialization or will not want to take the time necessary to verify that initialization is complete. Also, it isn't as convenient in the typical case where a small number of arguments are passed in to the instance creation method.

2.1.7 GARBAGE COLLECTION

Many object-oriented languages (and some conventional languages such as Lisp) have automatic garbage collection. This means that objects which are no longer referenced by other objects are automatically deallocated and the space used by them reclaimed. Explicit deallocation is no longer necessary, eliminating the need to worry about dangling pointers. In languages without garbage collection, explicit deallocation causes many problems, the most important one being violations of encapsulation. Knowing who is pointing at an object, or

even how many objects are pointing at it, is outside the object's reasonable domain of knowledge. Ada and (most implementations of) C++, require the programmer to spend a great deal of time worrying about deallocation. Conventions are often implemented to do manual reference counting (a simple garbage collection technique we will discuss below).

The first garbage collectors were developed for Lisp systems in 1960. McCarthy used the *mark and sweep* algorithm [McCarthy 60] which traverses the entire graph of referenced objects marking them, after which it compacts the marked objects. Collins used *reference counting* [Collins 60]. This technique requires an additional field in each object to keep track of the number of references to the object. When the reference count reaches zero, the object can be deallocated. Unlike mark and sweep, reference counting can be done *incrementally*. Incremental garbage collection is done concurrently with execution of the application—it occurs in the background. The problem with reference counting is that it does not collect unreferenced circular structures so mark and sweep is often used in conjunction with it to occasionally deallocate circular structures (and the objects being pointed to by them).

More recent garbage collectors have tried to reduce the time spent garbage collecting. Baker's half space algorithm [Baker 78] incrementally copies objects from one half space to another. The problem with this algorithm is that half of memory is wasted. Ungar's generation scavenger [Ungar 84] improves performance and reduces memory requirements by taking advantage of the fact that most objects are relatively short lived. Objects are stored in one of several spaces (or generations) depending on their age. Newer spaces are garbage collected more frequently and occupy substantially smaller fractions of the total space. Ungar manages to reduce garbage collection to 3% of total processing time.

Current research in garbage collection is concentrating on virtual memory, persistent object systems, and distributed processing. Virtual memory can provide persistent object storage conveniently. However, memory systems based on garbage collection exhibit little locality of reference because objects are constantly being moved around by the garbage collector. This makes them inefficient on platforms with virtual memory. Related objects can be grouped within pages on disk to reduce page faults [Williams et al 87]. Creepy [Nguyen 89, Stringham 89] is an incremental secondary storage garbage collector that works in conjunction with a

virtual memory system. It garbage collects blocks before storing them on disk. It also runs in an off-line mode to remove circular structures that span more than one disk block. Garbage collection on distributed processors is also being investigated [Eckhart & LeBlanc 87, Duimovich 88].

2.1.8 PROGRAMMING ENVIRONMENT

Many object-oriented languages (and some conventional languages) include extensive programming environments, often with graphical user interfaces (GUIs). These environments may include tools for browsing existing code, writing new code, running code, debugging code, and inspecting objects. There may also be tools for tracking source code modifications in a multi-programmer environment and for analyzing space and/or time efficiency.

Figure 2.3. The Smalltalk-80 4.1 programming environment on a Macintosh.

Smalltalk-80 includes several kinds of browsers, a debugger, and several inspectors (see Fig. 2.3). ObjectKit for ParcPlace Smalltalk-80 [ParcPlace] adds more browsers as well as profilers for analyzing space and time efficiency, and a class reporter for finding likely bugs (much like lint for C) and computing various code metrics. Another useful Smalltalk feature is its ability to recover changes lost when it crashes. All changes are written out to a log file

which can be scanned for changes since the last save. These changes can be browsed in a change list view.

Trellis includes a complete programming environment [O'Brien et al 87]. In order to improve performance, a database is used to maintain cross-reference information. This allows the programmer to browse the call graph more quickly than is possible in Smalltalk. Another advantage of Trellis programming environment over Smalltalk's is a tool called the *grass catcher*. This tool keeps track of inconsistencies, such as references to undeclared variables that have yet to be dealt with. The user can select an inconsistency and it will show the code at fault. Inconsistencies are automatically removed once they are corrected. The idea of this tool is that it allows the programmer to make changes which produce errors so long as he corrects them eventually. For example, when moving a field declaration to a supertype, the field would be removed from the subtype and then added to the supertype. During this operation, references to the variable are in error so they would appear in the grass catcher. Smalltalk has a similar, although much weaker, mechanism for handling such undeclared variables: the Undeclared dictionary.

Figure 2.4. Linked inspectors in SELF.

Although SELF's programming environment is far from fully functional, it demonstrates several unique ideas. The only tool provided is an inspector, but since code is stored in slots just like data, these inspectors can be used to browse code. SELF's inspectors are linked to the inspector that spawned them by a line indicating which slot was spawned (see Fig. 2.4). There can only be a single inspector on a given object. This allows the user to treat an inspector on an object as if it were the object itself.

There are tools provided in many programming environments to help manage changes. These tools often keep track of versions and let the programmer compare two versions for differences. Some tools support using a database for controlling access to code being edited by a group of programmers. These tools are considered an advanced feature. They are discussed in section 2.2.10.

2.2. ADVANCED FEATURES

2.2.1 MULTIPLE INHERITANCE

In languages with multiple inheritance, an object or class specifies several other objects or classes from which it inherits. The inheritance hierarchy¹ normally forms a directed acyclic graph (DAG). In some languages, the graph may contain cycles, in which case the loops are detected and avoided by the method lookup routine.

Multiple inheritance is useful for representing is-a relationships that are DAGs. This occurs frequently in computational geometry; eg., consider various types of polygons. A simple polygon is a polygon with no holes whose edges don't intersect except for adjacent edges. Star-shaped polygons are simple polygons that have some point inside them which can see the entire polygon. Monotone polygons consist of two chains which are monotone with respect to (w.r.t.) the same line, where a chain is a list of points connected by line segments. A chain is monotone w.r.t. a line if it is a function if we interpret the line as the horizontal axis. A convex polygon contains all line segments between end-points within the polygon. A convex polygon is always both star-shaped and monotone, but not all polygons which are both star-shaped and monotone are convex. Thus, the is-a hierarchy is the DAG shown in Fig. 2.5.

Figure 2.5. Is-a hierarchy for polygons.

¹Note that we do not define a “hierarchy” as a tree. It may be a DAG or, for that matter, any directed graph.

In a language that does not support multiple inheritance, we would use one of several work-arounds. We could make star-shaped and monotone polygons inherit from star-shaped or monotone polygons and duplicate the methods defined in the other one. There is a way to avoid such duplication. All polygons could be represented by the same class (in a class-based system), where each instance of the class includes a flag indicating which type of polygon it is. Methods can then test this flag to determine the appropriate behavior. This is slow and tedious, but it does work. When there are several independent attributes which affect the behavior, this is often the most appropriate solution.

There are two common ways to define multiple inheritance. One way is to provide an ordered list of parents to inherit from. The method lookup searches the list in order, stopping as soon as it finds the method. The other common technique is to provide an unordered set of parents to inherit from. The method lookup searches all the parents at once. If the method is found in more than one parent, an ambiguity error is reported. SELF uses a combination of these two techniques: the programmer provides a list of sets of parents by marking each parent with a priority. The method lookup searches through the sets in priority order, stopping as soon as it finds a unique method. In each set, all parents are searched simultaneously. If a matching method is found in more than one parent in the set, an ambiguity error is reported.

Figure 2.6. Ambiguity due to multiple inheritance of representation.

Multiple inheritance of representation presents some difficulties. Consider the scenario shown in Fig. 2.6. Two instance variables called `varB` are inherited, one from `ClassA` and one from `ClassB`. In some cases it is appropriate to merge these two instance variables into a single one. For example, in Fig. 2.7, we want to merge the two inherited positions into a single variable in `Box`. In others, the two variables should be kept separate [Ungar et al 91]. For example, in Fig. 2.8, we want to keep both origins and corners as independent variables in `LinearFrame`. In

this case, we would need some way to refer to these variables unambiguously. Actually, it may be appropriate to merge two instance variables with different names. Consider the example shown in Fig. 2.7. If *Rectangle*'s position was stored in a variable called *origin*, it would still be appropriate to merge it with *DisplayableObject*'s position in *Box*.

Figure 2.7. Merging multiple inherited representation.

Figure 2.8. Splitting multiple inherited representation.

Figure 2.9. Multiple inheritance paths to a single variable.

The scenario shown in Fig. 2.9 is quite different. `ClassSub` inherits `varA` defined in `ClassSuper` along two different paths. In this case, the desired result is always a single copy of the variable.

Given all these difficulties, it may be better to simply not inherit representation at all. As we will see, in SELF, representation is not normally inherited. We will also show how method aliasing can be used to solve these problems.

Multiple inheritance is often used to build classes by inheriting from groups of methods called *mix-ins*. Each of these mix-ins defines a protocol. For example, the `String` class might inherit from a `Comparison` mix-in. This mix-in would implement all the comparison operators (`<`, `>`, `<=`, `>=`, `=`, `~=`) in terms of the primitive¹ `<` and `=` operators. Mix-ins allow many combinations of a set of features to be defined without duplicating code.

2.2.2 PROTOTYPES

As we saw in section 2.1.6, classes require metaclasses to provide class representation and behavior. This results in a complex hierarchy (see Fig. 2.2). Class-based languages have a difficult time representing *one-of-a-kind objects*, such as `true`, `false`, `nil`, and the empty list. Since these objects have methods, they must have classes as well as the single instance. For example, in Smalltalk, `true` is an instance of the class `True`. This clutters the name space and causes confusion. *Light-weight classes* are classes that are not in the global name space. If the class `True` was light-weight, it would no longer be accessible except by asking `true` for its class. Light-weight classes reduce name space clutter but they can cause confusion themselves. How do we name light-weight classes?

There is an alternative to class-based languages: prototype- or exemplar-based languages [Lieberman 86, LaLonde 86]. One example is SELF, a prototype-based language currently being developed by Stanford University and Sun Microsystems [Ungar & Smith 87]. In these languages, objects describe their own behavior. A new object is created by *cloning* an existing

¹The word “primitive”, as it is being used here, means that the method accesses representation directly

prototype or *exemplar*. For example, to create a new list, we would clone the prototypical list. The prototype implicitly describes what all the instances look like by serving as a model for their creation. An object can include a pointer to a parent object from which it inherits or *delegates*. Rather than superclasses and subclasses, we have *delegates* and *delegators*. Thus we have a hierarchy of instances rather than classes. If multiple inheritance is supported, there may be several parents. Just as for class-based languages, the parents may be ordered (a list), unordered (a set), or partially ordered (a list of sets). In the latter two cases, ambiguities may arise which must be dealt with by the programmer.

Prototype-based languages are inherently simpler than class-based ones because there are fewer concepts to learn. However, there are problems with prototype-based languages. One problem is that if each object contains its own behavior, then objects take up a great deal of space. This problem can be eliminated by moving the methods into a parent of the prototype, called a *traits* object [Ungar & Smith 87]. This object is then pointed to by the prototype and all its clones. Note that traits are not the same as classes: they describe behavior but not representation. Objects that are never cloned (one-of-a-kind objects) can contain their methods directly. Another problem with prototype-based languages is that the prototype can be accidentally modified if a programmer forgets to clone it. This can be avoided by either protecting prototypes or by making the prototype accessor automatically clone the prototype. Neither solution is particularly elegant: protection complicates the language and automatic cloning doesn't let the programmer pass arguments to the clone method. Another problem with prototype-based languages is that you can't ask an object what class it is an instance of. Instead, you have to ask whether the object understands certain messages or protocols. If you want to know whether an object is a string, you must first ask whether it has a parent slot, and then ask whether this parent slot contains the string traits. Compare this to class-based languages in which you could simply ask whether the object's class was the string class (`isMemberOf: String`) or whether the object inherits from the string class (`isKindOf: String`). On the other hand, asking whether an object is an instance of the class string doesn't allow for the possibility that objects other than strings may work perfectly well, so long as they support the appropriate protocol.

Figure 2.10. Prototypes without representation inheritance.

Since we separate out behavior into traits objects, we effectively eliminate representation inheritance (see Fig. 2.10). This is not as big a problem as one might think. Most objects have relatively little representation so duplicating it is not unreasonable. When this is not reasonable, we can use *data parents* to provide representation inheritance in a prototype-based language. We use multiple inheritance to make the sub-object's prototype inherit from both the sub-object's traits and the super-object's prototype (see Fig. 2.11).

Figure 2.11. Using a data parent to inherit representation.

Now `prototype b` doesn't need its own copy of `prototype a`'s slots. When `prototype b` is copied, it will have to copy its data parent as well, but this is a minor problem. A more difficult problem arises when you try to assign an inheritance priority to the data parent. If it is given a low priority then it will be searched after the normal traits parent. Because method lookup is normally done in depth-first order, `traits a` will be searched before `prototype a`. This means that although `prototype a` could override `traits a`, `prototype b` will not inherit the override. If the data parent is given a high priority then it will be searched before the normal parent, causing `traits a` to be searched before `traits b`. If the data parent is given the same priority as the normal parent, then all three ancestors will be searched simultaneously. This means that neither `traits b` nor `prototype a` can override `traits a`. All of these problems can be overcome by defining slots in `prototype b` which direct the search appropriately, but this is tedious at best. SELF includes a complex ambiguity resolution

mechanism known as the *sender path tiebreaker* [Chambers et al 91] which is supposed to resolve these difficulties but does not always succeed in doing so. A better solution would be to change method lookup to always search children before their ancestors. The SELF group is considering doing this.

2.2.3 MULTIPLE POLYMORPHISM

Multiple polymorphism is an extension to message passing [Epstein 89]. In message passing, the receiver determines which method is being referred to when a message is sent. The method lookup routine uses the class of the receiver as a starting point. If there is no method with the appropriate name, the superclass is checked. This proceeds until the method to be executed is found, or we reach the top of the class hierarchy. With multiple polymorphism, some or all of the arguments may also be used to determine the method. The most obvious example of where this would be useful is in arithmetic. There could be separate methods (sometimes called *multi-methods*) for integer + integer and integer + float. This could be done using message passing with a technique called *double dispatching*. Integers would have a + method which turns it around and sends the argument `addFromInteger: self`. This would be implemented in both Integer and Float. This technique avoids the inefficiency of explicit type checking but it is messy, particularly when there are many arguments to polymorph on. To implement multiple polymorphism, the method lookup routine must search a DAG of methods for a particular message. A method is a descendent of another method in the DAG if its formal parameters are (non-strict) subclasses of the formal parameters of the other method. The search down the DAG can terminate with more than one method to execute. In this case, the message send is considered ambiguous, just as with multiple inheritance.

An example of multiple polymorphism that is more than two-way and that does not involve arithmetic operators is the `displayOn:at:rule:` message (see Table 2). There should be separate methods for displaying each of the 64 combinations.

Table 2. A 3-way multi-method.

form opaqueForm colorForm opaqueColorForm	displayOn:	form opaqueForm colorForm opaqueColorForm	at: point rule:	copy drawBlack eraseWhite reverse
--	-------------------	--	------------------------	--

The designers of SELF are considering incorporating multiple polymorphism into the language. It is not clear whether all arguments should be used for polymorphism, or only those arguments that are marked as polymorphic. In either case, methods can be “in” several objects at once. For example, the integer + float method would be in traits integer because the receiver is an integer, but it would also be in traits float because the argument is a float. Actually, we no longer refer to a “receiver” because there is no preferred argument position. Instead, we simply refer to the first argument just like any other—by whatever name we gave the first formal parameter.

2.2.4 DYNAMIC INHERITANCE

To explain dynamic inheritance, we will look at implementing a simplified version of the polygon example we discussed in section 2.2.1 in SELF (see Fig. 2.12).

Figure 2.12. Dynamic inheritance of polygon traits.

Each type of polygon in the polygon hierarchy (a DAG) would have a corresponding traits object containing the appropriate methods. These traits objects would inherit from one another appropriately. Rather than defining separate prototypes for each type of polygon, we define a single prototype polygon which can inherit from any of the traits objects. To do this we make this prototype's parent slot be *assignable*. Whenever the vertices of a polygon are changed in any way (added, deleted, or moved), the type of the polygon can change, necessitating an assignment of a different traits object to the parent slot of the object representing the polygon. For efficiency, we can delay the update until a sequence of changes has been made to the polygon. We could define a display method for each type of polygon, each slightly different to indicate the type of polygon to the user. Dragging around a vertex could then change the polygon type in real-time. For efficiency, it may be desirable to determine the areas where the vertex can be moved for each type of polygon when the user first picks up the vertex.

When we want to build a new polygon, we clone the single polygon prototype and then add vertices to the copy. This implies that the prototype should have no vertices, not a very "prototypical" example of a polygon. This is bad enough, but what traits object should the prototype inherit from? We could create a special traits object for empty polygons, but then what traits object do we use when we have added one or two vertices? The solution to all of these problems is to use *object mutation*. We start with a clone of the (empty) polygon prototype. When we add a vertex to it, it becomes a point. If we add another vertex to it, it becomes a line segment. Adding a third vertex turns it into a (convex) polygon. What makes these mutations different from the change that occurs when the type of a polygon changes is that the *representation* changes. A polygon is represented as a (possibly empty) list of vertices. A point is represented as an x and a y coordinate. A line segment is represented as a start point and an end point. To allow the representation to change, we use an assignable data parent rather than an assignable traits parent. The data parent inherits from the appropriate traits parent. To make vertices and line segments be able to act like polygons, we make the vertex and line segment traits objects inherit from the polygon traits object. The vertex and line segment traits objects must include a method to access their vertices as well as methods to add and remove vertices, mutating the object in the process. An added benefit of this scheme is that we can treat points and line segments as polygons. This is very useful for many divide-and-

conquer algorithms in computational geometry which break down polygons into pieces which are not always valid polygons.

Object mutation is simpler using Smalltalk's `become:` because the programmer does not have to add an extra level of indirection himself. However, Smalltalk would have to use `become:` even when the representation doesn't change, making it significantly slower.

Dynamic inheritance can be used to provide context-sensitive behavior. In the real world, objects behave differently in different circumstances. If you ask somebody a question, they take the context in which it was asked into account when answering. It would seem appropriate to carry this over to object-oriented programming. Consider integers, for example. Normally they respond to the `+ aNumber` message by doing simple addition. However, in some contexts, it would be more appropriate to do the addition in a Galois field; i.e., modulo some number. For example, in cryptography, most arithmetic is done modulo a large prime. Division and reciprocation are done using the Extended Euclidean Algorithm. Square roots are very difficult to compute in a Galois field—so difficult, in fact, that squaring can be used as a means of encryption. We could define a new kind of integer, say `moduloInteger`, but it would have to be specialized for each desired modulus. Also, the user would have to explicitly convert between normal integers and modulo integers. Instead, we can use context-sensitive behavior. For example, we could divide 3 by 4 in a modulo 13 field as follows:

```
13 useModuleArithmeticIn: [ 3 / 4 ] returns 4
```

Dynamic inheritance can be used to change the behavior of an object on the fly. By changing parent pointers whenever the context changes, we can make integers do arithmetic and comparison appropriately based on the context. Before we evaluate the `[ 3 / 4 ]` block, we save the current parent pointers and modulus, and then set them appropriately. After we evaluate the block, we restore the old parent pointers and modulus. The only practical problem is that most code assumes integers behave normally, so most code can't be used within the block. Another way context-sensitive behavior could be implemented would be to make the `integer /` method send itself and the argument to one of two objects: `normalInteger` or `moduloArithmetic`. Triple dispatching would be used unless the language supported multiple polymorphism directly.

2.2.5 REPRESENTATION AS BEHAVIOR

In SELF, references to instance variables (data slots) are indistinguishable from message sends. This is also true in Trellis. This means that if an object inherits from something which expects it to have a particular slot, the object can use a method rather than a variable. For example, if we want to intercept all changes to a data slot, we can put the data slot in a parent object and then override its assignment primitive locally to do whatever we want in addition to resending the assignment up to the parent.

To make inheritance more uniform, SELF and Trellis chose to always refer to instance variables through accessor methods. Then representation inheritance and privacy are automatically governed by the same rules as behavior. Trellis takes this a step further. If an instance variable is not accessible because its accessor methods have been overridden by other methods, then space is not actually allocated for that instance variable. Both SELF and Trellis make accessing instance variables through methods convenient by designing the syntax with this in mind. SELF uses an “implicit self” message send:

```
x          => self x
x: 5       => self x: 5
```

Trellis defines syntactic sugar which maps the conventional instance variable accessing syntax to calls to get and set methods:

```
point.x     => get_x (point)
point.x : 5  => put_x (point, 5)
```

Note that just because we make an instance variable public, does not mean that we cannot change the representation. So long as we provide the corresponding accessor methods, our clients will never be the wiser. Also, just because accessor methods are defined automatically, doesn't mean that all instance variables are public. We are given the same visibility control over these methods as any others.

We saw in section 2.1.5, how optional representation inheritance could be implemented by limiting inheritance of methods to those which do not use the missing instance variables. Trellis's ability to override accessor methods to remove inherited representation is superior.

Rather than re-implement the primitive methods, we simply implement a method that takes the place of the missing instance variable.

2.2.6 SEPARATING SPECIFICATION FROM IMPLEMENTATION

An *abstract data type* consists of two parts: the *specification* and the *implementation*. The specification includes a description of the data type and a list of the public messages it understands with descriptions of each. The implementation includes the representation and the actual implementation of all the methods (the code), including any implementation comments. The separation of the two parts insulates users of the data type from changes in its implementation. The specification can be thought of as a contract between the users of the data type and the implementor of the data type. We refer to the specification of a data type as a *type* (the *set type*, for example), and we refer to the implementation of a data type as a *data structure* (the *binary search tree data structure*, for example). We say that a data structure *implements* a type (or *conforms to* a type) if it provides the necessary operations with the appropriate semantics.

We usually think of a single type being implemented by a single data structure, but this is not necessarily the case. For example, the set type could be implemented by several data structures: binary search trees, AVL trees, hash tables, etc. It also works the other way around. Finger trees and splay trees implement several types: destructive lists, sets (including sorted and identity sets), and dictionaries (including sorted and identity dictionaries). The relationship between types and data structures is clearly many-to-many [Epstein 90]. In fact, a particular data structure can implement a particular type in several ways. For example, `OrderedCollection` could be used to implement `Queue` in either direction [van Gulik 92].

There are two reasons why a type should be implemented by several data structures. The first reason is to provide multiple representations, each with its own space/time trade-offs. Our example demonstrates this well. Binary search trees are fast for random insertions. AVL trees are slower but do not require random insertions. Hash tables are fast but they have a bad worst-case order. Which of these data structures you might choose to implement a particular `Set` would depend on the application. The second reason for providing multiple

implementations of a type is to allow some instances of the type to have different behavior (and representation) than other instances [LaLonde 89, LaLonde 86]. For example, an empty list should really have different behavior than a non-empty list. The empty list gives an error for most operations and it doesn't need any representation. By separating out the behavior for empty lists, we avoid testing for the empty case in each method. Another example of different behavior for different instances would be constructors. Each constructor type has different behavior.

Data structures use inheritance for code sharing. This is useful for two reasons: to reuse existing code rather than rewriting it, and to make it easier to modify later. For example, AVL trees could be efficiently described as a change to binary search trees with additional code for re-balancing. Subsequent changes made to binary search trees would then affect both automatically. Multiple inheritance may be needed to maximize code reuse, but it is not essential here. It has been suggested [LaLonde 89, LaLonde 86] that data structures use instance inheritance; i.e., that the data structures be prototypes rather than classes. Prototype-based systems provide more flexibility than class-based systems, and implementations can take advantage of this flexibility without affecting the user's view; i.e., the specification.

Types form a hierarchy based on *substitutability*. A type is a valid subtype of another type if it conforms to *subtype compatibility rules*. These rules ensure that an instance of the subtype can be used wherever an instance of the supertype is expected. We will discuss subtyping in more depth in section 2.2.8. There is no reason why a type should have only a single supertype so the type hierarchy can be a DAG rather than a tree.

It is clear that types and data structures each need their own hierarchies, but how are they integrated? If a data structure provides all the operations specified by a type, we can say that the data structure *implicitly* implements the type. The problem with this technique is that just because operations have the same name does not mean that they do the same thing. Consider Queue and Stack types defined with operations called *add* and *remove*. These would be considered identical. In a system which uses implicit type-data structure mapping, operation names must be carefully chosen to avoid such unwanted merges. The programming environment can help. Also, classes can be given sets of *properties*, arbitrary identifiers

representing semantic information. For a data structure to be matched up with a type, it must have all the properties listed in the type. POOL-1, a language we will see more of later, uses this technique to avoid undesired implicit merges.

The alternative to implicit type-data structure mapping is *explicit* type-data structure mapping. Each data structure includes a list of which types it implements in its definition. As with implicit mapping, operations with the same name are assumed to correspond. We cannot use a method name such as `enqueue` in our type if our data structure uses the name `insert`. This would appear to be a minor problem, but it turns out to be quite serious. What if we are using an existing data structure to implement a new type? We have to use the data structure's name for each operation. This violates the spirit of encapsulation since we are letting the implementation affect the specification. Also, we may use a single data structure to implement several types, each of which uses a different name for the same operation. What we need is an arbitrary mapping between methods in the type and methods in the data structure. In the following section, we look at *encapsulators* which give us this and more.

2.2.7 ENCAPSULATORS

An encapsulator [Pascoe 86] is an object which inherits from a type and which contains a data structure. We refer to encapsulators as *adapters* [Epstein 90] since they adapt data structures to conform to types. They are also known as *wrappers* [Knight 92]. The adapter implements each of the methods specified by the type as a “call through” to the contained data structure. This maps methods of the type to methods of the data structure. The adapter methods can be more than just call-throughs. For example, an adapter which implements the set type in terms of a hash table might implement a method to test for inclusion of a given element as:

SetAsHashTable methods:

Includes: anElement

^ (hashTable at: (hashTable find: anElement)) notNil

Data structures generally provide low level operations which are then used to implement the high level operations required by the type. The adapter can change the way the operation handles exceptions, for example.

Because the data structure is encapsulated within the adapter, we cannot send data structure messages to the adapter without getting an error. We have successfully insulated the users of the abstract data type from the implementor. We can have encapsulators for several different data structures implementing the same type and we can have encapsulators for several different types being implemented by the same data structure.

The biggest problem with encapsulators is that each data structure must be encapsulated. Continuing our above example of sets implemented using a hash table, consider the non-destructive union operation. This operation is supposed to return another set, so it must build an encapsulator around the resulting merged hash table as follows:

SetAsHashTable methods:

union: aSet

^ SetAsHashTable new hashTable: (hashTable merge: aSet hashTable)

This takes both time and space, as well as complicating the encapsulator's methods considerably.

Another alternative for integrating a type and a data structure is to use multiple inheritance. We build an adapter object which inherits from both a type and a data structure. This object implements each of the operations specified in the type by sending data structure messages to itself. There will often be name conflicts between type messages and data structure messages. In this case, a directed resend can be used to specify which parent to use. The data structure is a data parent of the adapter so it must be copied when the adapter is copied. Operations that return data structures must build a new adapter. For example, the *rest* operation on lists would have to return an adapter which inherits from both the linked list and the list type. There are several problems with this alternative. The user can send a data structure message to the adapter and it will be accepted. This can be avoided by declaring all data structure slots as private, but this is not always reasonable since the data structures are often built-in. Protocols cannot be used to group related methods in the adapter because directed resends must refer to a parent of the method holder. The two parents of the adapter must be given the same priority so that data structure methods don't end up calling type methods. This means that all the type's methods must be implemented in the adapter, even those which are implemented by the type.

As we have seen, the relationship between types and data structures is many-to-many. Thus we have names for types, names for data structures, and names for adapters. It would simplify things considerably if we could contain this name-space explosion by merging types and data structures into a single named entity: classes. Assume we have classes which define a specification (by their public interface) and an implementation [Epstein 92]. A class inherits implementation from its superclass and specification(s) from its supertype(s). In other words, there are still separate hierarchies for specification and implementation but they share common elements: classes. This works well for the typical case, where there is a one-to-one correspondence between specifications and implementations. If we have several implementations for a single specification, we can make each of the implementation classes be a subtype of a common type. If we have several specifications for a single implementation, we can create subclasses of the implementation class which are subtypes of each of the specification classes. These subclasses can make minor changes to behavior. If these subclasses would override too many methods, they could be implemented as encapsulators.

2.2.8 TYPING

Whether languages should be typed has been a long-standing debate in computer science. There are two major issues: speed and reliability. Type information can be used by the compiler to produce more efficient code. The debate here is how significant this speedup is. Type information can be used by the compiler to find bugs at compile time that wouldn't show up until run-time otherwise. On the other hand, it takes time for the programmer to declare the types of all the variables he uses. The debates here are whether the extra time taken to declare types has a significant effect on rapid prototyping and whether type errors are actually common for experienced programmers.

Strongly typed languages have been around for a long time, so typing may not be considered an advanced feature. However, in the object-oriented world they are a relatively new idea. As we have seen, dynamic typing is a basic requirement of object-oriented languages. Smalltalk is an untyped language: variables are declared with no type information at all—only their name is declared. Ralph Johnson experimented with adding typing to Smalltalk [Johnson et al 88]. In his language, variables were declared to be of a particular *type*. Classes are treated as

implementations of the types (specifications). The run-time type of the variable can then be any subtype of the declared type. For example, a variable declared to contain a **Number** can contain an instance of a class which implements the **Integer** type at run-time. The default declared type is **Object** so that types do not need to be filled in when prototyping. The type information can be used by the compiler to produce faster code.

Trellis [Schaffert et al 86] is another typed object-oriented language created by Digital Equipment Corporation. It does not separate the specification and implementation hierarchies but it does allow the run-time type of a variable to be a subtype of the declared type. The single hierarchy in Trellis is considered a type hierarchy because subtypes must conform to certain *subtype compatibility rules* that ensure an instance of the subtype can be used in place of the supertype. When a method is overridden in a subtype, the declared argument types can only get more general and the declared result type can only get more specific:

```
type module SuperType1
  operation foo (me, argument : SuperType2) returns ( SuperType3);
end type module;

type module SubType1 subtype_of (SuperType1)
  operation foo (me, argument : SuperType2) returns (SubType3)
    override;
end type module;
```

The `foo` operation in `SubType1` must accept an argument of `SuperType2` or a supertype, and it must return a result of `SubType3` or a subtype. The argument type gets more general as the type itself gets more specific, so it is *contravariant*. The result type gets more specific so it is *covariant*. Another subtype compatibility rule disallows the exclusion of public operations.

Both Typed Smalltalk and Trellis support *union types* and *parameterized types* (also known as *generic types* or *type generators*). Union types consist of a list of types. The variable can contain an object of any of the types listed. A union type can be thought of as an unnamed supertype of all the types listed. Parameterized types accept other types as arguments. The most common example is collections. We can declare a collection of integers by instantiating the generic collection type. Note that heterogeneous collections are still possible in both languages by using the type **Object** as the type parameter. Parameterized types effectively generate a family of types.

In the previous section, we considered having separate hierarchies for specifications and implementations but sharing elements. We argued that we could define multiple implementations of a single specification and vice-versa by making classes be subtypes of other classes. In a strongly typed language, subtypes must obey co/contravariant subtype compatibility rules. Consider two implementations of the two-dimensional point specification: one using cartesian coordinates and another using polar coordinates (see Fig. 2.13).

Figure 2.13. Multiple implementations of the Point specification.

The **Point** class would be an abstract class with no behavior; it would define a specification only. The **+** method in **Point** would look like:

```
+ aPoint <Point> ^ <Point>
  "Non destructive sum of coordinates."
```

Syntactically, this can be read as a method with selector **+** and argument **aPoint** of type **Point** which returns a result of type **Point**. The **+** methods in **CartesianPoint** and **PolarPoint** would provide implementations:

```
+ aPoint <Point> ^ <CartesianPoint>
  "Non destructive sum of cartesian coordinates."
  ^ self class x: self x + aPoint x y: self y + aPoint y

+ aPoint <Point> ^ <PolarPoint>
  "Non destructive sum of polar coordinates."
  ^ (self asCartesianPoint + aPoint asCartesianPoint) asPolarPoint
```

Note that the argument must always be of type **Point** (rather than **CartesianPoint** or **PolarPoint**) to obey the contravariant subtype compatibility rule. The result type can be **CartesianPoint** or **PolarPoint** since it is covariant.

We make a distinction between the *subtype* relationship and the *is-a* relationship [Lafonde & Pugh 91]. Subtypes must meet subtype compatibility rules ensuring that they are substitutable wherever a supertype is expected. On the other hand, the *is-a* relationship is based on the idea of a type as a (possibly infinite) set of potential instances. For example, we say that an **Integer**

“is a” Rational since the set of integers is a subset of the set of rationals. This corresponds to what we mean when we say “is a”. We demonstrate the difference between these two concepts using two examples.

Rational methods:

```

numerator ^ <Integer>
    ^ numerator
numerator: value <Integer>
    numerator := value

```

Integer methods:

```

numerator ^ <Integer>
    ^ self
numerator: value <Integer>
    self error: 'inappropriate'

```

In this example, we see how an integer n tries to pretend to be a rational by acting as if it were $n/1$. However, the integer can't keep up the illusion when it is asked to change its numerator. It is forced to signal an error¹. This violates the intent (although not the letter) of the subtype compatibility rules. Instead of getting a subtype compatibility error, no error is produced until run-time. What is the point of subtype compatibility rules if they don't protect us from such run-time errors? Our second example also fails when we execute a destructive method:

Array methods:

```

at: index <Integer> ^ <Object>
    ..
at: index <Integer> put: value <Object>
    ...

```

String methods:

```

at: index <Integer> ^ <Character>
    ...
at: index <Integer> put: value <Character>
    ...

```

¹It could try to mutate itself into a Rational at this point, but this hardly seems appropriate.

We say that a `String` is an `Array` because only some arrays have elements which are all characters. In this example, the subtype compatibility rules are violated outright. `String`'s `at:put:` method cannot accept any `Object`—it can only accept a `Character`. Generally if b is a subtype of a , then *list of b* is not a subtype of *list of a* , because *list of b* cannot be substituted for *list of a* if destructive operations are performed.

It is evident that subtype compatibility rules cannot ensure substitutability because they must be violated routinely. There are several ways of handling this situation. The simplest solution is to weaken the subtype compatibility rules so that they are merely suggestions. The programmer would be warned when he violates the rules, but that is all. Since the system cannot ensure substitutability in general, attempting to use a subtype where a supertype is expected may fail. The system can detect such calls which may fail at compile-time. Another solution is to reduce the set of methods for which substitutability is ensured. The programmer declares some of the public methods to be required in subtypes. The subtype compatibility rules will only be checked for these methods. Using a method which is not required in subtypes is less reliable since subtype substitution may fail. As with our first solution, the system can detect such calls. The problem with this solution is that the programmer cannot know which methods will be provided in subtypes. If he requires all of them, then later addition of subtypes becomes difficult or impossible. Our last solution is to allow subtypes to violate subtype compatibility rules but only if they declare that they are doing so. This moves the onus to the subtype. All of these solutions are essentially the same, in that exceptions to the subtype compatibility rules are allowed. Only a more limited substitutability can then be ensured.

Figure 2.14. A valid subtype hierarchy.

Rather than weaken the subtype compatibility rules, we can change our type hierarchy so that we conform to them. In both of the above examples, we can separate the read and write (destructive) behavior to avoid the incompatibility (see Fig. 2.14). Methods which both read and write cannot be put in the read-only or write-only types. Note that `String` and `Array` are no longer related. The problem with this solution is that the type hierarchy becomes quite complex.

2.2.9 METHOD ALIASING

Smalltalk allows subclasses to define new methods and to override existing methods. Trellis allows subtypes to exclude methods as well¹. Consider the subclass as a description of how to change the superclass in order to make it into the desired subclass. In that case, an override really tells the system to replace the superclass's method with a new one. An exclude tells the system to delete the superclass's method locally. So we can add new methods locally, replace inherited methods with local ones, and delete inherited methods altogether. What other edits would be appropriate and useful? One is *aliasing*;² a feature found only in Eiffel and POOL-I. This feature allows a subclass to inherit a method or instance variable from its superclass but under a different name. An example in Smalltalk is the instance variable `selection` in the `ListView` class hierarchy. One of the subclasses supports multiple selections. In this case, the variable contains a set of indices rather than a single index. The variable really should be aliased to `selections` or `selectionSet`.

Aliasing has many uses. One is to rename inherited methods and instance variables which function correctly but have inappropriate names in the subclass. We have already seen an example of this use. Another application of aliasing is in resolving conflicts resulting from multiple inheritance. As we have seen, sometimes instance variables with the same name should be merged and sometimes they shouldn't be. It may even be appropriate to merge

¹Subtype, visible methods can be excluded in Trellis. Private methods are not inherited so there is no need to exclude them.

²This is distinct from the use of the word "aliasing" to describe the undesired sharing resulting from multiple pointers to the same object

instance variables with different names. To solve this problem we use aliasing. To merge two (or more) variables with different names, we alias them to the same name. To avoid merging two (or more) variables with the same name, we alias one or both of them. Not only does aliasing allow us to specify whether or not variables should be merged, it allows us to refer to distinct variables easily and it prevents us from referring to merged variables distinctly. Note that Eiffel does not support merging of instance variables using aliasing, and therefore does not gain full advantage of this feature.

The third and final use of aliasing is to make the relationship between specifications and implementations be implicit. We have seen how encapsulators (adapters) can be used to map method names from the specification to the implementation. They can also provide their own behavior. However, encapsulators have many disadvantages. They add an extra level of indirection, taking up both time and space. Whenever new objects are created (by non-destructive operations for example), we must build new encapsulators around them. The only advantage of encapsulators over simple method name maps is their ability to include code which “polishes” the low-level methods provided by the implementation. However, this code cannot be reused (inherited) effectively. We can use aliasing to create a subclass which has method names which exactly match the type. Then we can make the “implements” relation be implicit—we no longer need to specify which classes implement which types. For example, we can define a subclass of `OrderedCollection` which renames the `addLast:` and `removeFirst` operations to `push:` and `pop`. This subclass would then implicitly be an implementation of the `Stack` type because it provides the required operations. We do not have to specify that this subclass implements the `Stack` type. A further advantage of this approach is the ability to include code in this class as well as aliases. This code can make minor changes to the operations provided by the superclass in order to make it conform exactly to the specification, and, unlike code in encapsulators, it can be inherited. For example, the `addLast:` operation returns the object being added whereas the `push:` operation might be expected to return the receiver.

2.2.10 PROGRAMMING IN THE LARGE

Smalltalk-80 comes with a fairly extensive class library and it is getting larger with each new version. Very few large applications have been written in Smalltalk to date. The largest application written in Smalltalk is still much smaller than many applications written in C. One reason that no very large applications have been written in Smalltalk (or other object-oriented languages) is that no company is willing to risk an unknown language on such a major undertaking. Another reason is that rapid prototyping, a software development technique frequently employed in Smalltalk, is best in small groups.

We must ask ourselves if Smalltalk is ready for very large applications. Smalltalk-80 has several features which are essential for large applications, including categories and change sets. *Class categories* allow the programmer to organize his classes into various categories. By convention, class categories form a two level hierarchy. For example, the class `Point` is in the `Graphics-Geometry` category. Methods within a class are organized similarly using *method categories* or *protocols*. It is important that method categories be consistent between classes, particularly when these classes are related. Categories make it easier to find things because they reduce the number of items you have to look through. However, the decision as to which category an item belongs in is somewhat arbitrary and it takes a long time for novice Smalltalk programmers to learn the conventions. Categories effectively separate applications from one another. However, the name space is still global. There can be only one class with a given name, and only one method in a class with a given name. This means that name conflicts can arise when two applications are filed in to the same image.

In Smalltalk-80, all changes made to code are recorded in a *change set*. This frees the programmer from remembering what he has changed. By switching between several *projects*, he can keep track of changes for several things at once. These change sets can be filed out to produce files which affect the changes made when filed in. The files can be browsed to handle conflicts between them. Smalltalk/V's *application browser* is similar although it does not automatically record changes. (This is a fairly easy feature to add.) All of these features allow several programmers to work on a project simultaneously. However, there is no central database to control access to the code. This means that there is no mechanism to prevent two

programmers from changing the same piece of code. All they can do is deal with the conflicts after the fact.

ENVY [OTI 91] is one system which handles multiple programmers vying for the same piece of code. It is an extension to Smalltalk-80 and Smalltalk/V. A central database is used to store versions of code and provide concurrency control. A programmer can check out a piece of code after which only he is allowed to edit it until he checks it back in. ENVY includes a large set of programming tools, including tools for comparing versions of methods. It also supports versioning larger pieces of code by constructing a *baseline* of versions of the components. Code is divided up into modules which explicitly reference one another. Modules are normally "owned" by an individual programmer. Modules can contain classes as well as methods to be added to classes defined in other modules. ENVY is a reasonable solution to the problem of programming in the large, but it is not perfect. It has no hierarchy browser or full browser, so there is no way to navigate the class hierarchy to browse inherited methods. It is difficult to rename classes, a problem familiar to Smalltalk/V programmers, but a new difficulty for those used to Smalltalk-80. Modules cannot edit methods defined in other modules; they can only add methods.

Versions of: ChangeSorter text from	
15. 12 Sep 88 2:45:54 pm kimr	15. 12 Sep 88 2:45:54 pm kimr
16. 12 Sep 88 2:47:29 pm kimr	16. 12 Sep 88 2:47:29 pm kimr
17. 12 Sep 88 2:48:16 pm kimr	17. 12 Sep 88 2:48:16 pm kimr

<pre> quotes in the string" self sourceSet comment: aStream contents]] ifTrue: "edit specialDots" [self sourceSet specialDots: OrderedCollection new. self extractDotsFrom: viewContents asString. self changed: #text]. self sourceSet modified: true. *true </pre>	<pre> quotes in the string" self sourceSet comment: aStream contents]] ifTrue: "edit specialDots" [self sourceSet specialDots: OrderedCollection new. self extractDotsFrom: viewContents asString. self changed: #text]. self sourceSet modified: true. *false </pre>

Figure 2.15. A Tektronix version browser.

Before ENVY existed, several other much more limited tools existed. In fact, Apple Smalltalk-80 had simple method versioning. Tektronix produced a set of team programming tools for their version of Smalltalk [Tektronix 88] (see Fig. 2.15). These tools supported multi-programmer development by improving the way change sets are treated. Rather than

having a single change set for each project (a screen full of windows), they were decoupled from projects and could be edited using the change sorter. The version browser allowed the programmer to see the entire history of a given method or class definition. Modular Smalltalk [Wirfs-Brock & Wilkerson 88] was an experimental version of Smalltalk that implemented modules similar to ENVY's. More recently, Digitalk announced a set of team programming tools for their Smalltalk/V for OS/2. These tools provide features similar to ENVY: modules, version control, and concurrency control.

Modules are one way to reduce global name space conflicts. Another way was proposed in [LaLonde 89]. A *family* of data types groups together several related classes and provides a single mechanism to access them. For example, to get an ordered collection, you would ask the collection family for a class with the appropriate properties: contiguous integer keys starting at one, one value per key, duplicate values allowed. The exact protocol used to access the classes in a family may be specific to that family. It would probably be useful to include a few global names for the commonly used classes within the family. At the very least, the family should include no-argument methods, such as `Collection list` for accessing these classes. Families of classes are one form of light-weight classes, classes which do not appear in the global name space.

2.3 SUMMARY

Before we discuss how the features listed in this chapter interact, we must consider the various options associated with each and how they affect one another. We start by considering the inheritance mechanism. Inheritance always supports defining new methods locally as well as overriding inherited methods. We may require overrides to be explicitly declared by the programmer. Excludes and/or aliases may be allowed. Representation can be inherited separately from behavior or it can use the same mechanism by always using accessor methods. In the latter case, unused instance variables need not be allocated. We may support dynamic inheritance. If multiple inheritance is supported, the superclasses can be linearized, prioritized into a list of sets, or unordered. To resolve ambiguities caused by representation inheritance in the latter two cases, we would like to support splitting and merging of inherited instance variables. There are two ways to do this. We can support explicit merges and splits as C++

does [Ungar et al 91, Stroustrup 91] or we can inherit representation as behavior and use method aliasing to control merges and splits. To access inherited methods, we use a super call. This can be either named or unnamed, or we can support both. We may require the overridden method to specify where the overriding method gets called (Beta's `inner`).

Another area where there are many options is typing. If variables are untyped, there can be no subtype compatibility rules. The result is an ad hoc implementation hierarchy, as in Smalltalk. If we put type declarations on variables, we can allow possibly invalid method calls (fuzzy typing) or not. We have several options for organizing types into a hierarchy. The type hierarchy could be a DAG to allow typical is-a relationships to be expressed¹. The subtype relationship could either be explicit or implicit. We could have a single hierarchy of types with subtype compatibility rules, or we could have separate hierarchies for types and classes. In the latter case, types and classes could be matched up either implicitly, using method aliasing to rename as necessary, or explicitly, where the corresponding methods are associated by the programmer. Yet another option is to keep separate hierarchies but share common elements between them.

If we have classes, rather than prototypes, we may include metaclasses for class representation and behavior. Special one-of-a-kind objects are another option. If we have separate hierarchies for types and classes, we have the option of replacing the class hierarchy with an instance hierarchy of prototypes [LaLonde 89]. If we want to support privacy, we have the option of limiting visibility to the class, the (recursive) subclasses, or we can provide both options. Another option is to support modules separately from classes and limit visibility based on them.

If we have a single hierarchy with subtype compatibility rules and we don't support methods being declared subtype visible, then there is no point supporting excludes since they will invariably break the compatibility rules. However, we may want to allow excludes anyway and live with a more limited substitutability. The programmer can be warned of violations of the subtype compatibility rules, or the programmer can explicitly declare some methods as

¹Using subtypes to represent is-a relationships has problems, as we saw in section 2.2.8, but it is still useful.

being required in subtypes. Another options is to allow the programmer to explicitly declare violations of the subtype compatibility rules. In all three cases, the system can detect the substitutability problems that arise as a result of the violations.

Figure 2.16. Feature option diagram.

Lastly, if we have multiple polymorphism, the programmer may be allowed to specify which arguments are polymorphic. The relationships between all the options we have discussed here are shown diagrammatically in Fig. 2.16.

Table 3. Advanced feature interaction.

	Multiple Inherit	Multiple Polymorph	Proto-type	Dynamic Inherit	Typing	Strict Hier	Method Aliases
Multiple Inherit							
Multiple Polymorph	similar cons ^{1,2}						
Prototype	work well together ³	work well together ⁴					
Dynamic Inherit	work well together ³	no conflict	work well together ³				
Typing	work well together ⁵	types req'd if MP&alias ^{1,2,7}	in-appropriate	types make DI safe			
Strict Hier	strict req MI ^{1,5,6}	no conflict ¹	in-appropriate	in-appropriate	strict req typing ^{1,5}		
Method Aliases	works well together ¹	if both then req typing ^{1,7}	no conflict	difficult	types req'd if MP&alias ^{1,7}	no conflict ¹	

Notes:

- 1 TMPL, an experimental language we will discuss in chapter 4, combines these features effectively.
- 2 CLOS and Dylan combine these features effectively.
- 3 SELF combines these features effectively.
- 4 The SELF group is considering adding multiple polymorphism to the language.
- 5 Trellis and Eiffel combine these features effectively.
- 6 A strict hierarchy is too restrictive without multiple inheritance. The is-a hierarchies being modeled are often DAGs.
- 7 We will see why in section 4.2.6.

We now consider how some of these features interact (see Table 3). Some features complement each other. For example, multiple inheritance and multiple polymorphism have many of the same disadvantages. Once a language includes one, it might as well include the other, since few additional difficulties are incurred. Other features conflict with each other.

For example, dynamic inheritance in a strict subtype hierarchy would be inappropriate. If two features have been combined successfully, the language(s) in which this was done are noted.

Not all features are shown in the table due to lack of space. Modules are essentially independent of all other features. The only contingency we must consider is that if we have a single subtype hierarchy with modules, then there is no point supporting excludes since they will always violate the subtype rules. We may choose to allow this and live with limited substitutability. Modular Smalltalk [Wirfs-Brock & Wilkerson 88] succeeded in extending Smalltalk to support modules.

Table 3 includes only one entry for a strict hierarchy. We have seen that there are several options for multiple hierarchies as well. We mentioned earlier that the type hierarchy should be a DAG. We also pointed out the possibility of making the implementation hierarchy consist of prototypes rather than classes. In section 4.2.4, we will see that dual hierarchies, with or without common elements, reduce the effectiveness of the type system considerably. In a multiply polymorphic language, almost everything must be classed rather than typed.

Several of the features discussed affect the performance of the system. In some cases, including a feature has no effect unless that feature is used. Garbage collection causes a slight slowdown which can be avoided by maintaining a free-list within the language. Of course, the benefits of automatic garbage collection are lost. The problem with garbage collection is that it can take control for considerable periods of time occasionally. This makes garbage collection inappropriate in many real-time applications. Message sends can be optimized so that they are not significantly slower than procedure calls. Type declarations and/or type inferencing can eliminate many message sends. The most significant benefit is that the inlined method can be optimized using type information not available for the method in general. Multiple inheritance and multiple polymorphism both affect the performance of method lookup since more complex data structures are needed. Even accessing representation is slowed down by multiple inheritance since we cannot compile them to indices unless we compile a separate version for subclasses. By always accessing instance variables through accessor methods, we can use method lookup to eliminate this problem.

CHAPTER 3. OBJECT-ORIENTED LANGUAGES

3.1 SMALLTALK

Smalltalk is not the first object-oriented programming language (Simula 67 would have that honor), but it is the canonical object-oriented programming language in that all other object-oriented and object-based languages are inevitably compared to it. It started as a project at Xerox Palo Alto Research Center (PARC) [Krasner 83]. The first version of the language was an interpreter implemented in 1972. New versions were produced every other year until 1980. The final version, called Smalltalk-80 [Goldberg & Robson 83] was distributed to Apple, Digital Equipment Corporation, Hewlett-Packard, and Tektronix. Each of these companies produced their own implementation of the virtual machine (VM) running Xerox's virtual image (VI) on their own hardware. Meanwhile, the Software Concepts Group at Xerox PARC produced an implementation on the Dorado, an experimental high-performance microprogrammed computer. All these implementations led to many improvements in both the VM and the image. Xerox spun off the Smalltalk project, creating ParcPlace Systems Inc., the current vendor of Smalltalk-80.

As we have seen, Smalltalk is a class-based language with garbage collection and a complete graphical user interface with a sophisticated programming environment. It supports only single inheritance of representation (without overrides) and behavior (with overrides but without excludes). Smalltalk is really three things: an object-oriented programming language, a class library, and a GUI. Each of these components is important. The large class library makes it possible for programmers to spend less time "re-inventing the wheel" since they can reuse so much code. The GUI lets programmers build complex user interfaces for their applications quickly. It is all of these features combined which make Smalltalk such a productive programming tool. As the saying goes, "everything is an object" in Smalltalk. What this means is that classes, blocks, methods, messages, and so on are all *first-class objects*—they have a class and they can be stored in variables and passed around, just like other objects.

3.1.1 SMALLTALK-80

Smalltalk-80 has progressed significantly since 1980 although there are still parts of the image which have been untouched since then. The idea of a virtual image (VI) which can run on any virtual machine (VM) has been kept. Consequently, developers don't have to maintain separate versions for each platform. The source code for the entire class library is included. The image is substantially larger than it was in 1980, including many collection classes, numeric classes, user interface classes, operating system classes, and of course, kernel classes. Some extensions to the image are now marketed separately by ParcPlace [ParcPlace]. These include a binary object storage system and extensions to the programming environment.

The most recent version, Objectworks\Smalltalk-80 4.1, is quite different than the initial Smalltalk-80 implementations. It uses a much more sophisticated garbage collection scheme. The garbage collector executes *finalization code* after an object is deallocated. Methods are still compiled to compact byte codes, but these are no longer interpreted. Instead, they are translated into native machine code and stored in a cache. This technique is known as *dynamic translation* [ParcPlace 92a]. It avoids storing the larger native code methods and yet manages to achieve almost the speed of a native-code compiler. An *exception handling mechanism* is included in the language, including a `valueNowOrOnUnwindDo:` method which ensures that termination code runs, even if the main code terminates prematurely. When the definition of a class changes, all existing instances must *mutate* to conform to the new definition. This is done automatically. Instance variables which are no longer required are removed. New instance variables are given the value `nil`. Classes are instances of corresponding *metaclasses*. This feature, along with the fact that everything is first-class and the existence of source code, makes Smalltalk-80 suitable for *meta-programming*, the act of changing the language from within itself; e.g., adding support for backtracking [LaLonde & van Gulik 88].

Older versions of Smalltalk-80 (up to, but not including 4.0) had a very small set of graphics primitives, the primary one being `BitBit`. Smalltalk created all panes and windows itself. Newer versions interface to the host window system through a much wider protocol. The panes and widgets inside the windows are still created by Smalltalk, but the windows are now created by the host's window system. This allows applications written in Smalltalk to coexist

with other applications better. It is likely that Smalltalk-80 will eventually use the host window system's panes and widgets rather than creating its own. Their latest product, VisualWorks, emulates host widgets in Smalltalk [ParcPlace 92b].

3.1.2 SMALLTALK/V

Smalltalk/V [Digitalk 89] is a more recent dialect of Smalltalk produced by Digitalk Inc. The original version ran on IBM PCs and is now called Smalltalk/V DOS. A version for 286s and a version for the Macintosh soon followed. Now there are versions for Microsoft Windows and IBM's OS/2 Presentation Manager. These languages are sold at lower prices than Smalltalk-80, allowing them to reach a wider market. The class libraries aren't as extensive as Smalltalk-80's class library, but they are also simpler and therefore easier to learn. All the versions except for V/PM are dynamically translated. The PM version compiles straight to native code, producing large images which run quite fast. The different versions of Smalltalk/V are quite similar but they do not support image compatibility across platforms the way Smalltalk-80 does. On the other hand, each version provides convenient access to host-specific features, allowing Smalltalk/V to be used for fairly low-level programming and making Smalltalk/V applications look and feel more like other applications on the same platform.

3.1.3 PROBLEMS WITH SMALLTALK

Because Smalltalk has been around for a long time compared to many of the other languages discussed here, it has many problems which have been carried over from the original version. This is not to say that the language has not been improved since its inception. The semantics of the language have been cleaned up, particularly blocks, the garbage collection algorithms have been improved, the use of dynamic translations has improved performance, and the class library has been expanded, particularly the user interface classes. Smalltalk is an incredibly flexible language, particularly Smalltalk-80, because everything is accessible from within the language, even many things that could be considered the language itself.

In Smalltalk, representation is always inherited. Specializations often require fewer instance variables rather than more. For this reason, Smalltalk's class hierarchy is often based on

representation rather than behavior or some logical relationship between classes. For example, Dictionary is a subclass of Set because it is implemented as a set of associations, but it would more logically be treated as a generalization of Array with arbitrary indices [LaLonde 84]. Methods cannot be excluded or renamed in Smalltalk. Indexed fields are treated differently from ordinary named instance variables. They are accessed via a primitive rather than a byte code and they can effectively be excluded. Smalltalk-80 has class instance variables but Smalltalk/V does not. These are like class variables except that each subclass has its own value for the variable.

There is no formal way to declare methods as being private. The convention commonly used is to place private methods in a protocol (a method category) named `private`. There is nothing protecting private methods from being used inappropriately. A better convention is to name private methods differently; eg., `privateComputeCoordinates`. This ensures that callers of the method are aware that they are using a private method.

In Smalltalk, methods with names like `basicNew` are often used. Such methods are used to refer to "root" methods in submethods. These are needed only because the super message send does not support naming a superclass directly. In many cases, the programmer wants to invoke the "super super" method. In order to do this, he must define an alternate name for this method that will not be overridden. As we pointed out in section 2.1.6, the top of the class hierarchy is quite complicated, largely because of metaclasses (see Fig. 2.2). There are several classes with a single instance such as `True`, `False`, and `UndefinedObject`. These would be better represented as special objects.

There is one particularly annoying problem with Smalltalk's programming environment that has yet to be corrected. Browsers often end up showing old versions of code because the code has been edited in another browser or a debugger. When the old version is edited and saved, previous changes are lost. A proper update mechanism is needed to prevent this.

Although Smalltalk efficiency is much improved over early interpreters, it is still approximately ten times slower than optimized C code [Chambers & Ungar 90]. SELF has managed to improve on this significantly. We will see what makes this possible in section 3.3.

3.2 TYPED SMALLTALK

Typed Smalltalk is an experimental version of Smalltalk developed by Ralph Johnson with explicit type declarations added to the language [Johnson et al 88]. Rather than declaring variables to be instances of a particular class, Johnson chose to introduce types as distinct entities from classes in the language. Types are specifications and classes are implementations. If a class implements the operations required by a type then it can be said to *implement* the type. Types form a strict subtype hierarchy whereas classes are arranged for convenience of implementation and reuse. All variables in TS (temporary variables, arguments, instance variables, and class variables) can be declared to be of a particular type. Instances of any class which implements the specified type or any of its subtypes can be stored in the variable. If no declaration is given, any object can be stored in the variable. This is convenient for rapid prototyping. The type declarations in TS are verified and the user is notified of any contradictions. By gradually adding type declarations as an application matures, the reliability of typed languages can be achieved without the loss of flexibility normally entailed.

In addition to the improved reliability of typed code, TS provides another advantage over regular (untyped) Smalltalk. The compiler can use the type information provided by the programmer to optimize the compiled code. If the class of a variable can be determined at compile time, the method that will be executed when a message is sent to it can be determined. If this method is fairly small, it can be *inlined*, eliminating the cost of the method lookup and the call. The inlined code can be optimized further since the class of the receiver is known. Even if the class of a variable cannot be completely determined, so long as there are a small number of possible methods, each of which is small, the various options can all be inlined along with a case statement to choose the inlined method to execute at run-time. TS compiles to a register transfer language which can be optimized using conventional techniques such as peephole optimization and common subexpression elimination. Since primitives are written in this language as well, primitive calls can be inlined as well. The result of all these optimizations is a significant speedup over regular (untyped) Smalltalk. Of course, only code with type declarations can be optimized. Because the entire class library was not rewritten with type declarations, the full advantage of these optimizations was not realized. The difficulty in adding type declarations to the built-in Smalltalk classes indicates a potential problem with the

methodology as well. Perhaps adding type declarations to a prototype isn't as straightforward as we'd like to believe. It would appear that in order to take full advantage of typing, it must be considered from the outset.

As we mentioned in section 2.2.8, TS support parameterized types, such as *Array of someType*. Heterogeneous collections can still be created as *Array of Object*. TS also supports *signature types*. These types are based on behavior rather than classes. An example of a signature type is "anything that responds to *isEmpty* with a *Boolean*".

3.3 SELF

SELF is a prototype-based language currently under development at Stanford University by Dave Ungar et al [Ungar & Smith 87]. As we have seen, SELF uses prototypes instead of classes and it provides multiple inheritance with partially ordered parents (a list of sets) by prioritizing the parents of an object. The syntax of SELF is a slight modification of Smalltalk's syntax, the only important difference being the idea of a default receiver. If no receiver is specified, the message is sent to *self* (hence the name of the language). The virtual machine includes a generation-scavenging garbage collector [Ungar 84, Chambers et al 89]. As with most Smalltalk systems, the language is executed by dynamic translation. SELF's dynamic translator produces separate native code for each type of receiver of a given method [Chambers et al 89]. This takes more memory but it allows the translated code to be more efficient because more type inference can be done resulting in more inlining of methods. New versions of the SELF virtual machine go a step further. They translate using a fast non-optimizing¹ translator when a method is not in the cache. If the method gets used many times, it is re-translated using optimizations so that subsequent executions of the method are faster. The most recent versions of SELF run about four times as fast as ParcPlace Smalltalk-80, or almost half as fast as optimized C [Chambers & Ungar 90].

One of the primary goals in the design of SELF was simplicity [Ungar & Smith 87]. Inheritance is used in SELF for many purposes. A prototype and its clones inherit from their

¹Actually, some very quick optimizations are done even in the initial translation.

traits object (like a class but without representation, as we saw in section 2.2.2), just as an instance inherits from its class. A traits object inherits from one or more other traits objects, just as a class inherits from its superclass(es). Rather than containing methods directly, a traits object can store methods in several protocol objects which it inherits from. These protocol objects are ordinary objects being used to organize methods, just as a class uses protocols to organize its methods. In order to make an object globally accessible, we store it in a slot in the *lobby*. This is a special object which is used as the starting point for global variable lookups. Groups of related objects can be stored in category objects, similar to the protocol objects used to organize methods. SELF has successfully integrated all these relationships into a single one: inheritance. Actually, using objects and inheritance to provide class and method categories (protocols) is an improvement. The categories can be nested any number of levels deep-- even different numbers of levels in different areas. Objects can be placed in several categories without difficulty. In order to reduce the global name space, we can group globals into category objects which are not inherited by the lobby. To refer to globals in such categories, we must send a message to the category; eg., `traits: fact`. These category objects can be nested, in which case we must specify the entire path. This is known as *extensional naming*.

As we have seen, SELF also succeeds in merging the concepts of instance variables and methods into a single one: slots. These slots can be declared public or private. Only descendents and ancestors of an object can access its private slots. This implies that a method defined in a traits object can access the private slots of all the “instances”; ie., the prototype and all its clones. This is similar to C++’s type-based encapsulation which we will see in section 3.6. The concept of superclasses is also merged in with these since objects indicate other objects they inherit from by pointing to them in *parent slots*. Since these slots can be writable, an object can change which traits object it inherits from at run-time. This is dynamic inheritance. Because traits do not normally include representation¹, it is not normally inherited. As we have seen, data parents provide a way to inherit representation, but it is messy. SELF’s

¹If traits do contain representation, it is shared among all the clones/prototypes. This is equivalent to class variables.

sender path tiebreaker rules fall short of solving the ambiguity problems resulting from data parents.

SELF's programming environment is still in the experimental stage. Although it is far from fully functional, it demonstrates several unique ideas. As we saw in section 2.1.8, the only tool provided is an inspector, but since code is stored in slots just like data, these inspectors can be used to browse code. However, there is no way to write programs within the GUI. SELF's inspectors are similar to Smalltalk's inspectors with a few important differences. As we have seen, "spawned" inspectors are linked to the inspector that spawned them by a line (see Fig. 2.4). Also, there is a one-to-one correspondence between objects and their inspectors. The features of the SELF user interface are appropriate for programming with prototypes since they encourage direct interaction with objects rather than browsing (a class's) code.

In Smalltalk, and other conventional object-oriented languages, helper methods are simply methods which are marked as such, usually by putting them in a private category or naming them specially. SELF can support private methods directly, but it can also support scoping of helper methods within the public methods that use them, similar to the way helper methods can be scoped in Pascal. The helper method is simply a local variable of the public method that uses it. Arguments and local variables of the public method cannot be accessed directly in the helper method—they must be passed in to it. Note that current versions of SELF do not yet support local methods.

3.4 TRELLIS

Trellis (which used to be called Trellis/Owl) is an object-oriented language produced by Digital Equipment Corporation [Schaffert et al 86]. Unlike Smalltalk and SELF, it is designed to be compiled rather than interpreted or dynamically translated (Smalltalk/V PM is an exception—it is compiled). Like most compiled languages, Trellis is typed. As we have seen, the run-time type of an object stored in a variable is allowed to be a subtype of the declared type of the variable. This feature is essential since it allows message passing to be polymorphic.

Trellis uses modules with separate specifications and implementations to achieve encapsulation. Special `type_modules` are used to define *types*, Trellis's term for classes. As we have seen, types can inherit from a set of supertypes. Ambiguities arising from multiple inheritance must be resolved by the programmer. The type hierarchy (a DAG) is based on specification rather than implementation, so programmers must obey co/contravariant subtype compatibility rules which ensure that a subtype can be used in place of a supertype in all cases. Inherited operations can be overridden or excluded but overrides must be declared explicitly and excluding a public operation violates the subtyping rules. Variables can be declared to be of a union type, allowing the value to be a subtype of any of the declared types. Variables contain references to objects so assignment doesn't copy. Objects which are no longer referenced are automatically garbage collected. As we have seen, Trellis includes a sophisticated programming environment, in many ways superior to Smalltalk's. The type information allows the programmer to browse the call graph more effectively by reducing the number of methods listed. Incremental compilation at the method level provides the performance required for rapid prototyping.

The syntax of Trellis is similar to Ada's. Methods are called *operations*, and they may or may not return something. As we have seen, they can be declared to be `public`, `subtype_visible`, or `private`. A dot notation can be used to access the *components* of an object. This is syntactic sugar for get and put methods, each of which can be declared to be `public`, `subtype_visible`, or `private`. Components can be implemented as *fields* (actually stored) or as *operations* (methods pretending to be data). Type operations and type components (class methods and class variables) are supported. The ability to implement components as operations implies that making your representation public does not really violate encapsulation since the components can be implemented as operations if the representation ever changes.

Trellis supports *type generators* which allow types to be parameterized by other types, similar to Ada's generic packages. The programmer can define his own enumeration operations using *iterators*, similar to CLU. These are special methods which `yield` elements to the body of a `for` loop. Another feature taken from Ada is exceptions. The only difference is that the exceptions an operation can signal must be explicitly declared.

There are several problems with Trellis. There is no way to pass around code as an object so there is no way for the programmer to define his own control structures except for iterators. Similarly, only built-in operators can be used. The fact that the language is typed causes some complexity, including the need for type generators and co/contravariance. Of course, all variables can be declared to be of type `Object`, eliminating these complications. Overall, Trellis is a promising language. It is an effective compromise between rapid prototyping and production programming. The success of the language will depend largely on the quality of the class library and the availability of the system on several platforms.

3.5 EIFFEL

Eiffel is an object-oriented language and programming environment developed by Bertrand Meyer at Interactive Software Engineering, Inc. [Meyer 87]. It is very similar in many respects to Trellis. Like Trellis, Eiffel is a typed language, allowing it to be efficiently compiled. As in Trellis, the run-time class of an object stored in a variable may be a subclass of the declared class. This allows message passing to be polymorphic (in some cases) even though the language is typed.

Eiffel is a class-based language with a syntax similar to those of Trellis and Ada. There is a single hierarchy of classes (a DAG as we shall see shortly). Inherited routines can be *redefined* (overridden) explicitly but they cannot be excluded. *Routines* and *variables* (*features*) are both accessed with the dot notation so that users of a class do not need to know whether they are accessing representation or calling routines. Features must be explicitly *exported* or else they are *secret* (private). Features can also be exported to specific classes only. All objects must be explicitly created but they are automatically garbage collected when they are no longer referenced.

Other features shared by Eiffel and Trellis are multiple inheritance and generics. Eiffel classes inherit from a set of superclasses and no ambiguity is allowed. In Trellis, ambiguities are resolved by overriding. In Eiffel, another option is available. Features can be *renamed* as they are inherited to avoid name conflicts. This aliasing can also be used to rename features to more appropriate names. Eiffel's generic classes are just like Trellis's type generators.

Classes can be parameterized. For example, a list class could have a parameter for the element type.

One unusual feature of the language is *assertions*. Each routine can have *preconditions* and *postconditions* which are automatically checked. Also, each class can have *invariants* which are also checked. Assertions are essentially formalized comments which can be verified by the compiler and/or the run-time system. When a routine is redefined (overridden), the new assertions must obey co/contravariant rules. The precondition can only get weaker and the postcondition can only get stronger.

The Eiffel implementation compiles to C. This is done to achieve portability but it implies that any change to your code requires the entire class being changed to be recompiled to C as well as any other classes affected by the change, after which the resulting C code must be compiled and linked. The programming environment takes care of all this for you but it is quite slow. It also provides a debugger and an inspector. A set of basic classes is included.

If we compare Eiffel to Trellis, we can see that the Trellis implementation is far more polished. They have spent more time on the programming environment and on incremental compilation and dynamic linking. The result is a more seamless and easier to use system. However, Eiffel has several features that Trellis does not. Method aliasing and assertions are both important features.

3.6 C++

C++ is an object-oriented version of C produced by AT&T [Stroustrup 91]. As an extension to C, it supports all of C as well as new object-oriented features. Like C, C++ uses value-based semantics so objects are copied when assigned or passed as parameters. Copying large objects can be avoided by passing references; these are equivalent to Pascal's `VAR` parameters. Because C++ is backward-compatible to C, it does not force programmer to program in an object-oriented style:

"Because [C++ and Objective-C] mix procedural and object-oriented concepts, hybrids make it harder for a beginner to learn just about the new concepts. ... In a hybrid language one can write objects, but only if you know what an object is. Nothing forces you to make objects, or even guides you towards objects." [Smith 91]

C++ is a fairly conventional object-oriented language in that it is class-based and uses message passing. However, most implementations do not provide automatic garbage collection and C++ does support multiple inheritance. As we pointed out in section 2.1.7, manual deallocation of objects can violate encapsulation [Sakkinen 88]. There is a single class hierarchy which is a DAG. This hierarchy is a subtype hierarchy in that subclasses can be used wherever superclasses are expected. No subtype compatibility rules enforce this substitutability; it is left up to the programmer. One unusual feature of C++ is its ability to define a subclass which is not considered a subtype. This is equivalent to containing the superclass but does not entail the overhead associated with encapsulators. All variables in C++ are typed. The value of a variable can be any subtype of the declared type. However, it is illegal to send a message to a variable whose contents may not understand it. *Templates* provide generic or parameterized classes such as *Stack of <anytype>*. Heterogeneous collections are supported but *Stack of Object* is not, both because there is no *Object* class and because it would be illegal to send most messages to the elements.

C++'s multiple inheritance mechanism has several unusual features. A class declares a set of *base classes* (C++'s term for superclasses) from which it is derived (C++'s term for a subclass is a *derived class*). Ambiguities produce compile-time errors. If these base classes share a common ancestor, its members (representation and behavior) may be duplicated. As we argued in section 2.2.1, this is rarely desired. By declaring the common ancestor to be

virtual, we avoid duplicating its members. However, C++ provides a mechanism for merging and splitting variables inherited from multiple base classes. A class can be declared abstract by defining *pure virtual functions* which must be overridden in derived classes. The compiler will give an error if an attempt is made to instantiate such a class or a derived class which has not overridden all the required methods.

Like Ada, C++ supports optional arguments and user-defined operators. Because C++ does not have Ada's name-parameter association, if an optional argument is included, then all preceding arguments must be included as well. Methods can define their own methods for the standard operators such as `+`. As in Ada, the precedence rules are fixed. However, C++ allows the programmer to define operators for assignment, subscripting, pointer dereferencing, and even function calls. (Assignment is not inherited normally—the default assignment operator calls the assignment operators for all the base classes.) Operators are just syntactic sugar for functions; to make a super call (only named super calls are supported) to an operator, you must use the function syntax.

Another feature that C++ has in common with Ada is *operator overloading*. Actually, C++ generalized this concept to *function overloading*, since operators are just syntactic sugar for functions. Several *member functions* (C++'s term for methods) with the same name can be defined in a class without conflict, so long as they accept different numbers and/or types of arguments. This is not true multiple polymorphism since all of these methods can be overridden by a single method with this name in a derived class [Goldsmith & Palevich 90]. Method lookup is based on the class of the receiver, first, and then on the argument types.

In order to avoid having to define overloaded operators for every pair of numeric types, the programmer can define *coercion operations*. These operations will be called automatically where necessary to convert from one class to another. Either the source or the target class can define the method. The system will not use multiple coercion operations transitively so the programmer must define these coercions if desired. Defining a large number of coercion operations can easily introduce ambiguities.

C++ supports both C's separately compiled modules (source files) and class based visibility. Members can be declared public, private, or *protected* (subclass visible). Data members

(instance variables) should always be declared private or protected. If we want to allow access to a data member, we can define a public accessor method. This allows derived classes to override this method. Unlike most object-oriented languages, C++ uses *type-based encapsulation*. Private members of any instance of a class are visible in methods defined in that class. This allows binary operators and instance creation methods (called *constructors* in C++) to manipulate the private members directly, avoiding the need for public accessors and initializers [Ungar et al 91]. C++ also supports *friends*; e.g., vectors and matrices could be friends, allowing them to access each other's private members.

3.7 COMMON LISP OBJECT SYSTEM (CLOS)

The Common Lisp Object System [DeMichiel & Gabriel 87, Bobrow et al 88] is a standard object system for Common Lisp [Steele 84]. It is the result of merging two object-oriented extensions to Common Lisp: Symbolics' New Flavors [Moon 86] (a version of Flavors with generic functions, a feature which will be discussed below) and Xerox's Common Loops [Bobrow et al 86]. Lucid was involved in the technical design work.

CLOS is a class-based language with garbage collection (a given for Lisp systems). The class system extends Common Lisp's built-in type system so that built-in Lisp types (such as `Array`) have corresponding classes in CLOS. All variables in Common Lisp can be typed. This is also true of CLOS. Classes specify structure by listing slots. Accessing methods can be defined automatically if desired. Slots can be declared as either *local* (equivalent to an instance variable) or *shared* (equivalent to a class variable). Subclasses can *shadow* (override) a superclass's slot descriptions, making the types more specific. As we will see, methods can also be overridden. Excluding a slot or method is not supported. The only encapsulation mechanism in CLOS is the (somewhat primitive) module system in Common Lisp.

CLOS is based on three primary concepts: generic functions, multiple inheritance, and method combination. Generic functions implement multiple polymorphism in CLOS; they are a replacement for message passing. A *generic function* is a set of methods, all with the same name. The method (or methods, as we'll see) to execute when the generic function is called is determined by the classes (or identities) of the arguments. A default method can also be

specified. A CLOS class has a list of (immediate) superclasses (so the class hierarchy is a DAG). The order is called the *local precedence order*. To resolve ambiguities caused by multiple inheritance of slots or methods, the (transitive) superclasses are linearized into a full ordering which is consistent with the partial order defined by each class's local precedence order. Multiple inheritance of slots with the same name never causes these slots to be merged. If a generic function has several methods which can be applied to the given arguments, a *method combination* rule specifies which of these methods execute and in which order they do so. The potential methods are ordered (based on the linearized list of superclasses and the order of the arguments) so that the most specific method is first. The programmer can define his own method combination rule or he can use the standard one. In the latter case, the most specific method is executed. If it evaluates `call-next-method` (equivalent to `super`), the next method in the list is executed. Pre- and post-actions are supported. Generic functions provide method inheritance with overrides since passing in a subclass for one of the arguments will execute the more specific method, provided all other arguments remain unchanged.

When the definition of a class changes, all of its instances must *mutate* to conform to the new slot descriptions. This is done automatically (as in Smalltalk-80), but the user can define mutation methods to do it more intelligently. For example, if the definition of the `point` class is changed from cartesian to polar coordinates, all existing instances can be converted to the new coordinate system. CLOS is based on a *meta-object protocol*; it is also implemented in layers. This makes the language ideal for experimenting with different object-oriented paradigms. It also makes it possible to do *meta-programming*, adding features to the language itself. Classes and generic functions are first-class objects—they can be passed around and stored in variables, just like other objects. Classes are instances of corresponding metaclasses, as in Smalltalk.

3.8 DYLAN

Dylan is a new object-oriented language designed by Apple [Betz 92]. It has many similarities to CLOS, including a Lisp-like syntax, automatic garbage collection, dynamic typing, classes with multiple inheritance, and multiple polymorphism using generic functions. Instead of CLOS's complex method combination rules, Dylan supports only simple *next-method* calls.

Unlike CLOS, Dylan is not built on top of a functional language—all the built-in types are classes. Like most Lisp-like languages, Dylan supports optional arguments as well as lists of arguments. A more unusual feature is multiple return values. Although Dylan is a class-based language, it allows slots and methods to be associated with a particular instance, gaining many of the benefits of prototype-based systems.

3.9 POOL-I

POOL-I is an experimental typed object-oriented language with inheritance and subtyping [America & van der Linden 90]. It includes support for parallelism as well, but we won't consider this feature of the language here. The most significant feature of the language is its separation of types and classes. The programmer defines types and classes separately. Classes inherit from their superclasses. Types form a separate hierarchy based on co/contravariant subtype rules much like those of Trellis. The programmer need not declare the subtype relationships explicitly; if he does, the compiler verifies this assertion. The matching of types and classes which implement them is implicit. There is a many-to-many relationship between types and classes. Types need not be defined for all classes; the system defines a type with the same name as the class automatically. Generic types such as *List of <anytype>* are supported. The argument type can be bounded from above to ensure that it provides certain operations. The programmer can indicate that the generic type is covariant or contravariant in the argument type. As we saw in section 2.2.8, this is not the case for destructive collections such as lists.

POOL-I has multiple inheritance; classes declare a set of superclasses from which they inherit. Ambiguities can be resolved by renaming inherited methods. A garbage collector automatically deallocates unreferenced objects. Accessor methods for instance variables are created automatically if requested. Both types and classes can include a list of *properties*, as mentioned in section 2.2.6. These are simply keywords used to declare various attributes of the type or class, such as *FLAT*, *NonDestructive*, *Truncatable*, etc. These property lists prevent the system from equating two types or classes with the same operation names. For example, *queue* and *stack* could easily be confused since their operations could be named

identically. By declaring the `Queue` as `FIFO` and the `Stack` as `LIFO`, these types will no longer be considered equal.

3.10 SUMMARY

Table 4 shows which languages support which features. This chart is useful for comparing these languages. The first column shows whether the language supports multiple inheritance and if so how it resolves ambiguities. It also indicates whether representation is treated as accessor methods by the inheritance mechanism. The second column indicates whether the language supports multiple polymorphism and whether blocks are fully supported. The third column indicates whether the language is prototype-based and whether classes are first class objects. The next column indicates the status and quality of the programming environment. The next column indicates what kind(s) of hierarchy the language has and what kind of visibility control is supported. The word “private” here means that methods can be declared private making them visible only locally and in subclasses. The words “subtype visible” indicate that the programmer can choose either subtype visible or truly private. The last column indicates how the language is typed and what kind of subtype compatibility rules are required. All the languages we have discussed are listed, as well as Objective C [Cox 86], Beta [Kristensen et al 87], Modula-3 [Harbison 92], and TMPL, an experimental language which we discuss in the next chapter.

Table 4. Features supported by object-oriented languages.

	Inheritance Rep	Polymorphism Blocks	Classes 1st class	Prog Env	Spec & Impl Visibility	Typing Rules
Smalltalk	single no rep override	message passing full blocks	classes 1 st class	good	impl hier	untyped
SELF	multiple ¹ rep as behavior	message passing full blocks ²	prototypes	prototype	impl instance hier private	untyped
Trellis	multiple set ³ rep as behavior	message passing iterators only	type 2 nd class	very good	spec hier subtype visible	typed co/contra
Eiffel	multiple set ⁴ rep as behavior	message passing	classes	adequate	spec hier classes are modules	typed co/contra
C++	multiple set	message passing ⁵ function params	classes 2 nd class	-	spec hier files&subtype vis.	typed none
CLOS	multiple list virtual slots ⁶	generic functions lambdas	classes 1 st class	-	impl hier Lisp modules	optional
Dylan	multiple virtual slots	generic functions full blocks	singletons 1 st class	-	impl hier	optional
POOL-1	multiple set auto accessors	message passing	classes 2 nd class	-	spec&impl hiers -	typed co/contra
Objective C	single	message passing	classes	-	impl hier	typed
Beta	single	message passing full blocks	classes	good	spec hier	typed
Modula-3	single	message passing	classes	-	spec hier modules	typed
TMPL	multiple set ⁴ rep as behavior	multiple full blocks	classes 1 st class	prototype	spec hier modules ⁷	fuzzy ⁸ covariant

Notes:

- 1 SELF prioritizes supers as a list of sets.
- 2 The SELF group is considering adding multiple polymorphism to the language.
- 3 Trellis supports excluding inherited methods.
- 4 TMPL and Eiffel support method aliasing which can be used to resolve ambiguities caused by multiple inheritance.
- 5 C++ supports operator overloading, providing limited multiple polymorphism.
- 6 CLOS method combination can be used to resolve ambiguities caused by multiple inheritance.
- 7 TMPL does not currently support modules.
- 8 TMPL requires result types to be covariant only so instance variables can be specialized.

CHAPTER 4. TMPL: AN EXPERIMENTAL LANGUAGE

In order to investigate each of the features discussed in chapter 2 more fully, several of these features were incorporated into a language design. A prototype implementation of this language was then implemented. TMPL is an acronym for Typed Multiply Polymorphic Language, these being the two most significant features of the language, as we will soon see.

4.1 LANGUAGE DESIGN

Before choosing which individual features to include in our language, we should look at existing languages for overall approach. Trellis is the language which comes closest to the style of programming which we believe an advanced language should support. But of course, it is only a starting point. Like Smalltalk, it includes a large class library with user interface classes. Its programming environment is superior to Smalltalk's in many ways. Unlike Smalltalk, it is typed, it supports multiple inheritance, and access to representation is strictly through accessor methods. Now we will go through each feature and decide whether our language should incorporate it, and if so, how it should be incorporated. In this section, we describe the design, and not the implementation. As we will see in section 4.2, several changes were made to the design during implementation.

4.1.1 INFORMATION HIDING

One of Smalltalk's deficiencies is its lack of privacy enforcement. Privacy is just a matter of convention; private methods are put in separate method categories. Many languages make privacy declarations formal and then enforce them. We will take this approach in TMPL. Like type declarations, privacy declarations are annoying during rapid prototyping. For this reason, SELF chose to make the default visibility be *public*. The only remaining decision is whether to distinguish between *private* and *subtree visible*. Truly *private* methods are not inherited by subclasses whereas *subtree visible* methods are. Since subclasses can be added later which require access to inherited *private* methods, it is best to support only *public* and *subtree visible* (which would be called *private*) methods.

Since TMPL is multiply polymorphic (as we will see), we cannot use the class of the receiver to determine the scope of private methods. The alternative is to use *modules*. These provide separate name spaces to reduce name conflicts in large systems and they allow collaborating classes to use one another's private methods without the need to define pool dictionaries or friends. Simply putting all the definitions into a single module provides the desired visibility. Modules explicitly declare which other modules they *use* and which other modules they *specialize*. A module which specializes another module can access its public and private elements. A module which uses another module can only access its public elements.

4.1.2 INHERITANCE

All object-oriented languages support inheritance by definition. Smalltalk supports method overrides but does not support excluding methods or renaming them. Instance variables can only be added. To make inheritance more uniform, SELF and Trellis chose to always refer to instance variables through accessor methods. This simplifies the inheritance mechanism since representation and behavior are governed by the same rules. Trellis avoids allocating instance variables which have been overridden. Trellis supports excluding methods (and therefore instance variables) as well as overriding them. For TMPL, we will follow Trellis's example for all these features.

Proper method aliasing, in which the old name becomes unreachable, is an important feature. It allows you to rename an inherited method (possibly a get or set method) to a more appropriate name for the subclass. Simply defining a call-through method does not make the old name illegal. Overriding the old method with an error will make the other inherited methods fail because they will use the old name. One of the uses for method aliasing is splitting inherited methods with the same name. If a class inherits two methods with the same name, one from each of its superclasses, then they will conflict. If these methods are really different, then one or both can be aliased to remove the conflict. Inherited methods in both superclasses will work correctly. Another use for method aliasing is merging inherited methods with different names. If they both mean the same thing, one or both can be aliased so that they conflict. The conflict must then be resolved by overriding both inherited methods with one that specifies which to use or does something different. This solves the problems

with multiple inheritance of representation discussed in section 2.2.1. We will support method aliasing in TMPL.

In some languages (Beta, for example) a method which is overridden must include a call to the overriding method explicitly (`super`). Other languages require the overriding method to explicitly declare the fact that it is an override. This prevents accidental overrides due to name conflicts. Smalltalk, on the other hand, requires no explicit indication that a method is being overridden or that a method is an override. The problem with making the overridden method explicitly call the overriding method is that you have to know, in advance, that a subclass is going to override the method. Explicitly declaring method overrides poses no such problem. However, a sophisticated programming environment which indicates method overrides in some way, makes this redundant. In TMPL, we will follow Smalltalk's approach. Overriding will be entirely implicit. However, the programming environment will indicate overrides to the user.

4.1.3 PROGRAMMING ENVIRONMENT

Smalltalk's programming environment is amazingly powerful. It allows the programmer to browse existing classes, trace the call graph, inspect objects, step through the execution of his code, do a performance analysis, etc. Trellis managed to improve on Smalltalk's programming environment in two major ways. It maintains a database, so traversing the call graph is much faster than in Smalltalk. In addition, Trellis is a typed language. This reduces the spread of the call graph, since type information limits the possible answers to each query.

For the design of TMPL's programming environment, we will include all tools provided in Smalltalk. TMPL will maintain a database for the call graph. As we will soon see, TMPL is a typed language so we can restrict the call graph using type information. As we have already decided, the programming environment must indicate method overrides to the programmer since he does not explicitly declare overrides as such. Excludes and aliases are also shown. The browser must show the programmer what is actually inherited.

4.1.4 MULTIPLE INHERITANCE

Multiple inheritance has often been called the goto of object-oriented programming. This is a particularly appropriate analogy. The `goto` statement in Pascal is often misused resulting in “spaghetti” code. However, there are cases where goto *should* be used. One such case is for premature exits from subprograms. To avoid the use of a goto in this case, the entire remainder of the subprogram must be enclosed in an `if` statement. If there are several such premature exits, the subprogram will be indented further and further. This is obscuring the normal flow of execution. Goto is a feature which is easily abused but is also quite useful when used appropriately. Multiple inheritance is also easily abused, resulting in such a messy class hierarchy that it becomes impossible to find anything. However, it is also quite useful when used carefully. In single inheritance languages, representation and behavior must often be copied. It is quite easy, once this is done, to forget to duplicate future changes. Today, we take inheritance itself for granted, but there was once a similar debate over this feature. Those who wanted it pointed out that code sharing minimized code size and reduced errors. Those who didn’t want it pointed out that it violated the encapsulation of data types.

The biggest problem with multiple inheritance is not that it can be abused, but that there are many decisions to be made when implementing it. The first and simplest question that must be answered is how to deal with ambiguities. SELF’s solution, a list of sets of superclasses, seems to be the best answer. Unlike a simple list of superclasses, it doesn’t impose an arbitrary order, and unlike a simple set of superclasses, it doesn’t require too many overrides to specify who to inherit from. However, experience using SELF indicates that difficulties still arise. The “sender path tiebreaker rule” [Chambers et al 91] is really too complicated for the programmer to understand fully. For TMPL we will use a simple set of superclasses, just as Trellis does.

The second question is how to multiply inherit representation. Since we support method aliasing in TMPL, and instance variables are always referenced through accessor methods, we can use this feature to merge and split instance variables inherited from different superclasses. Thus multiple inheritance of representation poses no difficulties.

In a single inheritance system, overriding methods often call the overridden method with a `super` call. This causes the method lookup search to start at the superclass of the class defining the method. Sometimes this is inadequate. In Smalltalk, you often see methods such as `basicNew`. These are only needed because the `super` call does not support naming a superclass directly. In a multiple inheritance system, we need some way to identify each of our `super` methods when we are overriding methods from more than one superclass. The solution to both of these problems is the same: support a *named super call* as well as an unnamed one. In a named `super` call, we specify the name of the class to start searching from explicitly. Multiple unnamed `supers` (as in `super super new`) are a bad idea. Unnamed `supers` allow the programmer to call the inherited method without hard-coding knowledge of the hierarchy. Using multiple unnamed `supers` effectively hard-codes the structure of the hierarchy without naming the elements. Better to explicitly name them so that the system can detect errors caused by changes in the hierarchy.

4.1.5 MULTIPLE POLYMORPHISM

Multiple polymorphism hasn't received as much attention in the object-oriented programming community as multiple inheritance has. This is probably because people see message passing as being central to object-oriented programming and they don't want to give it up. But as we have seen, multiple polymorphism can be thought of as an extension to message passing. It has several advantages over message passing. The most obvious use of multiple polymorphism is for arithmetic operations. It eliminates the need for case statements on the type of the second argument or double dispatching, both of which complicate the addition of new math classes later. Although double dispatching is quite fast, it produces a large number of methods with non-intuitive names, particularly when extended to more than two arguments. The idea of multiple polymorphism extends quite easily to method patterns allowing optional and repeated arguments, arbitrary argument ordering, and multiple method aliases (these copy rather than rename). Since the first argument is no longer treated specially as the receiver, we don't have to rephrase methods to place the receiver first.

The most obvious disadvantage of multiple polymorphism is that we can no longer think in terms of sending messages to objects; we must think procedurally instead in some cases.

Although we were taught to think of “1 + 2” as sending the message “+2” to “1”, it seems doubtful that many object-oriented programmers actually do think this way about arithmetic. We already think procedurally in this case, so why not act procedurally too? A more important issue is encapsulation. Which class does a multiply polymorphic method “belong” to? The answer is all of them—the method is shared between several classes. The programmer no longer needs to choose an arbitrary argument to be the receiver, nor does he need to define call-throughs for every other argument as receiver. The most important problem with multiple polymorphism is how it complicates method inheritance. Even in a single inheritance system, we get a DAG of methods resulting in ambiguous message sends and we need a more complicated way of calling inherited methods than Smalltalk’s `super send`. In a multiple inheritance system, these problems have already been dealt with (although we must now specify a class for each argument when we call an inherited method). Multiple inheritance and multiple polymorphism work well together for this reason. In TMPL, we will support multiple polymorphism as well as multiple inheritance. We will extend our named and unnamed `super` call to a per-argument basis; i.e., we will allow each argument of a method call to specify `super` or a class to start searching at. Since there is no “defining class” of a method, we will restrict unnamed `super` to “recursive” calls—calls to another method with the same name. We start searching at the superclass(es) of the class of the corresponding argument.

If we are going to include multiple polymorphism in a language, we must decide whether all arguments are polymorphic or only those that the programmer indicates as such. If all arguments are polymorphic then methods such as `displayAt:` end up in the `Point` class as well as in the geometric object classes. In a system where the programmer indicates which arguments are polymorphic, the programmer would mark only the first argument as polymorphic to get normal message sending behavior. For arithmetic operators, he would mark both arguments. Generally, major arguments are marked and minor arguments are left unmarked. The problem with this approach is that the programmer must make an arbitrary decision in many cases, much like the C++ programmer’s decisions as to which functions should be virtual. For TMPL, we will make all arguments polymorphic. We will extend multiple polymorphism to allow method headers to be arbitrary regular expressions. As we have seen, this makes it easy to specify optional arguments, etc.

4.1.6 CLASSES OR PROTOTYPES?

We have seen how prototype-based systems are conceptually simpler than class-based systems. Rather than having classes which describe what their instances look like and how they behave, we just have objects. SELF manages to use inheritance to achieve many goals that would normally require explicit language support. The main disadvantage of prototype-based systems is that they provide a less structured environment to program in. Conventions must be adopted in order for code to be readable. SELF's traits and categories are examples of such conventions. There is one place where convention is not sufficient—the type hierarchy. When we are describing specifications, we want to ensure strict conformance to the contract they represent. Although prototypes are conceptually simpler from the language designer's point of view, they are actually more complex from the programmer's point of view. Classes provide a framework within which everything is structured in a uniform way.

One problem with classes is easy to alleviate. We often define classes which only have a single instance. Examples in Smalltalk are `UndefinedObject`, `True`, and `False`. Like Dylan, we will support special one-of-a-kind objects as well as classes in TMPL. These special objects can define representation and behavior, just like classes. To get class behavior, we will need metaclasses. Here, we will use Smalltalk-80's approach. Each class will be an instance of a corresponding metaclass which defines the class behavior. Metaclasses can also define class variables and class instance variables.

4.1.7 SEPARATING SPECIFICATION AND IMPLEMENTATION

We must decide whether to separate specifications from implementations in TMPL. The Smalltalk approach is to leave them unified into a single entity: a class. We can define several implementations of the same specification by defining an abstract class with the specification and any simple non-primitive¹ behavior that can be defined in terms of the operations provided by the concrete subclasses. We can define several specifications for a single implementation

¹The word "non-primitive", as it is being used here, means that the method does not access representation directly, although it may call other "primitive" methods.

using encapsulators. However, encapsulators have many disadvantages. They add an extra level of indirection, taking up both time and space. Whenever new objects are created (by non-destructive operations for example), we must build new encapsulators around them. Encapsulators can define their own code which “polishes” the low-level methods provided by the implementation. However, this code can rarely be reused (inherited).

Another approach, discussed in section 2.2.7, is to separate the specifications and implementations into separate hierarchies but to make these hierarchies share common elements. The specification hierarchy can then be a formal one while the implementation hierarchy can be based on convenience and reuse. This is the approach we will use in TMPL. We will support multiple inheritance in both hierarchies. The headers of methods define the public interfaces of the classes. Subtypes must obey subtype compatibility rules. They must implement all operations defined in the supertypes and they must be co/contravariant.

4.1.8 TYPING

In a typed language, all variables (including instance variables, arguments, results, temporaries, etc.) have a declared type. Since we still want to have dynamic typing, any object which conforms to the declared type of a variable or any (recursive) subtype may be stored in that variable. Since TMPL is multiply polymorphic, all argument types are implicitly type checked by the method lookup. Result types and assignments must be type checked.

There are two advantages to typing: speed and reliability. Neither of these advantages has been proved but typing certainly doesn't make code slow or unreliable. The only disadvantage to typing is the time it takes to declare variables. Making the default type of variables be `Object` eliminates this problem during rapid prototyping. However, we will still get type errors if we try to use a variable which is automatically declared to be of type `Object` since `Object` does not implement all methods. To avoid this problem we will use *fuzzy typing* in TMPL [van Gulik 93b]. In such a system, method calls are legal so long as they *may* succeed. The possibility of a run-time error does not make the method call invalid at compile-time, although a warning will be issued.

Union types are convenient, particularly the common `aTypeOrNil`. Without them, there is a tendency to define a common ancestor type even when this is not appropriate. A union type can be thought of as an implicitly declared common ancestor type. Union types are a special case of arbitrary regular expressions in method headers.

Named super calls can be used to make assertions about the types of the arguments. This is useful when the programmer wants to declare the type of an intermediate expression without assigning it to a temporary variable.

4.1.9 SYNTAX

The syntax chosen for TMPL was loosely based on Smalltalk's syntax. Smalltalk defines three types of methods and corresponding message sends: unary, binary, and keyword, in decreasing order of precedence. For example,

```
i mod: 2 + 3 negated
```

parses as:

```
1 mod: (2 + (3 negated)).
```

At a single precedence level, parsing goes left to right. The beauty of this syntax is that it makes parentheses almost unnecessary. The keyword before each argument normally identifies that argument in some way. Often keywords (and variable names) are concatenations of several words, using case changes to denote word breaks. For example:

```
anOrderedCollection includesAnySuchThat: [:anElement | anElement isNil]
```

One shortcoming of this syntax is that it always starts with the receiver and always ends with an argument (except for unary messages). A Smalltalk programmer is forced to write method names such as:

```
indexOfOrZero: (anElement)
```

rather than:

```
indexOf: (anElement)orZero
```

Since TMPL is multiply polymorphic, all arguments should be treated equally. For the TMPL syntax, we chose to allow any keyword methods to start and end with a keyword. We also allow several keywords or arguments in a row. This improves the readability of methods

because there are almost no constraints. Since the first argument is no different than the others, the method can be arranged to read as a sentence. To parse this syntax, we need a way to distinguish between keywords and identifiers (variables). We will use case rather than Smalltalk's colon. Keywords will be entirely lowercase. Identifiers must have at least one uppercase letter. By convention, globals (such as class names) will be capitalized. Temporary variables and arguments will start in lowercase but must contain an uppercase letter. This makes single letter names for these variables impossible—not a significant problem since using such variable names is bad style anyway. An example of the proposed syntax for TMPL would be:

```
anArray. Array at anIndex Integer if absent anAction Block
```

Other minor improvements to Smalltalk's syntax will be made. Binary operators in Smalltalk consist of one or two special characters (such as '+'). This caused the "not identical" operator to be called `==` rather than `===`. The limit of two characters will be eliminated. We distinguish between procedures and functions, both for methods and for blocks. A procedure call cannot be used as an expression and a function call cannot be used as a statement. To throw away an unwanted result, we pass it into an `ignore` method which does nothing. We support explicit returns from blocks as well as methods, even when no result is being returned. We allow accented characters, question mark and exclamation point to be used in both keywords and identifiers. This is particularly useful for predicates and boolean variables.

Since variables in TMPL are always accessed through methods, we define two pieces of syntactic sugar to make this easier:

```
anObject.variableName is syntactic sugar for (anObject variablename).
```

```
anObject.variableName := aValue is syntactic sugar for (anObject variablename aValue).
```

This dot syntax will normally be used for variable access, but this is not enforced. A method which acts like a variable can be called using the dot syntax. The first syntax requires a result and the second must not return one. Note that dot expressions have the highest precedence, higher even than unary messages. Also note that the keyword `need` need not be in uppercase since it is clearly a keyword from the context in which it appears. This means we can keep the mix of upper and lowercase used to indicate word breaks.

4.1.10 SUMMARY

- Class-based but supports special objects with representation and behavior. Examples would be `true`, `false`, and `nil`. Classes form a class hierarchy based on the superclass/subclass relation. This hierarchy is a DAG. Classes also form a type hierarchy based on the supertype/subtype relationship. This is also a DAG. There is a root class called `Object` which defines no behavior. The subclass `Default` defines common behavior for copying, comparing, etc.
- Methods have a name, no receiver, can accept zero or more arguments, and possibly return a result. All parameters and the result, if any, are typed. Details of typing are summarized below. Methods can be declared to be public or private to their module; the default is public. Private methods are inheritable. Method headers are regular expressions, allowing multiple names for the same operations, optional parameters, and repeated parameters. These are treated as multiple (possibly an infinite number of) method definitions.
- Classes define instance variables. All references to them are through their accessor methods which are implicitly declared. These methods can be made public or private individually. They are typed according to the declared type of the instance variable. If there is no way an inherited instance variable can be accessed, due to overrides or excludes, then space is not allocated for this variable.
- Method inheritance occurs when two methods have the same name and the corresponding parameter types are non strict subtypes. Simply defining the submethod will cause an override. The submethod can also be declared to be excluded or aliased to another method name. The programming environment must indicate that excluded methods are not inherited and that aliased methods are inherited under the new name only.
- All methods which can accept an instance of a particular class (or a special instance) as an argument are considered methods of that class (or special instance). The headers of the methods define its interface. This must conform to the interfaces of all the supertypes of the class (or special instance). These supertypes are unordered. The subtype compatibility rules require all public methods in the supertype to exist in the subtype and to obey co/contravariance.
- All parameters, result types, instance variables, and temporary variables are typed. Types can be any class or special instance, or a union of them. A shorthand is used for “`!nil`” since it is so common. Method lookup ensures all arguments are type safe, but results must be type-checked. Assignment must also do type checking. It is legal to call a method even if the declared types of the arguments are not sufficient to ensure the existence of an appropriate method.

- Method calls normally specify a method name and all the arguments. Each argument can include an indication to start the method lookup at a particular type. If the method call is being made from another method with the same name, then we can use `super` rather than specifying a particular type. This causes the search to start at the supertype(s) of the type of the corresponding parameter in the calling method.
- All names are scoped within a module. Modules have full access to everything defined in other modules that they declare as super-modules. They can only access public elements of modules declared as being used. All methods and classes exist globally but their visibility is determined by module scopes.

4.2 IMPLEMENTATION

A prototype of TMPL was implemented in ParcPlace Smalltalk-80 version 4.1. The feature set was reduced somewhat to simplify the implementation. Modules and privacy were not implemented. A browser and a crude debugger were implemented but a full programming environment was not. Regular expression method headers were not implemented but union types were. The implementation is a prototype; its purpose is to discover how these features interact and what difficulties there are in implementing and using each feature. The implementation also verifies the design decisions.

4.2.1 PARSER

The first stage in implementing TMPL is to build a scanner and parser. We chose to hand-code the scanner and parser rather than using a parser generator because the syntax is simple enough that parsing is quite straightforward. Rather than making the parser a subclass of the scanner, as Smalltalk does, we made the parser contain the scanner in an instance variable. The scanner reads from a stream of characters and maintains a single token look ahead buffer. The parser reads from the scanner and generates parse tree nodes. Originally, we required keywords to be entirely lowercase. Identifiers were required to have at least one uppercase letter in them. The problem with this syntax was readability. It was difficult for the eye to parse. We decided to make keywords entirely uppercase to avoid this problem. This may be unnecessary; simply putting the keywords in bold might prove adequate for readability and would be easier to type. Before we built the parser, we formalized the syntax by defining a BNF grammar for TMPL.

The syntax we will use for defining the grammar is as follows:

- Non-terminals in angle brackets; eg., <non-terminal>.
- Terminals in single quotes in this font; eg., 'CLASS'.
- Optional elements surrounded by square brackets; eg., ['e' <integer>].
- Options (logical or) are separated by vertical bars; eg., <option1> | <option2>.
- Ranges of options are separated by ellipsis; eg., 'a' .. 'z'.
- Curly braces are used for grouping; eg., {<option1> | <option2>}.
- An asterisk as a suffix denotes zero or more repetitions; eg., <digit>*.
- A plus as a suffix denotes one or more repetitions; eg., <digit>+.
- Text in italics is commentary—it is not part of the grammar; eg., *grammar comment*.

The scanner reads tokens. It ignores comments which are delimited by double quotes or (possibly nested) curly braces. Examples are "comment" and {comment {nested}}. Whitespace is ignored except after a period, where it indicates a statement separator and between a minus sign and a digit where it makes the minus negate the numeric literal rather than producing a subtraction operation.

The types of characters are defined as follows:

- <character> ::= any character, including 16 bit extended characters such as accented letters
and including whitespace such as tab and carriage return
- <lowercase letter> ::= 'a' .. 'z' | 'é' | 'u' | ... *accented letters included*
- <uppercase letter> ::= 'A' .. 'Z' | 'É' | 'Ü' | ... *accented letters included*
- <letter> ::= <lowercase letter> | <uppercase letter>
- <digit> ::= '0' .. '9'
- <extended digit> ::= <digit> | 'a' .. 'f' | 'A' .. 'F'
- <identifier character> ::= <letter> | <digit> | '?' | '_'
- <operator symbol> ::= '+' | '-' | '~' | '=' | '*' | '/' | '@' | ':' | ';' | '<' | '>' | '&'
- <quote> ::= a single quote
- <non-quote character> ::= any <character> except <quote>
- <dot> ::= '.' which is followed by a letter
- <period> ::= '.' which is not followed by a letter

These characters are formed into words as follows:

<keyword> ::= <uppercase letter> { <uppercase letter> | <digit> | '?' | '!' }*
must be entirely uppercase

<identifier> ::= <lowercase letter> <identifier character>* |
<letter> <identifier character>* <lowercase letter> <identifier
character>†
must contain at least one lowercase letter, case insensitive

<operator> ::= <operator symbol>+
must contain only operator symbols, may be any length

Literals in TMPL are much like in Smalltalk:

<string literal> ::=
<quote> { <non-quote character>* | { <quote> <quote> } }* <quote>
can contain paired quotes to indicate embedded quotes

<character literal> ::= '\$' <character>

<integer literal> ::= ['-'] <digit>+ [{'r' | 'R'} ['-'] <extended digit>+]
can include base, no space after minus sign, sign can go in either position

<real literal> ::= ['-'] <digit>+ '.' <digit>* [{'e' | 'E'} <integer literal>]
can't start with decimal point, no space after minus sign or decimal point

<number literal> ::= <integer literal> | <real literal>

<literal> ::= <string literal> | <character literal> | <number literal>

Expressions are parsed using the precedence rules described in section 4.1.9. Note that super calls are not supported when using the dot notation. Simply use the corresponding keyword call when supers are needed.

<variable reference> ::= <identifier>
refers to an argument or a global or local variable

<super search> ::= '.' ['super' | <union type>]
affects containing method call

<simple expression> ::= <variable reference> | <literal> | '(' <expression> ')' | <block>
highest precedence

<super simple expression> ::= <simple expression> [<super search>]
simple with optional super — super applies to parent

`<dot call> ::= <dot expression> <dot> {<keyword> | <identifier>}`
syntactic sugar, keyword can be in any case

`<dot expression> ::= <simple expression> | <dot call>`
high precedence, no super on dot calls

`<super dot expression> ::= <super simple expression> | {<dot call> [<super search>]}`
dot with optional super—super applies to parent

`<binary call> ::= <super binary expression> <operator> <super dot expression>`
super on binary calls since no other syntax possible

`<binary expression> ::= <dot expression> | <binary call>`
low precedence

`<super binary expression> ::= <super dot expression> | <binary call>`
binary with optional super, super binds tight

`<keyword call> ::= {<keyword> | <super binary expression>}+ <keyword>`
`{<keyword> | <super binary expression>}+`
at least one keyword

`<expression> ::= <binary expression> | <keyword call>`
lowest precedence

Statements are distinguished from expressions. We require expressions to return a result whereas we don't allow statements to.

`<return statement> ::= { 'return' | 'return' } [<expression>]`
returns from a block or method procedure or function

`<assignment statement> ::= <variable reference> '=' <expression>`

`<dot assignment statement> ::= <dot call> '=' <expression>`
syntactic sugar, keyword can be in any case

`<statement> ::=`
`<keyword call> | <return statement> | <assignment statement> | <dot assignment statement>`
must do some action

Types and type declarations are defined as follows:

`<simple type> ::= <variable reference>`
a reference to a class

`<union type> ::= <simple type> { '|' <simple type> }+`
a possibly union type

<variable declaration> ::= <variable reference> ':' <union type>

a variable and its type

<variable declarations> ::= <variable reference> {'<variable reference>'}* ':' <union type>

a list of one or more variables and their type

Primitives are Smalltalk methods. They are called as follows:

<primitive expression> ::=

'<#>' <binary expression> <string literal> <binary expression>* '^' <union type> '#>'

receiver, message, arguments, result type

<primitive statement> ::=

'<#>' <binary expression> <string literal> <binary expression>* '#>'

receiver, message, arguments, must return nil

Blocks and methods are built out of sequences of statements:

<statement list> ::=

{<variable declarations> <period>}* [<statement> <period>]* [<statement>]

optional local variables with zero or more statements separated or terminated by periods

<block> ::= '[' ['{<variable declarations> [<period>]}* ')] ['^' <union type> [<period>]]

<statement list> '['

can have arguments as well as locals

<method header> ::=

'(' <variable declaration> <operator> <variable declaration> ')' '^' <union type>

['(' {<keyword> | <variable declarations>}* <keyword>

{<keyword> | <variable declarations>}* ')' ['^' <union type> [<period>]]

binary or unary/keyword, at least one keyword

<method> ::= <method header>

['->' {<binary call> | <keyword call>}]

<statement list>

optional alias clause indicates method being renamed by calling it.

override if there is a body

If a method includes an alias clause, the method is an *alias method*—a locally renamed version of some target method. The super binary or keyword call in the alias clause identifies this target method. The target method will be treated as a supermethod of the alias method. If the

alias method has a body, it is treated as an override of the target. This is useful for merging inherited instance variables with different names since we want to actually keep one or the other of the variables. Note that we may want to alias a target and override it with an empty method. To do this, we simply call a “do nothing” method to make the body non-empty.

Classes and special objects are defined as follows:

```

<class definition> ::= ['ABSTRACT'] 'CLASS' <identifier>
    'SUPERCLASSES' <variable reference>*
    'INSTANCE VARIABLES' <variable declarations>+
    ['CLASS VARIABLES' <variable declarations>+ ]
    ['CLASS INSTANCE VARIABLES' <variable declarations>+ ]

<special object definition> ::= 'SPECIAL OBJECT' <identifier>
    'SUPERCLASSES' <variable reference>*
    'VARIABLES' <variable declarations>*

```

The programmer distinguishes between abstract classes (those without instances) and concrete classes (those with instances) for reasons we will come to later. Also, we do not have both supertypes and superclasses as our design indicated, also for reasons we will come to later. Special objects can inherit from other special objects and from classes. Classes can only inherit from other classes. This prevents a class’s instance behavior, and class behavior from being merged into a special object and then split apart again in a subclass. This would be difficult because the special object does not distinguish between instance and class behavior [Gregory 93].

A few examples should help clarify the syntax. Consider the following:

```

(NOT t: True) ^^ Boolean
    ^^ False

(a: Rational = b: Rational) ^^ Boolean
    ^^ (a.numerator * b.denominator) = (b.numerator * a.denominator)

(NEW aClass: class List) ^^ List
    {Lists are primitive Smalltalk objects.}
    ^^ <#aClass 'newSmalltalkObject' ^ List#> "primitive call"

```

In the first example, we have a method which starts with a keyword and has a single argument of type True (a special object under Boolean). Note the double caret symbol is used to indicate

the return type of a method as well as the “return from method” command. The second example shows how binary methods are defined and called. Note the use of dot calls to access the instance variables of the `Rationals`. The last example shows how class arguments are specified. The `NEW` method accepts the class `List` as an argument rather than an instance of the class `List`. Note the two types of comments and the primitive call. Unlike Smalltalk, primitive calls are not associated with the method—they are ordinary expressions. In this case, the Smalltalk object representing the argument (the TMPL class `List`) is being sent the message `newSmalltalkObject` with no arguments. The result is expected to be of type `List`.

Smalltalk’s exception mechanism was used several places in the parser. In many cases, more than a single token of look ahead is required. When this happens, we can try parsing one way and try again another way if the first way fails. Exception handlers are also used to rename exceptions to more appropriate names. For example, when parsing for an expression, we first parse for a simple expression. If none is found, we report “simple expression expected”. In this case, we rename the exception to “expression expected”.

4.2.2 METHOD HIERARCHIES

Now that we have a scanner and a parser, we can generate a parse tree for each method. Before we can successfully execute this parse tree, we must store the parsed methods into some structure that allows us to do method lookups. Since TMPL is multiply polymorphic, we do not store methods in classes. Instead, we divide methods in hierarchies for each method name. Each method hierarchy is a DAG of methods in which more general methods are ancestors of more specific ones. A method *a* is at least as general as another method *b* if *b* accepts arguments at least as general as *a* does. This relation meets all the requirements to be a valid partial ordering. We define a most general root method and a most specific leaf method to avoid special cases in the insertion and deletion code. To store a method into such a hierarchy, we find all methods which are at least as general as the method being stored. We eliminate any methods which are more general than ones already included. These will be the parents of the new method. We then do the opposite operation: we find all methods which are at least as specific as the method being stored, eliminating methods which are more specific than others already included. These will be the children of the new method. The parents and children will

form a bipartite graph. These links must be broken before the new method is connected to the parents and children. For example, in Fig. 4.1, Rational + Rational is being added causing the dashed links to be broken. If the method being inserted already exists, it is overwritten.

Figure 4.1. Inserting into method hierarchy.

To lookup a method at run-time, we start by going to the method hierarchy for methods with the appropriate name. We then search down from the root for all methods which accept the arguments. The root will always accept the arguments since it is the most general method, by definition. As in the insertion algorithm above, we then eliminate methods which are more general than others already included. When we are done, we have the most specific methods which accept the arguments. If there is exactly one method, we are done. Otherwise, we raise a signal indicating a method call failure. If there are several methods which accept the arguments, we indicate an ambiguous call. If there are none (or just the most general root method which doesn't actually exist), then we indicate the equivalent of Smalltalk's "message not understood" error.

There are two ways to handle union types. One is to treat union types as more general types. Think of $a \mid b$ as an unnamed supertype of a and b . The other approach is to treat union types just like regular expressions in method headers. Think of $\{a \mid b\}$ NEGATED as a NEGATED $\mid b$ NEGATED. The advantage to the first technique is that the methods stored are what the user typed verbatim so that the browser is more intuitive. The advantage of the second technique is that it generalizes to full regular expressions in method headers. To keep the user interface intuitive, the actual method can be left with the regular expression header but the nodes in the

method hierarchies can include their own "instance" of the regular expression: a simple header. There is a semantic difference as well. If we store union types, then we can use (named or unnamed) super calls to invoke methods accepting union types. We chose to store union types for this reason.

4.2.3 INTERPRETER

Our prototype implementation is a true interpreter, so we need to define methods for each type of parse node to execute it in a given environment. Before we can do this, we must define an environment. Each environment points to a parent environment in which to continue the search. This is needed to allow arguments and temporary variables to lexically override global variables. It is also needed for lexically scoped blocks. Each environment defines bindings from variable names to values. The type specified in the variable declaration is also stored to allow type checking to be done when values are stored into environments. The values are instances of a Smalltalk class `TmplObject` which defines the representation of TMPL objects. TMPL objects are an instance of some TMPL class and they store a set of instance variables. A class can inherit instance variables with the same name from each of its superclasses without conflict using method aliasing to rename one or both of the accessor methods. Consequently, there can be more than one instance variable with the same name. In order to distinguish between them, we store an object's instance variables under both its name and its defining class. Class instance variables are also stored this way. Class variables need not store the defining class since they only exist once.

We represent classes and other special TMPL objects as instances of various subclasses of `TmplObject`. For example, TMPL classes are instances of `TmplClass`, a subclass of `TmplObject` which defines instance variables for superclasses, etc. Therefore, TMPL code cannot access the attributes of a class except through primitives. The alternative to this approach is to represent everything directly in TMPL, but this makes bootstrapping the system extremely difficult.

The default value of all TMPL variables is Smalltalk's `nil`. This is treated as a special "uninitialized" value. It is always a legal value but it is an error to read it. There is no way to

test if a variable has been initialized since the programmer should assign some initial value to compare with. This makes it unnecessary to declare all variables as `aType | Nil` so we chose not to define a special syntax for this.

We can now define an `evaluateIn:` method for each type of parse node. Literals are self-evaluating, so they just return their value. However, the value must be converted into a TMPL object. To do this, we evaluate a piece of TMPL code to generate a new TMPL object of the appropriate class. We then store the value into this new object. Most literal types are primitive, so their value is stored in a Smalltalk instance variable which is accessed through primitives. Blocks are really literals. Evaluating them results in a full closure: a block and the environment in which it was defined. Evaluating the resulting full closure (which is done by a block evaluation primitive) actually executes the statements in the block in a sub-environment with the formal parameters bound to the actual parameters. Evaluating a method call causes all the arguments to be evaluated. The types of the arguments are then used to do the method lookup, as discussed in section 4.2.2. This method is then evaluated in a new environment with the formal parameters bound to the actual parameters.

Figure 4.2. The top of the TMPL hierarchies.

We can “file in” most of the TMPL code for our system, but we need to hand wire the top of the instance-class and inheritance hierarchies since everything is dependent on it, even itself.

This is how we bootstrap the system. Compare the TMPL hierarchies shown in Fig. 4.2¹ with the Smalltalk-80 hierarchies in Fig. 2.2. Note that the class `Object` has no superclasses. Rather than pointing to `nil`, we have an empty list of superclasses. Since `Class` and `Metaclass` are special objects in TMPL, they do not need corresponding metaclasses. We chose to make `Metaclass` inherit from `Class` rather than being a sibling. As in Smalltalk-80, all metaclasses are instance of `Metaclass`. All special objects, including `Special` itself, are instances of `Special`. Beyond these top-level classes, we need many more classes before the language can function reasonably. At the very least, we need to define classes with primitive Smalltalk representation for all the literal types, including blocks. We must implement many primitives for these classes to be functional. Before we can do any conditionals, we must define a `Boolean` class and `True` and `False` special objects as subclasses of it. We can then define conditional methods entirely within TMPL. They need not be primitive since the method lookup can choose the appropriate method to execute:

```
(IF cond: True THEN then: Block ELSE else: Block)
  EXECUTE then.
```

```
(IF cond: False THEN then: Block ELSE else: Block)
  EXECUTE else.
```

Smalltalk's exception mechanism was also used in the interpreter, both for handling run-time errors and for returning (possibly prematurely) from methods and blocks. When we execute a return statement (either from a block or a method), we simply raise an exception which is handled by the method which is evaluating the block or method. Since blocks are unnamed, we always return from the innermost block.

¹ The text shown in the rounded rectangles is the `printString` of the object. Classes and special objects print as their name. Metaclasses print as their name followed by the word `class`. In all three cases, this text would evaluate to the object.

4.2.4 ONE HIERARCHY OR TWO?

Our design called for two separate hierarchies, one for specifications and one for implementations, both sharing common elements: classes. We found this separation was impossible to maintain. If we consider a method such as:

```
(x: Real + y: Real) ^^ Real
```

we are declaring *x* and *y* to be of type *Real*. But we want this method (its implementation, that is) to be inherited by subclasses of *Real*. So we are really saying that *x* and *y* are of *class* *Real*. But this means that inside this method, we don't know the types of *x* and *y*; we only know their classes. The actual value of *x* could be an instance of a subclass of *Real* which is not a subtype of *Real*. So the programmer cannot be assured of the type safety of the method calls he makes with the arguments. The subtype compatibility rules in the type hierarchy are of little use since they don't guarantee the safety of method calls. Method results must also be classed rather than typed so that we can write:

```
(MINIMUM x, y: Real) ^^ Real  
IF x < y THEN [^x] ELSE [^y]
```

without getting a type error when we return *x* or *y*. Instance variables, class variables, and class instance variables must be classed rather than typed because they are passed as arguments to set methods and returned as results from get methods. This leaves global variables as the only typed entities.

After realizing the above problem, we decided to eliminate the dual hierarchies. This left us with one question to be answered. Should the single hierarchy be a strict one? In other words, do we maintain the subtype compatibility rules? Since TMPL is a typed language, we decided that it was more appropriate to make the single hierarchy a strict type hierarchy rather than the more ad hoc hierarchy used in Smalltalk. Multiple inheritance and multiple polymorphism work well in a strict hierarchy since they reduce the need to violate subtype compatibility rules. In such a strict hierarchy, we require subclasses to define all the methods defined by the superclasses. This makes method excludes illegal so this feature was removed as well. We also require method overrides to be co/contravariant. In Trellis, and Eiffel, arguments must be contravariant and results must be covariant. This causes instance variables to be invariant since they are passed as arguments to set methods and as results from get methods. The receiver is

2 OF 2

PM-1 3 1/2" x 4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

automatically covariant since it must be an instance of the subclass for the method to be executed in the first place. Since TMPL is multiply polymorphic, all arguments are treated like the receiver. They are automatically covariant since a method is treated as an override only if the argument types are at least as specific as those of the overridden method. So only the result type is restricted by subclass compatibility rules. It must be covariant. Since instance variables (as well as class variables and class instance variables) are returned by get methods, they must also be covariant. This is a significant improvement over Trellis's invariant instance variables. For example, we can define `GridPolygon` (with Integer coordinates) as a subclass of `Polygon` (with Real coordinate).

4.2.5 STATIC ANALYSIS

Although our TMPL implementation is an interpreter, we perform some operations on the code statically. This analysis should really be done dynamically but this would require a sophisticated update mechanism to maintain the correct analysis when changes are made to the code. The static analysis mechanism does several things before it returns a list of errors and warnings, much like Trellis's grass catcher. Analysis is actually required before code can be executed because method aliases require analysis to determine which method to use, as we will see in the next section.

The first thing done during static analysis is to generate primitive accessor methods for all instance variables, class variables, and class instance variables. Two methods are created for each variable, a get function and a set procedure. These methods invoke a Smalltalk method using a primitive call. The information given to Smalltalk to identify the variable is the variable name (a string) and the class which defined the variable. As we pointed out in section 4.2.3, there can be more than one variable with the same name without conflict. We distinguish between ordinary methods defined by the programmer and primitive accessor methods which are defined automatically by the system. The latter are not included when code is filed out, etc. If an ordinary method would be *overwritten* (as opposed to overridden) by a primitive accessor method, we report an error (in the existing method) rather than overwrite it.

Each method, including the primitive accessor methods defined earlier, are put through several tests during static analysis. The first test done verifies the subclass compatibility rules. As we pointed out in the previous section, the only compatibility rule in TMPL is covariance of the result type. Each method is first tested to determine whether it is an override. If it is, the method is compared with the overridden method(s). If one is a function and the other is a procedure, we report an error. If both are functions and the result types are not covariant, then we report a compatibility error.

The next static analysis check performed on each method is variable scoping. We look for duplicate variable definitions. If they occur in the same scope, we report an error. If they occur in different scopes so that one lexically overrides the other, we report a warning. We also detect variable references which don't exist in the scope of the reference.

The last thing we do for each method is type inference. We compute the inferred type of everything in a method. The inferred type of a literal is trivial. The inferred type of a variable reference is the declared type of the variable being referred to. This was computed during the scope checks mentioned above. We currently assume globals can be statically typed. The type inferencer uses the actual stored value rather than the declared type which is usually just `Default`. Otherwise declaring a class would require referring to its metaclass before it exists. Computing the inferred type of a method call is more difficult. We first recursively compute the inferred types of all the arguments. We then go through all permutations of these types and determine which methods could actually execute for each. The inferred type of the method call is the union of the result types of these methods. If some are functions and some are procedures, we warn of a potential "result required" or "result unwanted" error. If all of them are procedures, but the call is being used as an expression, or all of them are functions and the call is being used as a statement, then we report an error. If any of the permutations would result in a "message not understood" error or an "ambiguous message" error, we generate a warning. If all of the permutations would result in an error, we report that error immediately.

One of the benefits of doing type inference on all the code is that we can store the set of nodes to start searching from in each method call. At run-time, we know that the desired method will be found below these nodes. In many cases, there is only one possible method so we can

simply store this method in the method call. In a compiled system, this would enable us to *inline* the method call when the method is small. For normal non-super calls, we can inline if there is only one possible method with no submethods. For super calls, we can inline if there is only one possible method and all its submethods do not satisfy the super search criteria specified in the call (so they could never execute).

The system is told which classes are abstract. It can use this information during type inference to continue the search down the method hierarchy when the children cover all the subclasses of an abstract class. For example, when searching for a method with a Boolean argument, if there are submethods for True and False, we can go down to the children. This keeps the system from giving warnings whenever we pass a Boolean in to a method implemented by both True and False but not by Boolean. The search can also continue down to the children when the argument type is a union type and the submethods cover each element in the union. This improves the run-time efficiency slightly. This optimization is not done for super calls. The search must be able to stop at an abstract method¹ in this case because the desired method is often abstract. Type inference of a method call uses all permutations of argument types. This means that any union types are expanded to each component type and any abstract classes are expanded to their concrete subclasses (except for super calls).

The last things done during static analysis before we report the errors and warnings to the programmer is to determine which instance variables are actually required for each class. Because instance variables (as well as class variables and class instance variables) are always accessed through primitive accessor methods which are defined automatically, they can be overridden in subclasses with other methods that replace representation with behavior. (Of course, the overriding methods must conform to the subclass compatibility rules.) If both accessor methods are overridden, then the instance variable does not actually need to be allocated. This is fairly easy to detect. You simply check if the primitive accessor methods would be found if you did a method lookup. But the primitive accessor methods can also be

¹An *abstract method* is a method which has an abstract class as one of its argument types.

called via a super method call or aliased to a different name. These situations are harder to detect.

The solution is to go through all method calls and find those which could conceivably execute a primitive accessor method. Then only allocate instance variables which are actually accessible. To do this, we look at the stored search nodes and all their descendents. For super calls, we can eliminate descendents which do not satisfy the super search criteria specified in the call. For each such method, we can determine the possible classes of the “receiver”¹. The receiver must be of the declared argument type in the method. But it must *not* be of the declared argument types of any of the submethods or else they would execute instead. Super calls must be treated differently. The “receiver” type cannot be determined from the method which executes, so we can only infer the type normally. We know that the instance variable being accessed by this primitive accessor method must be allocated for each of the possible “receiver” classes.

This technique has the advantage that instance variables which aren’t being used for any reason are not allocated. For example, the `Integer` class is a subclass of `Rational` which defines numerator and denominator instance variables. `Integer` should really override the primitive accessor methods as:

```
(me: Integer NUMERATOR) ^^ Integer
    ^me.

(me: Integer NUMERATOR value: Integer)
    ERROR 'Can't set numerator of an integer'.

(me: Integer DENOMINATOR) ^^ Integer
    ^1.

(me: Integer DENOMINATOR value: Integer)
    ERROR 'Can't set denominator of an integer'.
```

This would make the inherited instance variables inaccessible. But even if these methods weren’t included, so long as no methods tried to access the numerator and denominator of an integer, none would be allocated. Note that we have run into the problem mentioned in section

¹The “receiver” is the first argument—the object whose instance variable is being read or written.

2.2.8. Although an integer “is a” rational, Integer is not really a valid subtype of Rational. We are forced to exclude methods by defining them as errors.

If only one accessor is overridden, then the instance variable can actually be removed. If the get method is overridden then the set method can be made into a no-op. If the set method is overridden then the get method can be made to return the uninitialized value since the value could never be changed to anything else.

4.2.6 METHOD ALIASING

In a language with message passing, the semantics of method aliases are fairly straightforward. Each method that a class inherits can be renamed locally. If a method *a* is inherited from a superclass and renamed to *b*, then other local methods (methods defined in this class rather than inherited methods) must refer to the method under the name *b*. As we decided in section 4.1.2, our aliases must rename rather than copying. These semantics for single polymorphism do not generalize to multiple polymorphism since we can no longer refer to “other local methods”. We need some other criterion for determining which name, *a* or *b*, must be used in any given method call. Our solution is to use the inferred argument types. As we saw in section 4.2.1, method aliases are specified by defining alias methods. An alias method contains a super call to the method being renamed. This method accepts more general arguments¹. During static analysis, we use type inference to determine the possible argument types to each method call. If we can determine that the arguments will be accepted by the alias method, then we require the call to use the alias method’s name. Otherwise, we require the call to use the super method’s name. This eliminates the apparent conflict between method aliasing and multiple polymorphism. This solution is due to [van Gulik 93a].

Alias methods are stored the same way as normal methods. An alias method is inserted into the method hierarchy for its name, not the super method’s name. The only difference is that a special link is added from the alias method to the method being renamed. During method

¹Actually, any given argument need only be at least as general, but at least one argument must be strictly more general.

insertion and lookup, we treat these links differently. This is what gives us the semantics described above. During method insertion these links are ignored. We stick to a single method hierarchy. When we look up search nodes during static analysis, we follow alias links to other hierarchies, but when we are done, we throw away any nodes in different hierarchies. This allows us to follow an alias link back to the original hierarchy. At run-time, we follow alias links but we give them a higher priority than normal links. We always choose alias children over other children. If an alias method has no body, we treat it as if it were the method being renamed. If the alias method has a body, then this code overrides the renamed method so we execute it instead of calling the supermethod.

As noted in the previous section, during static analysis, we replace abstract methods with their submethods, except for super calls. This optimization is also not done when there are aliases of the method because it would make it too difficult for the programmer to determine which alias to use. Just because there are submethods that cover all the cases doesn't mean you want to require the method call to use the original name rather than the alias for the abstract method. The static analysis of the code determines which method is being aliased by computing the super call. The alias link is then added. Tests are done to ensure that this is a legal alias. For example, the number of arguments must match. The method being renamed must be strictly more general than the alias method, although any given argument need only be at least as general.

Using an unnamed super call to specify the method being renamed allows the super method to change automatically when a class is inserted into the class hierarchy. Unnamed super calls are always supposed to call methods with the same name as the current method to ensure that the arguments match. But in the presence of method aliasing, this rule is too strict. So long as the method being called is a super method, it must be legal.

4.2.7 PROGRAMMING ENVIRONMENT

Figure 4.3. The TMPL browser.

The TMPL programming environment is centered around a browser (see Fig. 4.3). The top left pane lists the class hierarchy in outline form. Since this is a DAG, classes can appear more than once. The first occurrence of a node will include its descendents. Subsequent occurrences will leave out the descendents and show an ellipsis. The last item in this pane is a special one call NO ARGUMENTS. Clicking on a class lets you browse methods that accept instances of the class. Clicking on NO ARGUMENTS lets you browse methods which accept no arguments. Selecting nothing in this pane lets you browse all methods in the system. The buttons below the class list let you toggle between instance methods and class methods (methods of the metaclass). The buttons to the right of these toggle between local methods and

inherited ones. In `inherit` mode, all methods which can accept an argument of the selected class (or metaclass) are shown. In `local` mode, only methods which accept an argument of the selected class and not one of its superclasses are shown. The middle pane and the right pane together allow the programmer to browse through the set of methods selected with the left pane. The middle pane lists method names as they are stored. Argument positions are shown as colons and adjacent keywords are separated by underscores. The right pane lists the methods with the name chosen in the middle pane. The header is shown for each method. Argument names are not shown; only their types are. Submethods are listed under supermethods in an outline. The method hierarchy is a DAG so the piece of it shown in the right pane may be a DAG as well. The same technique used for the class pane is employed here to show a DAG in an outline.

Using the menu in the class list pane, the programmer can reinitialize the TMPL system. This regenerates the root level classes and then files in the standard library of classes and methods. This library includes all the essential classes. The programmer can also save the current state of the system as the standard library. This output, as well as the output produced when code is filed out, are in superclass order¹. The menu in the method list pane allows the programmer to browse senders of the selected method or possible implementors of method calls in the selected method. These take advantage of the information saved during static analysis to limit the size of the resulting set of methods based on type information. For example, `calls` uses the stored search nodes in the selected method call to determine the possible methods that could execute. To browse implementers, the programmer simply de-selects the selected class to show all methods in the entire system. Since a particular method name is still selected, all implementers appear in the right pane.

The original source code is saved in method nodes and in class definitions so that comments and formatting are maintained. The (arbitrary) order of superclasses, instance variables, etc. is then left alone. The programmer can include comments after each variable as well as general class comments. This is a better technique than maintaining a separate class comment as

¹All of a class's superclasses are filed out before the class.

Smalltalk does because the instance variable comments cannot get out of sync with the actual instance variable declarations. Also, the programmer is more likely to read the comment if it appears with the definition. A method can be formatted by simply throwing away the stored source code and regenerating it by pretty-printing the parse tree. The pretty-printer implemented is fairly crude, but is sufficient for the purpose.

When the programmer selects static analysis, an entire static analysis is done. This takes several minutes with the standard library. Some optimization has been done. As we mentioned earlier, in the long run the static analysis mechanism should be made dynamic, so this would not be a problem. The result of static analysis is a list of errors and warnings. See Fig. 4.4.

Figure 4.4. Error list produced by static analysis.

A proper TMPL debugger was not implemented, but a simple addition to the standard Smalltalk debugger helps when debugging TMPL code. A menu item was added to the standard debugger that scans the stack for method and block execution. These contexts are used to build an `OrderedCollection` that represents the TMPL stack. See Fig. 4.5. Each TMPL stack frame includes the actual method or block executing and the values of all the arguments.

Figure 4.5. Debugging TMPL code.

4.3 USING TMPL

In order to verify our language design, a small class library was implemented in TMPL. All the necessary primitive classes were implemented first so that the system would be functional. This includes all literal classes, including `Block`. A small set of numeric classes and a more extensive set of collection classes were then implemented.

4.3.1 NUMERIC CLASSES

In Smalltalk-80, the relationship between `LimitedPrecisionReal`, `Fraction`, and `Integer`, is based on implementation. Ignoring the lack of precision and range, we could say that `LimitedPrecisionReal` can represent any number on the number line. `Fraction` can represent any rational number and `Integer` can represent any integer on the number line. So a more logical hierarchy would place `Integer` under `Rational` (a more appropriate name), which is itself under

Real (since the lack of precision is being ignored). We will use this hierarchy for our TMPL class library. In Smalltalk-80, the superclass of all three of these classes is **ArithmeticValue** which is under **Magnitude**. An **ArithmeticValue** is anything that supports $+$, $-$, $*$, and $/$. A **Magnitude** is anything that supports $<$.

For our TMPL class library, we would prefer more formal concepts. A **Group** is a set of elements which can be added [Bloch 87]. A **Group** has a "zero" which is the element which has no effect when added to other elements. We define subtraction as the inverse function of addition. A **Ring** is a subclass of **Group** which defines multiplication as well as addition. It has a "one" which is the element which has no effect when multiplied by other element. As for **Groups**, division is defined as the inverse operation. A **Group** requires very little of its addition operation: it must be associative but need not be commutative¹. However, a **Ring** requires its addition operation to be commutative and its multiplication operation must distribute over its addition operation². They do not require any inequality operation so these classes do not inherit from **Magnitude**. In fact, **Magnitude** can also be formalized. We replace it with two classes: **PartiallyOrdered** and **FullyOrdered**. The instances of a subclass of **PartiallyOrdered** can be compared and the result will be one of the following: $<$, $=$, $>$, or *incomparable*. An example of such a subclass would be a tree or DAG node. The comparison operation would indicate whether one node is an ancestor or descendant of another. They would be *incomparable* if they did not have such a relationship. Partial orderings must be reflexive, antisymmetric, and transitive³. **FullyOrdered** is a subclass of **PartiallyOrdered** which eliminates the possibility of *incomparable* elements. Example of full orderings are common; numbers, characters, and strings being obvious examples. Thus our numeric classes are as shown in Fig. 4.6.

¹Associative law: $(a \sim b) \sim c = a \sim (b \sim c)$. Commutative law: $a \sim b = b \sim a$. Both are valid for numeric addition and multiplication.

²Distributive laws required for rings: *left* $a * (b + c) = (a * b) + (a * c)$ *right* $(a + b) * c = (a * c) + (b * c)$. Addition need not distribute over multiplication: *left* $a + (b * c) = (a + b) * (a + c)$ *right* $(a * b) + c = (a + c) * (b + c)$. Both are valid for booleans but only the first is valid for numbers.

³Reflexive law: $a \leq a$. Antisymmetric law: $a \leq b$ and $b \leq a$ implies $a = b$. Transitive law: $a \leq b$ and $b \leq c$ implies $a \leq c$.

Figure 4.6. The TMPL numeric classes.

The classes `Group`, `Ring`, `PartiallyOrdered`, and `FullyOrdered` are abstract classes, providing only behavior for their subclasses. `Reals` and `Integers` are represented as instances of a special subclass of the Smalltalk class `TmplObject` which includes an extra instance variable for their value. Special primitives are used to create and manipulate this value. `Rationals` are represented as ordinary TMPL objects with two instance variables: `numerator` and `denominator`. `Group` defines a class instance variable for each subclass's "zero" and `Ring` defines a similar class instance variable for each subclass's "one". These are initialized to 0.0, 0/1, 0 and 1.0, 1/1, 1 respectively.

We can obviously take advantage of multiple polymorphism in our numeric classes. We can support cross-class operations so that implicit coercion takes place. For example:

```
(x: Integer + y: Real) ^^ Real
^^ (CONVERT x TO Real) + y
```

Note how we chose to implement explicit conversion. Rather than having a method name such as `asReal`, we simply pass the class, as well as the object to be converted, to the conversion method, taking advantage of multiple polymorphism once again.

4.3.2 COLLECTION CLASSES

The set of collection classes defined in Smalltalk-80 [Goldberg & Robson 89] is quite large but there are still some fairly obvious extensions. Some other collection classes that would be

useful are a `Dictionary` with duplicate keys, a `SortedCollection` without duplicates, and a `Dictionary` with sorted keys. As we did in the previous section, we will generalize the concepts found in Smalltalk. We will start by considering attributes of various collections. We can then decide which attributes will be represented as classes and which will be instance variables.

Collections can be sharable or non-sharable. Most of the Smalltalk-80 collection classes are non-sharable. Examples of sharable collections are suffix sharing linked lists and some user-maintained trees. We will only consider non-sharable collections in detail. Separate but related concepts are mutability and destructiveness. A collection with fixed contents is immutable. Normally such collections would support non-destructive operations which would return new immutable collections. Mutable collections, on the other hand, are normally modified in place by destructive operations. They are preferred for efficiency but must be copied explicitly to avoid unwanted sharing.

A collection, above all else, is a container for its elements. Some collections allow you to refer to the elements by some key or index. With others, you can only use the elements themselves. From this point of view, indexed collections are a specialization of keyed collections since indices can be thought of as integer keys [LaLonde 84]. Both keyed and indexed collections can distinguish between mutability of the keys and mutability of the collection as a whole. For example, an `Array` has fixed indices but its elements are mutable.

The difference between a `Set` and a `Bag` is that a `Set` doesn't allow duplicate elements. `Dictionaries` don't allow duplicate keys but we would like to support `Dictionaries` that do. Once again, for keyed collections, we can distinguish between allowing duplicate keys and allowing duplicate elements (values). In order to compare elements for equality, we must use some operation. In Smalltalk, we always use `=` except for `IdentitySet` and `IdentityDictionary` which use `==`. Both equality and identity comparison are common tests but we should also support arbitrary blocks.

Sorted collections are written to as unkeyed collections but are read as indexed collections. How can we deal with a `Dictionary` with sorted keys? We must support this specially. A collection with sorted keys really has indices (the ordering of the keys), keys (the keys themselves), and elements (the values corresponding to the keys). In order to sort the keys or

elements, we must be able to compare them. Normally they will be instances of a subclass of **FullyOrdered** so we can use the **<** operator. But we should also support partially ordered keys or elements. Also, the elements may form the desired full or partial ordering only when we use a block to compare them.

Now we are ready to decide how to implement each of these attributes. If we were to define classes for all reasonable attribute combinations, we would have too many classes to keep track of. Therefore, we will represent several of the attributes using instance variables. Attributes which have little effect on behavior will be chosen. The first such attribute is mutability. We will have a **locked?** flag as well as a **lockedKeys?** flag. We will also use instance variables to store the **allowDuplicateElements?** flag, as well as both equality and inequality comparison methods for both elements and keys. For sorted collections, we will want to know whether the keys or elements being sorted can be incomparable. This will also be stored in an instance variable.

The remaining attributes will be represented as classes. We start with the root **Collection** class (see Fig. 4.7). Under this we put **NonSharableCollection** (we are not considering sharable collections currently). We now want to include two subclasses, one for unkeyed collections and one for keyed collections (under which we can put indexed collections). But, as we said above, sorted collections are written to as unkeyed collections but are read as indexed collections. So we must separate out the read and write behavior. The class **WriteSet** defines the behavior for writing to a unkeyed collection and the class **ReadMapping** defines the behavior for reading from a keyed collection. We can now put the full versions of these classes under the partial versions. **Set** and **Mapping** define the missing read or write behavior. A **Mapping** is a more general keyed collection than Smalltalk's **Dictionary**. It supports several values per key. Under **Mapping**, we define **Function** which limits us to one value per key. We also define a corresponding **ReadFunction** class. We said earlier that indexed collections were a specialization of keyed ones. We define **OrderedCollection** under **Function** since there is only one element at a given index. We also define a corresponding **ReadOrderedCollection** class. We can now define **SortedElements** under **WriteSet** and **ReadOrderedCollection**. **SortedMapping** inherits from **Mapping** since we do not want to treat the indices as keys. Since **SortedElements**

have some behavior in common with `SortedKeys`, we define a common superclass `SortedCollection`.

Figure 4.7. The TMPL collection classes.

Of all the classes defined so far, only `Set` is concrete. Since sets are hashed, we must include the necessary hash function corresponding to the equality comparison used. Note that `Set` can be used to represent a `Bag` by setting `allowDuplicateElements?` to true. Under `OrderedCollection`, we define a `List` class implemented primitively using a Smalltalk `OrderedCollection`. We can also define a `LinkedList` class which would be circular and doubly linked since it is non-sharable. Under `List`, we define `String` (a list of characters) and `Interval`. `Interval` defines its own representation.

Under `Function`, we define `Dictionary`. Again, duplicates may be allowed, depending on the `allowDuplicateElements?` flag. We also define `Permutation` under `Function` since each key is mapped back to another key. Under `SortedElements`, we define `SortedSet` and `AVLTree`. The difference between these is the way they are implemented. `SortedSet` is implemented much like Smalltalk's `SortedCollection`. Elements are kept in order. New elements are inserted at the appropriate point. Adding large numbers of elements at once will cause a fast sort to be done¹. `AVLTree` maintains a balanced binary search tree, providing a worst case speed of $O(n \log n)$ for most operations. Similar classes are defined for sorting keys: `SortedDictionary` and `AVLDictionary`.

Those attributes which we are representing as instance variables can now be defined in the appropriate classes (see Table 5). Note that `String` appears in both the collection hierarchy and the numeric hierarchy (since `Strings` are fully ordered lexicographically). Actually, `Strings` can also inherit from `Group` since concatenation is associative. The "zero" is the empty string since it has no effect when concatenated onto a string. We can use multiple polymorphism to implement concatenation methods for mixed collection classes. In general, we must convert all input collection to the lowest common denominator before concatenating them. For example, concatenating a `Set` and a `Dictionary` would result in a `Set`.

There are some classes which might appear to be collections. `Polygons` are one such class. We can think of a polygon as a collection of vertices. This, however, does not tell the entire picture. Vertices in a polygon form a circular ordering. There is no "first" or "last" vertex. In such cases, it is best to encapsulate a collection, rather than inheriting from one.

¹Smalltalk uses quick sort but heap sort would be more appropriate since the elements may already be partially sorted. An even better solution is to sort the new elements using quick sort and then merge them into the existing elements.

Table 5. TMPL collection hierarchy instance variables.

Class	Representation
Collection	comparison, hashFunction, allowDuplicateElements?
NonSharableCollection	locked?
WriteSet	-
Set	-
ReadMapping	keyComparison, keyHashFunction
ReadFunction	-
ReadOrderedCollection	keyComparison, keyHashFunction, always = and hash
OrderedCollection	-
List	contents: Smalltalk OrderedCollection
String	elements are Characters
Interval	from, to, and by replace Smalltalk contents ¹
LinkedList	extraNode
SortedElements	comparison becomes inequality, hash function unused
SortedSet	list of elements maintained in sorted order
AVLTree	balanced binary search tree of elements
Function	-
Dictionary	set of associations
Permutation	association values are keys
Mapping	lockedKeys?
Function	...
SortedMapping	keyComparison becomes inequality, keyHashFunction unused
SortedDictionary	list of associations maintained in sorted order
AVLDictionary	balanced binary search tree of associations
SortedCollection	partiallyOrdered?
SortedElements	...
SortedMapping	...

4.3.3 USE OF FEATURES

In the previous sections, we saw how TMPL was actually used to implement the standard class library. Several of the language's features were used. Multiple inheritance was used in both the numeric classes and the collection classes. In fact, the `String` class was in both hierarchies! Multiple polymorphism was used in the numeric classes for cross-class operators as well as for explicit type conversion. Multiple polymorphism was used in the collection hierarchy for cross-class concatenation. The Boolean AND and OR operators use multiple polymorphism to

¹locked? and lockedKeys? always true, comparison always =, allowDuplicateElements? always false

short-circuit when given a `Block`. We took advantage of TMPL's ability to override representation with behavior in the collection hierarchy. The `Interval` class is non-destructive so its `locked?` flag is always true. We could have simply specialized the instance variable declaration with `locked?: True`, but this would still allocate space for the variable.

In many cases, the arguments passed to methods on subclasses are more specific. For example:

```
List ADD Default
String ADD Character
```

This happens whenever a subclass overrides an instance variable declaration with a more specific type (which is legal under TMPL subclass compatibility rules). The automatically created primitive accessor methods will follow this pattern. The problem is that if we try to call the submethod with an argument which is too general, we will simply invoke the supermethod rather than causing a type error. This is usually not the desired behavior. To get around this problem, we define a third method:

```
String ADD Default
```

and make it generate an error. Once again, we have run into the problem described in section 2.2.8. Although a string "is a" list, `String` is not really a valid subtype of `List`. Maybe supporting non-polymorphic arguments would help? The programmer could then specify that the second argument to `ADD` is not to be used during method lookup but it is still type checked. A simple extension to our example demonstrates the short-sightedness of this approach. We may want different behavior for adding 16-bit characters. If the second argument is not polymorphic, we cannot do this.

```
String ADD ExtendedCharacter
```

Method aliasing was not used in the numeric or collection hierarchies. An example using `DisplayableObject`, `Rectangle`, and `Box` was used to test merging and splitting to resolve ambiguities created by multiple inheritance. Special objects were not used in the numeric or collection classes. Of course, `True` and `False` were implemented as special objects. The empty sharable list would probably be a special object as well. Special objects are useful as enumeration types since they eliminate the need for case statements. Of course, all variables are typed, but it would be difficult to point out an example that demonstrates the usefulness of

typing. The call graph navigation in the browser certainly takes advantage of the type information. For example, in Smalltalk, it is impractical to navigate from a new message to the method that will execute. In TMPL, the type information would typically result in a fairly short list of methods.

Surprisingly, union types were rarely used. The one place they were used was to represent the element comparison method as `Boolean | Block`:

<code>=</code>	<code>False</code>
<code>==</code>	<code>True</code>
<code>user specified</code>	<code>[]</code>

This is really not such a good idea. It would be better to store an object which knows how to compare two elements. This object would then have special objects as subclasses for the first two cases and a regular subclass for the last case. It turned out that the union type `SomeType | Nil` was not commonly used. This is because uninitialized variables are not `Nil` (they contain Smalltalk's `nil` which indicates that they cannot be read). In fact, the `Nil` special instance was not needed. When a fixed size collection is created, we want to initialize the instances to some reasonable value. The default value for elements of most collections should be uninitialized. Since we cannot store this value, we store a special object instead. We can then override the store method for this special object so that it does nothing. In general, any use of `Nil` would probably be better implemented using an appropriate special object instead [van Gulik 93b].

The syntax proved somewhat difficult at first, but it quickly became quite easy to read and write methods in TMPL. The ability to make method calls read as proper English sentences proved invaluable. Examples of some particularly readable methods are:

```
REPEAT Block Integer TIMES
IF Boolean THEN Block
MUST BE OVERRIDDEN
OrderedCollection ADD Default AT END
OrderedCollection WITH Default Default Default
Collection INCLUDES? Default
EMPTY class NonSharableCollection
IS Default AN INSTANCE OF Class OR A SUBCLASS?
```

The programming environment proved to be adequate. A proper debugger would be essential before attempting any significant programming project in TMPL. The browser is dependent on

the information stored during static analysis to function properly. This causes problems when writing new code since the information is often missing or out of date. The browser is fast enough for normal use but TMPL code runs very slowly. This is to be expected since it is being interpreted and no optimization was done to improve run-time performance.

4.4 FUTURE EXTENSIONS

There are several ways in which TMPL might be improved. The browser should indicate overrides and aliases to the programmer somehow. As we mentioned in the previous section, a proper debugger needs to be implemented and the static analysis should be done dynamically. This would enable the browser to not allow invalid methods to be accepted in some cases. Also, warnings can produce dialogs with suggestions for possible solutions. For example, an undeclared variable could be declared automatically (as type `Default`). An inspector is also needed. It would be worthwhile experimenting with lowercase bold keywords. Lowercase keywords are easier to read and to type. The only problem is that it will make it harder to come up with variable names since they cannot be entirely lowercase.

Replacing the static analysis with a dynamic one would enable the system to maintain a list of warnings and errors, much like Trellis's grass catcher. Rather than inserting the error messages into the code and pretty-printing (thus losing the programmer's formatting), the error list should simply list error messages and highlight the code in error when the programmer selects one. Dynamic analysis would eliminate the long waits for the static analysis to complete as well as the problems caused when code fails because a static analysis has not been done or the information stored from the last one is out of date. Currently, primitive accessor methods are generated during static analysis. Changes to the definition of a class do not affect the primitive accessor methods until a static analysis is done. Also, changes to a class definition should either migrate old instances (difficult in a typed language) or produce a new version of the class for the new instances. Adding a class instance variable requires all existing subclasses to be changed. Currently deleting a class does not delete its methods. This needs to be done. The methods should really be copied down to the subclasses, if any.

TMPL should really be compiled. The type information produced during static analysis could be used to do inlining. Most calls have one or a small number of possible methods that can be executed. In the latter case, we can compile a case statement and then inline each case. The inlined code can take advantage of the type information known for the particular call. In order to conserve space, we would only inline when the inline method (or methods) would be small.

Many of the warnings being generated by the static analysis could be eliminated by improving the type system. One such improvement would be to support method arguments as valid result types. For example, the `MINIMUM` method could be written as:

```
(MINIMUM x, y: PartiallyOrdered) ^ x | y
```

This allows the type inferencer to infer the result type of a call to this method with more precision. This could be used in conjunction with the existing class prefix to improve `CLASS`:

```
(anObject: Default CLASS) ^ class anObject
```

An instance prefix should also be supported. This could be used to improve `NEW`:

```
(NEW aClass: class Default) ^ aClass
```

Note that covariance of result types would be slightly more difficult to check because result types are functions of argument types. Another improvement that could be made to the type system is to distinguish between different kinds of blocks. Currently blocks are undifferentiated. The programmer should be able to declare that a variable contains a block which accepts arguments of particular types and returns a result of a particular type (or doesn't return a result at all). Together, these improvements to the type system would eliminate most of the warnings being generated during static analysis of the standard class library. Many typed languages support generic or parameterized types. These would allow the programmer to specify a collection with a particular type of elements. If the TMPL type system was extended to support parametric types, it would have to be done in a general way - it cannot be specific to collections.

Method aliasing could be improved in several ways. One possible improvement would be to allow aliases to reorder arguments. Currently the super method of an alias method must have the same number of arguments and they must appear in the same order. One of the nice features of TMPL is that methods read as sentences. This is possible because the arguments

don't have to be reordered to put the appropriate one in the "receiver" position. This advantage is lost for aliases because they must preserve the argument order of the method being renamed. It would be fairly easy to support alias methods with super calls which list the arguments in a different order. When traversing the method hierarchy at run-time, the argument values would have to be reordered whenever an alias link is traversed. Another possible improvement to method aliasing would be to allow multiple methods to be renamed. An alias method currently identifies a single method with its super call. It should be able to list several methods. If a super call is ambiguous, it should rename all the possible methods, giving a warning in case this wasn't intended. It should be legal to make a normal method call rather than a super call. This would rename all supermethods of the method being called. For example, we could rename (Integer + Integer) to (Integer PLUS Integer):

```
(x: Integer PLUS y: Integer) ^^ Integer  
-> (x + y)
```

The current named and unnamed super were found to be inadequate. In a single inheritance hierarchy, we normally make each initialize method call the super method. In TMPL's multiple inheritance hierarchy, this proved to be difficult. When there are several supermethods, so that using an unnamed super call would be ambiguous, we can call each of them using named super calls. This has the undesired effect of re-executing any common super-initialization. To get around this problem, we must explicitly navigate the class hierarchy, executing local initialization for each class. A new form of super call which executes the method call on all supermethods in the appropriate order would make this more practical. Class initialization can have a similar problem, but it is actually easier to solve. Generally, classes must be initialized in a particular order. Actually, it is a partial order, determined by prerequisite constraints. All we have to do is declare these prerequisites, by implementing a method which returns a set of classes which must be initialized before the given class. The system can then be initialized by generating a partial ordering from the prerequisites and following any full ordering which satisfies it to do the actual initialization.

Method headers should be extended to specify a pattern. Any call which would match the pattern will execute it. The pattern would be represented as a regular expression in which the "terminals" are keywords and arguments. This lets us represent methods with optional or repeated arguments succinctly. We can also support multiple names for the same method and

“copying” aliases. Within the method, the code can ask questions about which keywords and arguments were actually supplied. Multiple polymorphism removes many case statements from our code. Pattern method headers lets us recombine cases that we would like to handle with a single piece of code. We mentioned in section 4.2.2 that union types can be thought of as regular expressions, as well as the problems with doing this. To implement pattern method headers, we would expand the regular expression to generate all possible simple method headers. Each of these would then be inserted into the appropriate hierarchy with a pointer to the actual method. Since there can potentially be an infinite number of these simple method headers, we must limit ourselves to methods which could actually be called, based on the maximum number of keywords and arguments in any method call. We can support class aliases by having several global variables all containing the same class. All the aliases must be considered identical during method lookup.

Modules should be added to TMPL to support privacy. Any method, including the automatically generated primitive accessor methods, can be declared as private. Private methods are inherited normally but they cannot be called by methods in other modules unless they declared themselves a submodule of the defining module. Public methods can be called from any module which declares itself a user of the defining module. Classes and other global variables are also scoped within a module. The browser would show a list of modules before the list of classes. If a module is selected, only classes and methods defined in that module are shown. The browser can be put in “specification” mode, causing it to show only public methods and only the headers (including comments) of these.

We saw in section 2.2.8 how using a subtype hierarchy to represent an is-a relationship caused problems. We discussed several alternatives for limiting substitutability. The first option—violations of compatibility rules implicitly limiting substitutability—would be appropriate in TMPL. In addition to listing subtype compatibility violations in the error list, we would also include methods which require substitutability which can no longer be ensured due to these violations. We could also implement the last option in TMPL. We would extend the syntax for variables and methods to allow explicit violations of covariance. Excludes, which always violate the subtype rules and thus limit substitutability, would also be supported.

CHAPTER 5. CONCLUSIONS

We have demonstrated how complex the relationships between features are (see Fig. 2.16). There are often several options for each feature, further complicating matters. Advanced features often allow the programmer to model his problem domain more accurately, producing a cleaner implementation with less code duplication. However, they can be dangerous if misused and they complicate the language making it more difficult to learn.

Two conflicts were encountered in TMPL after the design stage. Method aliasing appears to conflict with multiple polymorphism, but so long as the language is typed, we can use type inference to determine which name must be used. As a welcome side-effect, type inference detects many potential errors, providing the programmer with a grass catcher tool as part of his programming environment. It is worth noting that this use of type information is unusual. Normally type information is entirely declarative. It is used to do type checking and to optimize the compiled code. Here the type information affects the semantics.

The second conflict encountered involved the dual specification and implementation hierarchies. The original design called for separate hierarchies with common elements, but this feature has to be removed when it became apparent that almost all variables had to be classed rather than typed. The problem stems from the fact that all arguments are used for method lookup. In a message passing system, only the receiver would have to be classed. Arguments could remain typed. Result types can also remain typed, but this is trickier. In order for methods which return the receiver to be type-safe, we must introduce a type pseudo-variable to refer to the type corresponding to the class of the receiver. POOL-I uses this technique, for example:

```
CLASS Float

METHOD minimum (aFloat: MYTYPE) : MYTYPE
    IF SELF = aFloat
        THEN RESULT SELF
        ELSE RESULT aFloat
    FI
END minimum
```

If we simply declared the result type to be `Float`, this method would not be type-safe. This can be shown by defining a class `Complex` which is a subclass of `Float` but not a subtype of `Float`. `Complex` will inherit this `minimum` method. When it tries to return `SELF`, a type check will fail because `SELF` is not a `Float`. By declaring the result type to be `MYTYPE`, we are changing the specification for the inherited method to indicate that the result type is “the type of `SELF`”, which is `Complex`. Since the receiver is classed rather than typed, we can only send messages to `SELF` which are implemented locally and in superclasses. In general, it is not type-safe to pass `SELF` as an argument to a method since `SELF` may be a subclass of the class defining the method which is not a subtype. However, the use of `MYTYPE` as an *argument* type (as in the above example) can often be used to make such calls type-safe.

The problems we encountered in section 2.2.8 when using the subtype hierarchy to represent is-a relationships would be alleviated in a dual hierarchy. In order to ensure complete substitutability, most types must be divided into two parts—one for read behavior and one for write behavior. However, only a single class need be defined to implement both of these types. For example, we could pass in a `String` where an `Array` is expected by declaring the parameter to be of type `ReadOnlyArray`. The type hierarchy would be quite complex in order to allow full substitutability without exceptions (see Fig. 2.14), but the subtype relationship can be implicit so the programmer does not need to manipulate this complex hierarchy directly. The class hierarchy can be relatively small and simple. Note that even non-destructive objects usually need some destructive behavior for initialization.

Typed Smalltalk is another typed language with separate type and class hierarchies. It has no type pseudo-variable equivalent to POOL-I's `MYTYPE`. The following example is taken from [Johnson et al 88]:

```
{ arguments: aMagnitude <MagnitudeType>
  returnType: <MagnitudeType>}
max: aMagnitude
  ^ self < aMagnitude ifTrue: [aMagnitude] ifFalse: [self]
```

If we define a subclass of `Magnitude` which is not also a subtype, the inherited version of this method would not be type-safe. To use a somewhat artificial example, assume that `Date` inherits from `Magnitude` but is not a subtype of `Magnitude`. If we send a `max:` message to a `Date` and it tries to return `self`, a type error will occur because `self` is not a `Magnitude`. Furthermore,

the argument to the `max:` message could not be another `Date`. Although it looks like the method above describes how to compute the maximum of two objects in general, actually it is limited to subclasses of `Magnitude` which are also subtypes.

We can now look at Trellis in a new light. There are two major features that could be added to the language. The restrictions of its single type hierarchy would be less troublesome if a separate class hierarchy were added. Trellis's designers were considering doing this early on. They chose not to because the name space would be more crowded and the language would be more complicated. Trellis's single polymorphism implies that arguments are contravariant, and therefore instance variables are invariant. If a second hierarchy were added, this problem would disappear because accessor methods would be inherited by subclasses rather than subtypes. Another way instance variable invariance could be eliminated is by adding multiple polymorphism. Our experience with TMPL highlights the problems with this approach. To avoid calling more general methods than intended, we must implement methods as errors—hardly appropriate in a strict hierarchy. Limiting substitutability solves this problem as well as others that arise when we attempt to use the subtype hierarchy to represent is-a relationships. However, this would complicate the language significantly. Note that we could not add both the second hierarchy and multiple polymorphism since these features conflict. The designers of Trellis chose to include neither of these features, opting for a simple language instead. Whether this was for the best remains to be seen.

If we are designing an object-oriented language, and we choose to include typing, we will end up with a language much like Trellis. If we try to include more complicated features, we must complicate the language further in order to solve the problems that arise. The resulting language will be much like TMPL (with limited substitutability) or POOL-I. This approach looks promising but still may not be practical. We must ask ourselves if the advantages of typing are worth the ensuing problems. Are typed programs significantly more reliable than untyped ones? Can type inference deduce enough type information from untyped code to produce efficient code? SELF's performance is impressive, but can it compete with conventional languages like C? It is clear that declarative type information can only improve performance over type inference systems in untyped languages, but it is not clear how significant this difference is. Advances in type inference systems will likely narrow the gap.

If we choose to make our language untyped, we have many more non-conflicting choices. The wide variety of untyped object-oriented languages attests to this. Smalltalk does not include many advanced features which cause problems. However, it is much more complicated than necessary. SELF is a much smaller language but the lack of structure in a prototype-based system can be confusing for a beginning programmer. Classes provide a standard framework in which to visualize a program. The complexities of Smalltalk are largely due to metaclasses and Smalltalk's large class library. For learning object-oriented programming, a simplified version of Smalltalk may be more appropriate than SELF.

SELF's most significant contribution is its efficiency. Even without classes, SELF manages to outperform Smalltalk by a factor of four. The designers of SELF have advanced the state of the art in several areas, including garbage collection, type inference, and dynamic translation. SELF is now about half as fast as C and is constantly being improved. These optimizations will eventually become common in object-oriented systems, bringing their performance close enough to that of procedural languages so that most applications can be written in object-oriented languages. Many projects are implemented in conventional languages today because efficiency is perceived as a problem, even if the performance of existing object-oriented languages would be adequate. In many cases, prototypes are implemented in object-oriented systems, after which the final product is written in a procedural language. The problem with this approach is that the final version is difficult to maintain.

Object-oriented languages are gaining popularity as programmers and their managers come to realize their benefits. Rapid prototyping, reuse, and modifiability are some of the major advantages over conventional procedural languages. As hardware performance improves and software complexity rises, the relative cost of construction and maintenance of software is increasing. For applications with a small number of users, it is more cost-effective to buy faster hardware and use slower object-oriented languages. As the performance of these languages improves relative to procedural languages, this trade-off will shift. Even application software that is mass-marketed may be more cost-effective to implement in a fast object-oriented language, particularly today's feature-packed software packages. It has become common practice to use C++ for developing application software and operating systems. We can expect the optimization techniques pioneered in SELF to produce a new generation of

untyped object-oriented languages with performance approaching that of conventional typed languages. However, application developers will continue to shy away from untyped languages because of their perceived unreliability. We can expect untyped object-oriented languages to be used primarily for small applications, research, and prototypes, for quite some time. This is unfortunate because, as we have seen, typing complicates a language considerably. The task ahead of us is to convince project managers that untyped languages can produce reliable software.

INDEX

- abstract class **9**, 34, 58, 70, 80, 89, 98
- abstract data type **28**, 31
- abstract method 89, 92
- accessor methods **27**, 59, 61, 65, 87, 89
- Ada **2**, 5, 7, 13, 54
- adapters **30**, 38
- alias **8**, 37, 55, 66, 68, 106
- alias method 79, 91, 107
- ambiguity **18**, 21, 23, 53, 54, 55, 57, 60, 61, 67, 69, 82, 104
- application browser 39
- arguments 6, 7, 21, 23, 50, 53, 59
- assertions 56
- assignable 25
- base class 8, 57
- baseline 40
- Beta 62, 66
- bipartite graph 82
- blocks 79, 84, 107
- BNF 75
- bootstrap 83, 85
- browser 14, 49, 75, 93, 105, 106
- byte codes 47
- C **5**, 56
- C++ **1**, 6, 13, 41, 52, **57**, 113
- change set 39
- change sorter 41
- class categories 39
- class instance variables 70
- class methods 54
- class variables 50, 54, 59, 70
- classes **9**, 20, 32, 46, 50, 61, 70, 80
(also see *implementation*)
- clone **21**, 25, 51
- CLOS **59**, 60
- CLU **2**, 7, 54
- co/contravariant 54, 55, 56, 61, 71, 86
- code sharing 67
- coercion 58, 98
- collection classes 98
- comments 76, 94
- Common Lisp Object System **59**, 60
- compile 5, 32, 47, 48, 50, 53, 55, 58, 61, 89, 107
- components 54 (also see *slots*)
- conforms to 28
- constructors 59
- contravariant **33**, 61
- conversion 26, 98, 102, 103
- covariant **33**, 61, 88, 107
- DAG **17**, 23, 25, 29, 54, 55, 57, 60, 69, 74, 81, 93, 97
- data parents **22**, 25, 31, 52
- data slots **4**, 27
(also see *instance variables*)
- data structure **28**, 30
(also see *implementation*)
- data type 28, 67
- debugger 14, 49, 56, 75, 95, 105, 106
- Default 74
- delegates 21
- delegation 21
- delegators 21
- derived class 8, 57
- direct interaction 53
- directed acyclic graph 17
(also see *DAG*)
- dot call 73

- double dispatching **23**, 68
- Dylan **60**, 70
- dynamic inheritance **24**, 52
- dynamic translation 47, 51, 113
- dynamic typing **5**, 32, 60, 71
- Eiffel 37, **55**, 86
- encapsulation **4**, 67, 69
- encapsulators **30**, 38, 57, 71
- environment 83
- ENVY 40
- exception 47, 81, 85
- exclude **8**, 37, 46, 49, 54, 55, 59, 65, 66, 86, 91, 109
- exemplar 21 (also see *prototypes*)
- explicit 30
- exported 55
- expressions 77
- extensional naming 52
- family 41
- features 55
 - (also see *slots, components*)
- fields **5**, 54
 - (also see *instance variables*)
- finalization code 47
- first-class 46, 47, 60
- friends 59, 65
- full order 97
- function 73, 102
- function overloading 58
- fuzzy typing 71
- garbage collection **12**, 46, 47, 51, 54, 55, 57, 59, 60, 61, 113
- generic **33**, 55, 57, 61, 107
- generic function 59
- goto 67
- grammar 75
- graphical user interfaces 14
- grass catcher **15**, 87, 106, 110
- group **97**, 102
- GUI 14
- immutable 99
- implementation **28**, 38, 50, 70, 86
 - (also see *data structure, class*)
- implements **28**, 30, 38, 50
- implicit 29, 38, 61
- incremental garbage collection 13
- information hiding **4**, 64
- inheritance **7**, 52, 61, 65, 67, 70
- inline 50, 51, 89, 107
- inspector 14, 16, 56, 106
- instance variables **4**, 27, 50, 59, 83, 89
- instance-class 10
- instances 9
- interpreter 83
- invariant 56, 86
- is-a 17, **34**, 42, 91, 104, 109, 111
- iterators 54
- latent typing 5
- lexical scoping 83, 88
- light-weight classes 20, 41
- Lisp 5, 12, 13, 59, 60
- literals 77, 84
- lobby 52
- local precedence order 60
- local slot 59
- local variable 53
 - (also see *temporary variables*)
- lookup 82, 92
- manifest typing 5
- mapping 100
- mark and sweep 13
- member functions 58
- merge **18**, 37, 58, 65, 67, 80, 104
- message passing **6**, 68, 91
- message 6
- meta-object protocol 60
- meta-programming **12**, 47, 60

- metaclasses **10**, 20, 47, 49, 60, 70
- method aliasing **37**, 61, 65, 79, 83, 91, 104, 107
- method categories 39
- method combination 60
- method hierarchies 81
- method lookup **7**, 8, 23, 84
- method patterns 68
- method slots 4
- methods **4**, **6**, 79
- mix-ins 20
- Modula-2 2, 5
- Modula-3 62
- modules **4**, 40, 65, 75, 109
- multi-methods 23
- multiple inheritance **17**, 22, 37, 55, 57, 59, 61, 67, 103, 104
- multiple polymorphism **23**, 58, 68, 81, 91, 98, 102, 103
- mutate 47, 60
- MYTYPE 111
- name-parameter association 58
- named super 68, 72
- non-destructive 99, 111
- non-primitive 70
- numeric classes 96
- object **4**, 6, 8, 11, 33, 51, 55, 57, 71, 74, 85
- object mutation 25
- object-based 2
- object-oriented 1, 46
- Objective C 62
- one-of-a-kind objects **20**, 70
- operations **5**, 54 (also see *methods*)
- operator overloading 58
- optional arguments 58, 61, 69
- override **8**, 27, 32, 33, 46, 54, 55, 58, 59, 65, 66, 80, 86, 92, 104, 106
- package 4
- parameter 23, 24
- parameterized types **33**, 51, 57, 107
- parent slots 52
- parser 75
- partial order 21, 51, 60, 81, **97**, 100
- Pascal 5, 53, 57, 67
- permutation 102
- polymorphism **6**, 7, 53, 55
- POOL-1 30, 37, **61**, 110
- pool dictionaries 65
- postconditions 56
- precedence 72, 73
- preconditions 56
- prerequisite 108
- primitive 9, 20, 27, 28, 49, 50, 79, 83, 87, 98, 101
- private **4**, 27, 31, 42, 49, 52, 54, 55, 58, 64, 75, 109
- procedure 73
- profilers 14
- programming environment **14**, 66, 93, 105
- programming in the large 39
- projects 39
- properties **29**, 61
- protected 58
- protocols 39
- prototypes **20**, 21, 25, 29, 51, 53, 61, 70, 113
- public **4**, 27, 32, 33, 52, 54, 58, 71, 109
- pure virtual functions 58
- redefined 55
- reference counting 13
- regular expressions 69, 72, 82, 108
- rename 8, 37, 49, 55, 61, 65, 79, 83, 91

- representation 5, 8, 9, 18, 20, 25,
28, 46, 52, 54, 55, 65, 67, 83,
89, 104
- representation as behavior 27
- resend 27, 31
- ring 97
- routines 55 (also see *methods*)
- scanner 75
- secret 55
- SELF 2, 20, 49, **51**, 67, 70
- sender path tiebreaker **23**, 53
- shadow 8, 59
- sharable 99
- shared slot 59
- signature types 51
- slots **4**, 27, 52, 59
(also see *components*)
- Smalltalk 1, 3, **46**
- Smalltalk-80 **47**, 75
- Smalltalk/V 48
- special objects 49, 70, 80, 104
- specialize 65
- specification **4**, **28**, 38, 50, 70, 86
(also see *types*)
- split **18**, 58, 65, 67, 104
- statements 78
- static analysis 87, 94, 95, 106
- static typing 5
- strong typing 5
- subclass responsibility 10
- subclasses 8
- substitutability **29**, 34, 111
- subtree visible 5
- subtype 6, 29, 33, **34**, 50, 53, 54,
57, 61, 71, 86, 91, 104, 109,
111
- subtype compatibility rules **29**, **33**,
34, 54, 57, 61, 71, 86, 87, 88
- subtype visible 54
- super 49, 68, 77, 79, 83, 89, 91, 108
- superclass 8
- supertype 54
- syntactic sugar 27, 54, 58, 73
- templates 57
- temporary variables 50
(also see *local variable*)
- TMPL **62**, **64**
- traits **21**, 70
- Trellis 2, **53**, 64, 86
- Trellis/Owl 53
- type components 54
(also see *class variables*)
- type generators **33**, 54
- type hierarchy 70
- type inference 51, 88, 91, 107, 110,
113
- type operations 54
- type-based encapsulation 52, 59
- Typed Smalltalk 50
- types **28**, **32**, 50, 54, 61, 78, 107
(also see *specification*)
- typing **32**, 66, 104
- uninitialized 83, 91, 105
- union types **33**, 54, 72, 82, 105
- unnamed super 92
- use 65
- variables 5, 50, 53, 55, 59
- virtual image (VI) 46
- virtual machine (VM) 46
- void 6
- weak typing 5
- wrappers 30

REFERENCES

TRADEMARKS

Ada is a registered trademark of the US Dept. of Defense, AIPO.

Apple is a trademark of Apple Computer, Inc.

Dylan is a trademark of Apple Computer, Inc.

Eiffel is a trademark of Interactive Software Engineering, Inc.

Macintosh is a registered trademark licensed to Apple Computer, Inc.

MS-DOS is a registered trademark of Microsoft Corp.

Objectworks is a trademark of ParcPlace Systems, Inc.

OS/2 is a trademark of International Business Machines, Inc.

Smalltalk-80 is a trademark of ParcPlace Systems, Inc.

Smalltalk/V is a trademark of Digitalk, Inc.

UNIX is a registered trademark of AT&T Bell Laboratories.

Windows is a trademark of Microsoft Corp.

OBJECT-ORIENTED PROGRAMMING

[LaLonde 84] LaLonde, W.R. 95.202 course, Data Structures and Data Types, School of Computer Science, Carleton University, January–December 1984.

[LaLonde and Pugh 91] LaLonde W.R. and Pugh J.R. *Subclassing $\neq$ Subtyping $\neq$ Is-a*. JOOP, Journal of Object-Oriented Programming, The Smalltalk Column, vol. 3, no. 5, pp. 57–62, January 1991.

[Smith 91] Smith, D.. Concepts of Object-Oriented Programming, McGraw-Hill, 1991.

[Stroustrup 87] Stroustrup, B. *What is "Object-Oriented Programming"?* Proceedings of ECOOP '87, European Conference on Object-Oriented Programming, Paris France, pp. 51–70, June 15–17 1987. Published as Lecture Notes in Computer Science, Springer-Verlag, vol. 276, 1987 and SIGPLAN Notices, vol. 22, no. 12, December 1987.

[Wegner 87] Wegner, P. *Dimensions of Object-Based Language Design*. Proceedings of ECOOP '87, European Conference on Object-Oriented Programming, Paris France, pp. 227–241, June 15–17 1987. Published as Lecture Notes in Computer Science, Springer-Verlag, vol. 276, 1987 and SIGPLAN Notices, vol. 22, no. 12, December 1987.

GARBAGE COLLECTION

[Baker 78] Baker H.G. *List Processing in Real-Time on a Serial Computer*. Communications of the ACM, vol. 21, no. 4, pp. 280–294, April 1978.

[Collins 60] Collins G.E. *A Method for Overlapping and Erasure of Lists*. Communications of the ACM, vol. 3, no. 12, pp. 655–657, December 1960.

[Duimovich 88] Duimovich J. Garbage Collection in a Multiprocessor Smalltalk System. M.C.S. Thesis, School of Computer Science, Carleton University, December 1988.

[Eckhart & LeBlanc 87] Eckhart J.D. and LeBlanc R.J. *Distributed Garbage Collection*. SIGPLAN '87 Symposium on Interpreters and Interpreter Techniques, St. Paul Min., June 24–26, 1987. Published as SIGPLAN Notices, vol. 22, no. 7, pp. 264–273.

[McCarthy 60] McCarthy J. *Recursive Functions of Symbolic Expressions and Their Computation by Machine*. Communications of the ACM, vol. 3, no. 4, pp. 184–195, April 1960.

[Nguyen 89] Nguyen, T. Creepy: An Incremental Secondary Storage Garbage Collector. M.C.S. Thesis, School of Computer Science, Carleton University, December 1989.

[Stringham 89] Stringham, J. Persistent Objects. M.C.S. Thesis, School of Computer Science, Carleton University, December 1989.

[Ungar 84] Ungar, D. *Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm*. ACM SIGSOFT-SIGPLAN Practical Programming Environments Conference, Pittsburgh PA, April 1984.

- [Williams et al 87] Williams L., Woleczko M., and Hopkins T. *Dynamic Grouping in an Object-Oriented Virtual Memory Hierarchy*. Proceedings of ECOOP '87, European Conference on Object-Oriented Programming, Paris France, pp. 79–88, June 15–17 1987. Published as Lecture Notes in Computer Science, Springer-Verlag, vol. 276, 1987 and SIGPLAN Notices, vol. 22, no. 12, December 1987.

PROTOTYPES AND EXEMPLARS

- [LaLonde 86] LaLonde W.R. Why Exemplars Are Better Than Classes. Technical Report SCS-TR-93, School of Computer Science, Carleton University, May 1986.
- [LaLonde 89] LaLonde W.R. *Designing Families of Data Types Using Exemplars*. TOPLAS, ACM Transactions on Programming Languages and Systems, vol. 11, no. 2, pp. 212–248, April 1989.
- [Lieberman 86] Lieberman H. *Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Languages*. Proceedings of OOPSLA '86, ACM Conference on Object-Oriented Programming Systems Languages and Applications, Portland Oregon, pp. 214–223, September 1986. Published as SIGPLAN Notices, Addison-Wesley, vol. 21, no. 11, November 1986.

SMALLTALK

- [Digitalk 89] Smalltalk/V. Digitalk, 1987, 1989. Smalltalk/V-286. Digitalk, 1988. Smalltalk/V-PM. Digitalk, 1989.
- [Goldberg & Robson 83] Goldberg A. and Robson D. Smalltalk-80: The Language and its Implementation. Addison-Wesley, 1983.
- [Goldberg & Robson 89] Goldberg A. and Robson D. Smalltalk-80: The Language. Addison-Wesley, 1989.
- [Krasner 83] Krasner G. (ed.) Smalltalk-80: Bits of History, Words of Advice. Addison-Wesley, 1983.

[ParcPlace 92a] Objectworks\Smalltalk User's Guide. Release 4.1. ParcPlace Systems part number DS10002001, 1992.

SMALLTALK EXTENSIONS AND REPLACEMENTS

[Epstein 90] Epstein, D. *Specification and Implementation Hierarchies*. 95.514 Course Project, School of Computer Science, Carleton University, April 1990.

[Johnson et al 88] Johnson R.E., Graver J.O., and Zurawski L.W. *TS: An Optimizing Compiler for Smalltalk*. Proceedings of OOPSLA '88, ACM Conference on Object-Oriented Programming Systems Languages and Applications, San Diego CA, pp. 18–26, September 25–30, 1988. Published as SIGPLAN Notices, Meyrowitz N. (ed.), ACM Press, vol. 23, no. 11, November 1988.

[LaLonde & van Gulik 88] LaLonde W.R. and van Gulik M. *Building a Backtracking Facility in Smalltalk Without Kernel Support*. Proceedings of OOPSLA '88, ACM Conference on Object-Oriented Programming Systems Languages and Applications, San Diego CA, September 25–30, 1988. Published as SIGPLAN Notices, Meyrowitz N. (ed.), ACM Press, vol. 23, no. 11, November 1988.

[OTI 91] ENVY/Developer Course Notes. Object Technology International, Inc., 1991.

[ParcPlace 92b] VisualWorks. Release 1, draft, information sheets. ParcPlace Systems, October 1992.

[ParcPlace] Object Kit\Smalltalk Advanced Programming User Guide. Release 1.0 for Objectworks r4. ParcPlace Systems part number 593101.

[Pascoe 86] Pascoe, G. *Encapsulators: A New Software Paradigm in Smalltalk-80*. Proceedings of OOPSLA '86, ACM Conference on Object-Oriented Programming Systems Languages and Applications, Portland Oregon, pp. 341–346, September 1986. Published as SIGPLAN Notices, Addison-Wesley, vol. 21, no. 11, November 1986.

[Tektronix 88] Smalltalk-80 Team Programming Tools: Phase I. Tektronix information sheet, 1988.

[Wirfs-Brock & Wilkerson 88] Wirfs-Brock A. and Wilkerson B. *An Overview of Modular Smalltalk*. Proceedings of OOPSLA '88, ACM Conference on Object-Oriented Programming Systems Languages and Applications, San Diego CA, September 25–30, 1988. Published as SIGPLAN Notices, Meyrowitz N. (ed.), ACM Press, vol. 23, no. 11, November 1988.

SELF

[Chambers and Ungar 90] Chambers C. and Ungar D. *Iterative Type Analysis and Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs*. Proceedings of SIGPLAN '90, ACM Conference on Programming Language Design and Implementation, pp. 150–162. Published as SIGPLAN Notices, ACM Press, vol. 25, no. 6, 1990.

[Chambers et al 89] Chambers C., Ungar D., and Lee E. *An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes*. Proceedings of OOPSLA '89, ACM Conference on Object-Oriented Programming Systems Languages and Applications, pp. 49–70, October 1989. Published as SIGPLAN Notices, Addison-Wesley, vol. 24, no. 10, October 1989.

[Chambers et al 91] Chambers C., Ungar D., Chang B-W., and Hölzle U. *Parents are Shared Parts of Objects: Inheritance and Encapsulation in SELF*. LISP and Symbol Computation: an International Journal, Kluwer Academic Publishers, The Netherlands, vol. 4, no. 3, 1991.

[Ungar & Smith 87] Ungar D. and Smith R. *SELF: The Power of Simplicity*. Proceedings of OOPSLA '87, ACM Conference on Object-Oriented Programming Systems Languages and Applications, pp. 227–241, June 1987. Published as SIGPLAN Notices, vol. 22, no. 12, December 1987.

[Ungar et al 91] Ungar D., Chambers C., Chang B-W., and Hölzle U. *Organizing Programs Without Classes*. LISP and Symbol Computation: an International Journal, Kluwer Academic Publishers, The Netherlands, vol. 4, no. 3, 1991.

TRELLIS

[O'Brien et al 87] O'Brien P.D., Halbert D.C., and Kilian M. *The Trellis Programming Environment*. Proceedings of OOPSLA '87, ACM Conference on Object-Oriented Programming Systems Languages and Applications, June 1987. Published as SIGPLAN Notices, vol. 22, no. 12, December 1987. Also printed as Technical Report DEC-TR-503, Digital Equipment Corporation, April 1987.

[Schaffert et al 86] Schaffert C., Cooper T., Bullis B., Kilian M., and Wilpolt C. *An Introduction to Trellis/Owl*. Proceedings of OOPSLA '86, ACM Conference on Object-Oriented Programming Systems Languages and Applications, Portland Oregon, September 1986. Published as SIGPLAN Notices, Addison-Wesley, vol. 21, no. 11, November 1986. Also printed as Technical Report DEC-TR-438, Digital Equipment Corporation, April 1986.

EIFFEL

[Meyer 87] Meyer B. *Eiffel: Programming for Reusability and Extendibility*. SIGPLAN Notices, vol. 22, no. 2, pp. 85–94, February 1987.

C++

[Goldsmith & Palevich 90] Goldsmith D. and Palevich H. *Unofficial C++ Style Guide*. Develop, April 1990.

[Sakkinen 88] Sakkinen M. *On the Darker Side of C++*. Proceedings of ECOOP '88, European Conference on Object-Oriented Programming, Oslo Norway, pp. 162–176, August 15–17, 1988. Published as Lecture Notes in Computer Science, vol. 322, Springer-Verlag, 1988.

- [Stroustrup 91] Stroustrup B. The C++ Programming Language. Addison-Wesley, 1986. Second edition 1991.

LISP DERIVATIVES

- [Betz 92] Betz D. *A Taste of Dylan*. Dr. Dobb's Journal, vol. 193, pp. 50–57, October 1992.
- [Bobrow et al 86] Bobrow D.G., Kahn K., Kiczales G., Masinter L., Stefik M., and Zdybel F. *Common Loops: Merging Lisp and Object-Oriented Programming*. Proceedings of OOPSLA '86, ACM Conference on Object-Oriented Programming Systems Languages and Applications, Portland Oregon, September 1986. Published as SIGPLAN Notices, Addison-Wesley, vol. 21, no. 11, November 1986.
- [Bobrow et al 88] Bobrow D.G. et al. *Common Lisp Object System Specification*. SIGPLAN Notices, vol. 23 (Special Issue), September 1988.
- [DeMichiel & Gabriel 87] Linda G., DeMichiel L.G., and Gabriel R.P. *The Common Lisp Object System: An Overview*. Proceedings of ECOOP '87, European Conference on Object-Oriented Programming, Paris France, pp. 151–170, June 15–17 1987. Published as Lecture Notes in Computer Science, Springer-Verlag, vol. 276, 1987 and SIGPLAN Notices, vol. 22, no. 12, December 1987.
- [Moon 86] Moon D.A. *Object-Oriented Programming with Flavors*. Proceedings of OOPSLA '86, ACM Conference on Object-Oriented Programming Systems Languages and Applications, Portland Oregon, pp. 1–8, September 1986. Published as SIGPLAN Notices, Addison-Wesley, vol. 21, no. 11, November 1986.
- [Steele 84] Steele Jr. G.L. Common Lisp: The Language. Digital Press, 1984.

ADA AND MODULAS

- [Harbison 92] Harbison S. *Safe Programming with Modula-3*. Dr. Dobb's Journal, vol. 193, pp. 88–96, October 1992.

[Ichbiad et al 83] Ichbiad et al. The Ada Programming Language: Language Reference Manual. ANSI/MIL-STD-1815A-1983, US Department of Defense, US Government Printing Office, 1983.

[Wirth 88] Wirth N. Programming in Modula-2, 4th edition, Springer-Verlag, 1988.

OTHER LANGUAGES

[America & van der Linden 90] America P. and van der Linden F. *A Parallel Object-Oriented Language with Inheritance and Subtyping*. Proceedings of OOPSLA/ECOOP '90, ACM Conference on Object-Oriented Programming Systems Languages and Applications and European Conference on Object-Oriented Programming, Ottawa Canada, pp. 161–168, October 21–25, 1990. Published as SIGPLAN Notices, Meyrowitz, N. (ed.), Addison-Wesley, vol. 25, no. 10, October 1990.

[Cox 86] Cox B. Object-Oriented Programming—An Evolutionary Approach. Addison-Wesley, 1986.

[Epstein 89] Epstein, D. *PM: A Pattern Matching Language*. 95.501 Course Project, School of Computer Science, Carleton University, December 1989.

[Kristensen et al 87] Kristensen B.B., Madsen O.L., Møller-Pedersen B., and Nygaard K. *Classification of Actions or Inheritance Also for Methods*. Proceedings of ECOOP '87, European Conference on Object-Oriented Programming, Paris France, pp. 98–107, June 15–17 1987. Published as Lecture Notes in Computer Science, Springer-Verlag, vol. 276, 1987 and SIGPLAN Notices, vol. 22, no. 12, December 1987.

MATHEMATICS

[Bloch 87] Bloch N.J. Abstract Algebra with Applications. Prentice-Hall, 1987

PRIVATE COMMUNICATION

[Epstein 92] Epstein P. *Private communication*. Electronic mail, October, 1992.

[Epstein 93] Epstein P. *Private communication*. February 1993.

[Gregory 93] Gregory D. *Private communication*. February 1993.

[Johnson 92] Johnson, R. *Private communication*. Paraphrased from article, Usenet News Group comp.object, 1992.

[Knight 92] Knight A. *Private communication*. Electronic mail, October, 1992.

[van Gulik 92] van Gulik M. *Private communication*. October 1992.

[van Gulik 93a] van Gulik M. *Private communication*. February 1993.

[van Gulik 93b] van Gulik M. *Private communication*. March 1993.

END

14-12-93

FIN