Supporting the Evolution of Software

Supporting the Evolution of Software

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
Rector Magnificus, prof.dr. M. Rem, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op donderdag 1 juli 1999 om 16.00 uur

door

Petrus Johannes Schoenmakers
geboren te Breda

Dit proefschrift is goedgekeurd door de promotoren:

prof.Dr.-Ing. J.A.G. Jess
prof.Dr. P. Marwedel

Copromotor:
dr.ir. C.A.J. van Eijk

Printed by the Universiteitsdrukkerij Technische Universiteit Eindhoven from camera-ready copy provided by the author, typeset by LaTeX in Computer Modern, and prepared using GNU Emacs on an Apple Macintosh PowerBook.

Cover photograph by Net <net@gerbil.org>.

Visit the TOM website at `http://gerbil.org/tom`.

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Schoenmakers, Pieter J.

Supporting the evolution of software / by Pieter J. Schoenmakers. —
Eindhoven : Technische Universiteit Eindhoven, 1999.
Proefschrift. — ISBN 90-386-1620-1
NUGI 856
Trefw.: software ; hergebruik / object-georienteerde programmeertalen
/ object-georienteerd programmeren / debugging.
Subject headings: software reusability / object-oriented languages /
object-oriented programming / program testing.

Summary

[Zie voor een Nederlandse samenvatting bladzijde vii.]

Modern design methods are based on object-oriented analysis, which amounts to the classification and specification of objects and their interaction. A subsequent implementation in an object-oriented programming language encompasses the description of classes of objects. An important aspect of these classes is their suitability for being reused, since something that is reused does not need to be designed, implemented, and tested a second time.

Modern design systems and programming languages provide reuse mechanisms like subclassing and wrapping. Unfortunately, using these mechanisms, the choice of reusing a class is a binary one: the class is suitable for reuse or not, in which case a new class must be designed instead. Even minor imperfections and shortcomings imply a redesign instead of the desired reuse.

This dissertation discusses the validity of this observation and concludes that the usual approach to solving this problem, which concentrates on the development of classes, does not aid in solving the problem. The result of such an approach remains reuse following a model of revolution: either all is fine, or everything must be different.

The lion's share of this dissertation discusses the design, implementation, and use of the object-oriented programming language TOM. In TOM, to reuse a class is not a binary choice: a class can be adjusted to make it suitable for specific situations, even without availability of its source code. This way TOM supports the evolution of classes that adapt to the circumstances of reuse: either all is fine, or slightly different.

Samenvatting

[See page v for a summary in English.]

Moderne ontwerpmethoden zijn gebaseerd op object-georiënteerde analyse, wat neerkomt op de classificatie en specificatie van objecten en hun interactie. Een implementatie in een object-georiënteerde programmeertaal behelst het beschrijven van klassen van objecten. Een belangrijk aspect van deze klassen is hun mogelijke geschiktheid voor hergebruik. Immers, wanneer iets wordt hergebruikt hoeft het niet nogmaals ontworpen, geïmplementeerd en getest te worden.

Moderne ontwerpsystemen en programmeertalen bieden diverse mechanismen tot hergebruik zoals *subclassen* en *wrappen*. Echter, hierbij is de keuze tot hergebruik van een klasse een binaire keuze: ofwel de klasse is geschikt om hergebruikt te worden, ofwel zij is dat niet en een nieuwe klasse dient ontworpen te worden. Zelfs kleine onvolkomenheden en tekortkomingen leiden aldus tot herontwerp in plaats van het beoogde hergebruik.

Dit proefschrift bespreekt de validiteit van deze observatie en concludeert dat een aanpak van dit probleem op de gebruikelijke manier, geconcentreerd op de ontwikkeling van klassen, niet helpt. Een dergelijke aanpak kan nog steeds slechts hergebruik volgens het revolutiemodel herbergen: iets is goed, of alles moet anders.

Het leeuwendeel van dit proefschrift behelst ontwerp, implementatie, gebruik en evaluatie van de object-georiënteerde programmeertaal TOM. Hergebruik van een klasse is in TOM geen binaire keuze: aan een klasse kan van alles bijgesteld worden om haar geschikt te maken voor een specifieke situatie; hiervoor hoeft zelfs de broncode van de klasse niet beschikbaar te zijn. Op deze manier ondersteunt TOM de evolutie van klassen, die zich aanpassen aan de omgeving van hergebruik: iets is goed, of een klein beetje anders.

Preface

A Very Important Aspect of a dissertation is its value to the reader. A difficult aspect of writing a dissertation is ensuring it will be readable to many a reader while adequately expressing the author's intentions. A Very Important Aspect of writing a dissertation is learning to appreciate the mechanisms of written mass communication for a small public.

I am indebted to Koen van Eijk and Jochen Jess for their influential proofreading of early manuscripts of this dissertation. *Release early, release often*, the credo of the *open source*-software community, certainly applies to projects for which the producer initially can not adequately value the product and really needs the feedback, like this dissertation.

Of course I must also thank professor Jess for providing me with a nice mission in a very nice group. Thanks to all group members, past and present, for being this very nice group. Special thanks to Geert Janssen for allowing me into the smoke shack and for showing how to be a Lecturer. Thanks to Jos van Eijndhoven for adequate coaching. Also thanks to Raymond Nijssen for being a challenging discussion partner before leaving for the Valley (and for the apartment afterwards).

Ad ten Berg and Frans Theeuwen, now both at Philips Research' ED&T, deserve a special mention for allowing me to write them a tool in a language they did not know. If one thing has resulted from that effort, it must be the fixing of many bugs in the TOM compiler.

Thanks to Michael Brouwer for our early programming experiences—which have strongly influenced the design of TOM—and for the implementation discussions while enjoying the Amsterdam night life.

Last but not least: many thanks to Net, for the fun and the future.

`--Tiggr`

Eindhoven, May 1999

Contents

Chapter 1

Object orientation

This dissertation concerns object-oriented programming. The object-oriented programming paradigm employs a few simple concepts that can be combined and applied in various ways, thus giving rise to intricate terminology that can be confusing and overwhelming to the unsuspecting reader. To prevent terminology from being a barrier to the successful understanding and appreciation of this dissertation, this chapter provides a gentle introduction to the concepts of object-oriented programming and the associated terminology. The reader who is already familiar with the general terminology of object orientation and object-oriented programming is encouraged to bravely skip this chapter.

The terminology of object-oriented programming as it is used in this dissertation is based on the terminology used in Objective-C, which itself is based on that of Smalltalk and C. In those cases where Objective-C lacks similar concepts, terminology is borrowed from Eiffel and plain old English.

For a thorough explanation of the object-oriented programming paradigm, the reader is referred to books that, rather than focusing on a particular programming language, discuss object-oriented programming in general, with a particular object-oriented programming language in mind. Examples of fine books are *Object-oriented programming: an evolutionary approach*, the original Objective-C book by Brad Cox [10], *Object-oriented programming and the Objective-C language*, the book by the company that gave the language more widespread use [36], and *Object-oriented software construction*, by Bertrand Meyer of Eiffel fame [29].

1.1 Terminology

To start with the basics: a computer is concerned with the manipulation of data. Every piece of data has a *type*. A type defines a set of *operations* that can be performed on data with that type.

When a type S is a *subtype* of a type T, the operations of S include at least the operations of T, i.e., the set of operations of S is a superset of the set of operations of T. The subtype relation defines a partial order; it is reflexive, transitive, and antisymmetric.

When S is a subtype of T, T is a *supertype* of S. A piece of data *conforms to* the supertypes of its type. Any operation defined by one of the supertypes can also be performed on that data.

A type defines an *interface* to data that conforms to the type: interaction with such data is only possible through the operations that are offered by that interface.

In object-oriented programming, a type is *implemented* by a *class*. Any number of *instances* of the class can be created; each instance is a value of the type that is implemented by its class. These instances are usually referred to as *objects*.

In an object-oriented programming language, the types are implicitly defined by the classes. Each class defines a corresponding type. A class can *inherit* from another class, making the type defined by the inheriting class a subtype of the type defined by the class from which it inherits. The inheriting class is a *subclass* of the *superclass*.

A class has an implementation for each operation of its type. Such an implementation is called a *method*. When a class inherits from a superclass, it acquires not only a supertype but also the methods that implement the operations of the supertype. As a consequence, a class does not need to define methods for all operations that are defined by its supertypes.

A class can define additional methods, which adds operations to the type that is defined by the class. A class can also define methods that implement operations it inherits from a supertype. Such a method *overrides* the method for the same operation that the class inherits from a superclass.

The data making up an instance consists of a number of *instance variables*. The storage space needed for an instance variable is part of the storage needed for the instance.

A class inherits from a superclass, in addition to the methods, also the instance variables: the set of instance variables of a class is a superset of the set of instance variables of a superclass. Only the methods of a class can directly manipulate the instance variables: interaction with an instance can only take place through the invocation of its methods. This encapsulation of the instance variables is depicted in figure 1.1.

When the type of a variable is the type of a class, the value of the variable is either the data of an instance, or it is a reference to an instance. In the first case, the variable holds an instance of an *expanded class*; the second case is the normal, non-expanded, case. The storage space required

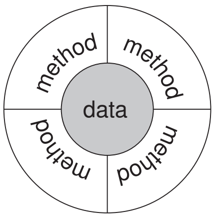

Figure 1.1: Methods encapsulate an object's data.

for a variable containing a reference to an instance does not vary with the class of the instance being referenced. Most importantly, the variable can reference any instance, of any size, as long as it conforms to the type of the variable.

Instances of the same class are discernible by their *identity* and their state. The state of an instance varies during its life time and two instances can happen to have the same state. However, distinct instances of the same class have, by definition, a different identity. The identity of an object does not change during its life time. All aspects of an object other than its state and identity, are defined by its class.

The distinction between the operations of a type and the methods of a class, i.e., between interface and implementation, enables a level of abstraction. When a variable references an object, performing an operation on the object causes an invocation of the corresponding method. Which operations *can* be invoked is determined by the type of the variable. Which method *will* be invoked for a given operation is defined by the class of the object that is referenced by the variable. This mechanism is called *dynamic method binding.* On the other hand, when the method to be invoked is determined from the type of the variable, the mechanism is called *static method binding.*

When a variable references an instance of a subclass conforming to the variable's type, dynamic method binding obeys the rules of a subclass being able to override methods of a superclass. This is called *polymorphism.* Static method binding disables polymorphism.

So far, this section has been concerned with objects as instances of a class. In languages like Smalltalk and Objective-C, each class is represented by a *class object.* The class object can be regarded as the sole instance of its *meta class.* The existence of a class object enables otherwise impossible behaviour: it can be referenced and messages can be sent to it. The archetypical use of class objects is in creating new instances: the class object provides a *class method* that returns a reference to a newly allocated object each time it is invoked.

Sending messages

An often-used metaphor for the invocation of an operation or method of an object is *sending a message* to that object, as depicted in figure 1.2. The object is the *receiver* of the message. A *message-send* contains a *selector*, which indicates the desired operation by name and the types of argument and return value (the *subject*, *arguments*, and *return-type* header fields in figure 1.2). Furthermore, the message can have *arguments*—none in the example—and a value can returned be in response, which will have a certain *return-type*. A message-send is blocking, in that the sender of the message waits for the answer, i.e., for the completion of the method.

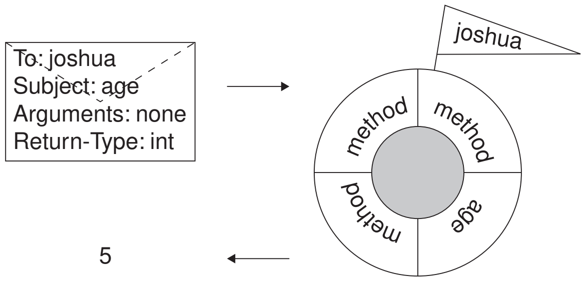

Figure 1.2: An invocation is like a message-send and waiting for an answer.

1.2 Execution model

This dissertation concerns code: source code in an object-oriented programming language, created by a human and translated by a compiler into object code, to be fed as instructions to a CPU. The body of the source code consists of methods, and the body of each method is a sequence of *statements*. The statements make up the executable content of the object code.

Method execution is sequential: when a method is invoked, its statements are executed one at a time, one after the other. Statements are available that influence the flow of control within a method to allow conditional and repetitive execution of one or more statements.

Method invocations are blocking: execution of a calling method is suspended while the called method is executing. As with every program, written in any programming language, running on a CPU, a stack is present to administer suspended method invocations and to provide storage space needed during the execution of a method, e.g., for local variables.

Execution of code needs a CPU, a stack, and an address space in which all code and data of the running program resides. The number of CPUs and stacks can vary though each CPU needs a stack. When multiple CPUs are available, the program can run *multi threaded.*

Execution of multiple threads is asynchronous: the processor instructions of all running threads are executed independently. On a machine with N processors, up to N threads of the same program can be running concurrently.

Though a programming language can explicitly support, use, or require a multi-threaded environment, multi-threaded code can be written in *any* programming language, even C.

1.3 Compilation model

The model of compilation used throughout this dissertation is very common on contemporary computers and operating systems. It deserves a basic explanation since understanding it helps distinguishing the *objects* from the *object code* and, in turn, the *object code* from the *object files.*

Figure 1.3 depicts three stages of code: source code, object code, and executable code. *Source code* is contained in *source files.* In figure 1.3, `a.c`, `b.c`, and `d.c` are source files containing source code written in C. Source files are translated by a *compiler*, at *compile time*, into *object files* that contain *object code.* In figure 1.3, the object files `a.o`, `b.o`, and `d.o` are the result of compiling the corresponding source files.

An object file contains symbol definitions, which are simple names like `_fac` or elaborate variations thereupon. These symbols denote the start of the object code that has resulted from compiling the source code of the corresponding function. An object file also contains references to symbols for which it does not contain a definition, like `_printf` in the object file `a.o` in figure 1.3. Any further information, for example that `fac` really is a C function that accepts an `int` and returns an `int`, or that `printf` accepts any number of arguments, the first one of which must be a string, is absent from the object file.

It is the task of the *linker*, which runs at *link time*, to resolve references between object files. Any remaining undefined references it resolves by retrieving object files from *libraries.* Only those objects that actually resolve an undefined reference will be retrieved from their containing library.

To the linker, a library is a collection of object files. In the figure, `libc.a` denotes the standard C library. A library is accompanied by various *header files* that declare the interface to the library. In figure 1.3, `stdio.h` declares how `printf` can be invoked. Header files are used by the compiler.

The linker produces object code that does not contain any unresolved references. Such code can be executed by a CPU. The file containing the executable code is called an *executable.* The executable is a program that can be run.

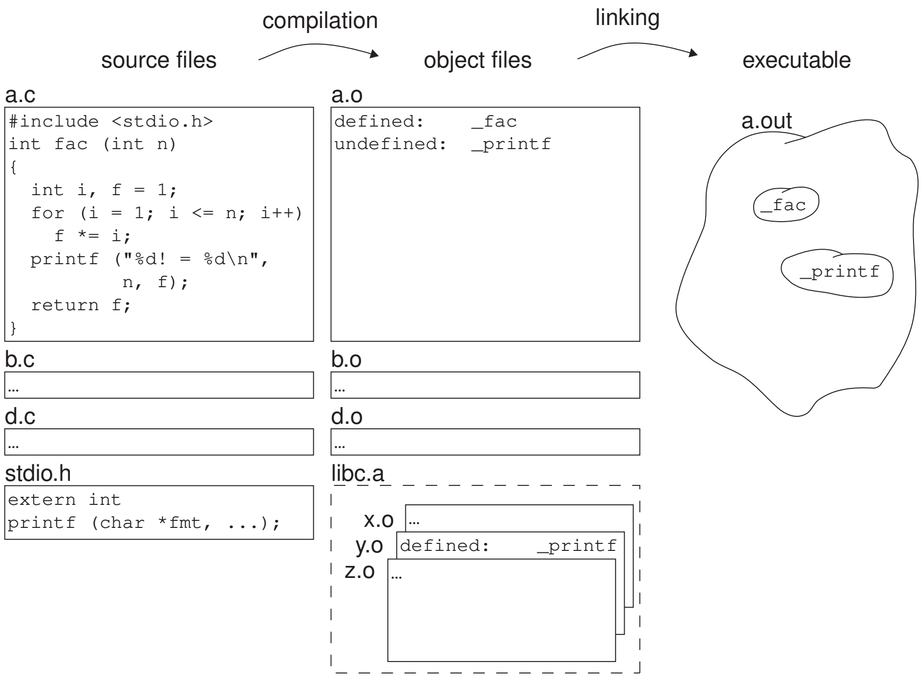

Figure 1.3: Source, object, and executable code.

The time during which a program is running is called *run time*. At run time, the program is supported by the *run-time environment*. Contemporary run-time environments provide linker features, thus enabling additional object files to be linked into a running program. This is commonly referred to as *dynamic linking* or *dynamic loading*.

Chapter 2

Limits of object orientation

2.1 Reuse

One often-touted feature of object-oriented programming is *reuse of code* [4, 14, 28]. More code is created as time progresses. More code is available to be reused and to ease the creation of new programs. Reuse of existing code saves time that would otherwise be needed to design, implement, debug, and test similar code.

In practice however, reuse of code appears to be rather limited. This is caused by the fact that the unit of reuse—the class—is also the unit of design: to reuse a class, its design must fit in the environment where it will be reused.

The reuse problem can be circumvented by the *traditional reuse*: copy the source code and modify it to make it fit the new design. However, the modification of source code is merely reuse of design and not reuse of code. The advantages of code reuse are absent: the modifications need to be designed and implemented, and the result after modification needs to be debugged and tested. Furthermore, any improvements and bug fixes to the original code will not affect the reuser, who copied the source code instead of linking the object code. Reuse of source is not reuse of code and, hence, not considered in this dissertation.

An obvious approach to solving the reuse problem, in an attempt to increase the level of reuse, is to increase the odds that a class will fit in the environment where it is going to be reused. The popularity of this approach is shown by the existence of various symposia devoted to it (e.g., [23, 37, 46]).

Much attention is paid to subjects like *domain analysis*, the underlying thought being that if a class is to be reusable, it must at least be reusable for applications in a single application domain. Another recurring theme is that of *reusable components* and the accompanying idea that, in the end, applications can be built solely from such components. However, a component is little more than a different name for a class, and domain analysis does not guarantee reusability of a class.

The problem of limited reuse is usually addressed by focusing on the development of classes, for instance through domain analysis. However the reuse problem is not merely a development problem. After all, the act of reusing is an act of consuming the fruit of prior labour, i.e., the classes that were previously developed. Yet reuse is an important issue to address: limited possibilities for reuse cause every software development effort to basically start from scratch. The same or similar abstractions are reinvented, redesigned, re-implemented, debugged, tested, and maintained. This wastes a lot of time and effort.

The next section (2.2) discusses various ways in which existing classes can or cannot be reused in contemporary object-oriented programming languages. Section 2.3 defines the problem that is addressed in the remainder of this dissertation. An overview of the dissertation is given in section 2.4.

2.2 Mechanisms

The easiest way to reuse a class is to use it without modifications. Most classes do not fit this scheme: they are developed as a subclass of their superclass precisely because the superclass does not suffice. Subclassing a class is an easy way to adapt a class for reuse. Unfortunately, this simple mechanism is not sufficient.

Problems with reuse stem from incompatibilities: a class provided by a library or component does not implement some desired functionality or does not conform to some desired type. In [7], an excellent example is given that concerns 'a text processing application [which] may add specialized tab-to-space conversion behaviour for strings and other collections of characters defined in the standard library.' Clearly such behaviour is too specialized to fit in a *string* class of a standard library. Equally clear is that such behaviour may very well be part of the string class in some text-processing application.

Below we consider some ways for adding this behaviour to the string class:

- *Subclass* the string class, i.e., create a subclass of the string class and use the new class instead of the original, throughout the application. Add the desired tab-to-space conversion behaviour to the subclass. A subclass is a specialized version of its superclass and in the application that is just what is needed: a specialized version of the general string class.

  Unfortunately, this approach fails in many circumstances and especially when reuse is at hand. When instances of the string class are created beyond the control of the user, e.g., in a library, the objects thus created will be instances of the general string class instead of the subclass.

- Implement the behaviour outside the string class, to operate on strings that are passed as an argument.

  This approach requires the string class to provide operations on which the approach can be based. For example, if the string class does not offer a possibility to insert and remove substrings, removing tabs and inserting spaces is not possible.

  In addition, this approach is like a return from object-oriented programming to structured programming, where programs are built from *abstract data types* (ADT) and functions that operate upon them. An ADT then resembles a class which only offers behaviour to read and write the instance variables. Clearly an undesirable approach.

- Employ a *wrapper*: for every string, also create a wrapper object. In new code, refer to the string through the wrapper. When invoking old code, which can handle strings and knows nothing about the wrappers, retrieve the string from the wrapper and pass it instead of the wrapper.

  New code maintains an object-oriented view of a string through the wrapper. The wrapper can implement the desired functionality to operate upon the string that it wraps, though it is hampered by the same restrictions as mentioned in the preceding approach. In addition, wrappers and their administration cost execution time and memory space.

In [21], various more mechanisms are described that attempt to address class incompatibilities during reuse. However, none of the solutions prove adequate.

2.3 The reuse problem

Existing object-oriented programming languages and design methods consider *the class* to be the unit of reuse (e.g., [4, 29]), and argue that a class can be subclassed to make it suitable for each problem at hand. That this assumption is not generally valid is shown in the preceding section. The assumption actually restricts reuse: the reuser is as free in reusing a class as allowed, envisioned, and *planned* by the developer of that class.

The class provides a mechanism for *planned reuse*. The advantages of considering only planned reuse are obvious: it implies a single-design approach in which the source code of everything is available for the programmer to modify, adjust, or fix; and for a compiler to read, extract information from, or otherwise digest. The source code is supposed to provide the necessary flexibility.

On the other hand, the disadvantages of considering only planned reuse can not be dismissed. Widespread reuse cannot be planned; it requires successful *unplanned reuse*. Widespread reuse implies extensive use of libraries, which lack the flexibility offered by source code.

In this dissertation, we plan for unplanned reuse. We examine reuse, reaching for the flexibility offered by source code while being restricted by source-code unavailability.

2.4 Overview

In chapters to follow, chapter 3 explores flexibility other than the flexibility of source code, as a key to unplanned reuse in object oriented programming languages: what is flexibility, what does it encompass, and what kind of flexibility is offered by existing programming languages. Chapter 4 describes the design of the TOM programming language, which has been developed to provide a level of flexibility not offered by existing programming languages. Chapter 5 describes an implementation of TOM by the author and chapter 6 reflects on the possibilities that are offered by the flexibility of TOM code. The conclusions of this dissertation form chapter 7.

Chapter 3

Flexibility of code

The flexibility of a tool is determined by its design. The design determines the many ways in which the tool can be used that were or were not envisioned by the tool's developers. This applies to any craft man's tool; the design of the hammer is worth mentioning in this context. It also applies to the design of the tool as a computer program.

Next to flexibility by design, there is flexibility by construction. The parts of a hammer can be reused in ways not envisioned by the hammer's creator. The parts of programs, libraries, and object files, are the pieces of code of which they are composed; flexibility of the construction is the flexibility of that code.

The flexibility of code determines the many ways in which the code can be used that were or were not envisioned, or *unplanned*, by the code's developers. Put differently: the flexibility of code determines the success of unplanned reuse. This chapter explores unplanned reuse through flexibility of code.

3.1 On the origin of code

A running program uses code from various sources. As shown in figure 3.1, underlying all code is the *hardware*, which is the machine on which the program is executing. The first layer of code is the *operating system* (OS), which abstracts programs from the hardware. The OS and hardware together are commonly referred to as a *computing platform*, or *platform* for short.

The next two layers, *libraries* and *program*, together form the *executable* that is run when the program is run. The libraries are used by many programs. The last layer represents additional object code that is not part of the executable proper, but linked into the program at run time. This kind of code is usually referred to as *plug-ins*.

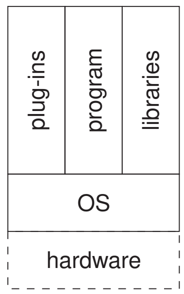

Figure 3.1: Four kinds of code in a running program.

In the following sections, we discuss these four kinds of code and their origins, as seen from the perspective of their respective developers. Since the subject of this dissertation is unplanned reuse, each of these kinds of code is assumed to be developed by a different team. When, for example, a team developing a program would also develop a library to be used in conjunction with the program, the library and program together would form a single design, and reuse of the library code would not be unplanned. Having separate teams for operating systems, libraries, programs, and plug-ins is common practice.

A consequence of considering different teams, which are separated in space and time, is the non-modifiability, and possible unavailability, of the source code of some parts of the final product. For example, a team developing a program can not modify a library used by the program. If the function `printf` in the C library were not to their liking, they could not change it, for it would need to be changed on all platforms providing an ANSI C library implementation.

3.1.1 The program

The program is that part of an executable written by the team developing the program. In general, the program depends on libraries (section 3.1.2), and together they depend on the services offered by the operating system (section 3.1.4). Furthermore, any plug-ins loaded depend on the program: the program *supports* many plug-ins (section 3.1.3).

3.1.2 Libraries

A program usually depends on several libraries. A library is available in the form of binary code, accompanied by an interface declaring the functionality that is offered by the library. The interface is used at compile time. As an example, an ANSI C library on a UNIX system consists of the `libc.a` archive of object files plus the interface in header files `stdio.h`, `stdlib.h`, &c.

A library can be *static* or *dynamic*: a static library is a collection of object files, for use by the linker when creating an executable. Dynamic libraries are also known as *shared* libraries: on a single computer their code is used by many programs, yet the actual code, i.e., the bytes making up the instructions, of the shared library is present only once. Linking with a static library adds the machine code from that library to the executable program. Linking with a dynamic library only attaches an *association* with the library to the program. This association is substituted by the library code every time the program is run. Consequently, when a new version of a dynamic library is installed, the program will automatically use the code of that new version. Put differently, the code in a dynamic library can change while the program executable does not. On the other hand, when a new version of a static library is needed, the program and library must be relinked to create a new executable, by the developer of the program so the program can be recompiled if needed.

Dynamic libraries are mostly useful on computers with many programs to employ them, or when the libraries are updated more frequently than the programs. They are usually not available on smaller machines with only a few or a single program, for instance embedded systems.

3.1.3 Plug-ins

A plug-in is object code that is dynamically linked into a *running* program. The plug-in extends the capabilities of the program, usually in an application-specific way that was anticipated by the program developers and part of the program's design.

In general, plug-ins for a program are developed after the program has been released, by programmers who do not have access to the program's source and who are not in a position to modify it. Requiring recompilation of the program to suit a new plug-in is impractical, to say the least. Similarly, a new version of the program can be installed without simultaneously upgrading all plug-ins, i.e., recompilation of the program must not require recompilation of the plug-ins.

Practical examples of various kinds of plug-ins are:

document processing Possibly the best known plug-ins are the processing filters offered by many a graphic-image manipulation program, where each plug-in offers some kind of processing of the image.

input filter Input filters enable a program to read file formats which it does not normally know how to handle. Input filters can be built into a program, but this obviously fixes the formats that can be read. Input filters can also be accommodated in plug-ins, with the advantage that they can be updated independently of the program and that it allows third parties to develop and release input filters for *their* favourite formats.

output filter Output filters allow a program to write files using various file formats. Their functionality is similar to that of input filters.

other There obviously is no limit on the number of kinds of plug-ins in addition to the above. It is interesting to briefly mention some of the more visible ones:

programming Many programming languages can be put to practical use by creating executable programs through compilation and linking. For some programming languages, the compiler is part of an environment in which new applications are written *and* deployed. When the environment itself is a running program, the applications thus developed can be regarded as plug-ins.

Outstanding examples of this paradigm are the Oberon system, in which applications can be developed using the Oberon language [45], and Emacs, which can be extended in Emacs-Lisp [39].

document generation Input and output filters convert a document in a program's internal data structures to or from some external file format. Not uncommon are plug-ins that generate documents, such as scanner plug-ins for an image manipulation program, which are used to control the scanning device and create new documents from the images thus scanned.

document display Mostly seen in various web browsers are plug-ins that enable the inline display of content that can be present in an HTML document but which the browser itself can not handle. Examples of such content are additional image formats and additional kinds of documents, e.g., moving images and virtual-reality models.

3.1.4 The operating system

Underlying all programs running on a machine is the operating system (OS) kernel. It can range from a full-featured UNIX kernel that manages a computer for multiple users, to a mere interrupt handler on an embedded system. Most programmers have no knowledge of it beyond some idea of the functionality it offers.

The programming interface to a kernel is usually very crude. It consists of system calls that are often implemented as processor traps or software interrupts. This is necessary because a system call requires a switch of the processor to a higher privilege level. Since such calls differ from one CPU model to the next, every OS comes with a library that enables the invocation of system calls through function calls in a popular programming language (like C).

The OS is generally not something one can choose; most often the target platforms are dictated by the application. When a program is developed to run under various different operating systems, one usually employs one or more libraries in the desired programming language to provide a common *application programming interface* (API) to abstract from the differences between those operating systems. For example, the IEEE POSIX 1003.1 standard defines a useful C API that is available on most modern operating systems [24].

An operating system kernel is normally not considered to be part of a running program; the two can reside in different memory address spaces for example. The kernel manages the machine on which the program runs and for security reasons it is not modifiable or extensible by or for just any program. Because of these reasons, and because of the availability of libraries in a high-level language that provide an easy interface to the OS kernel, in the remainder of this dissertation, the kernel is not further considered as a contributor of code in the running program. Where needed, the kernel will be referred to through its API library.

3.2 A taxonomy of code

Every program is written to depend on certain libraries: the program depends on functionality that is offered by the libraries. Similarly, every plug-in depends on a certain program. In the other direction: a library *supports* many programs, and each program may support many plug-ins. Figure 3.2 depicts this *code support relation*.

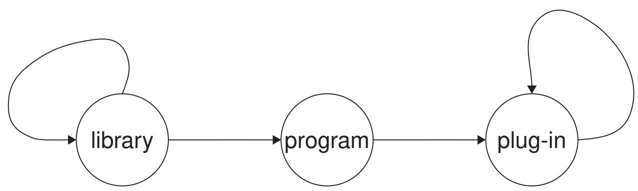

Figure 3.2: Code support relation.

An edge from node A to node B means that code from node A supports code in node B. The self loops express how a library can support other libraries and a plug-in can support other plug-ins.

The inverse of the code support relation is the *code dependency relation*: if code A supports code B, then code B depends on code A.

The absence of an edge from the library node to the plug-in node suggests that a plug-in can only depend on the program and not on any library. This is a mere simplification that corresponds to using a block diagram like in figure 3.3b instead of the diagram in figure 3.3a. The difference is cosmetic: of course a plug-in that is loaded into a C program may invoke the `printf` function that is provided by the C library.

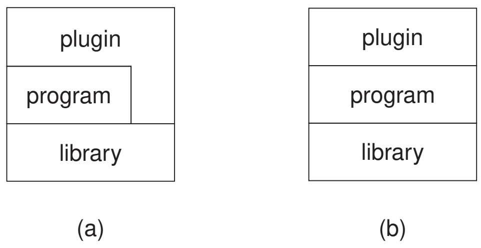

Figure 3.3: Transitive code support: explicit (a) versus implicit (b).

The transitive closure of the code support relation implies the following taxonomy of code, which is depicted in figure 3.4.

present code Code that is being developed by the *current programmer* who, by definition, has modifying source access.

future code Any code represented by the nodes that are reachable from the present code in the code support relation.

past code Code depicted by the nodes that are reachable from the present code in the code dependency relation.

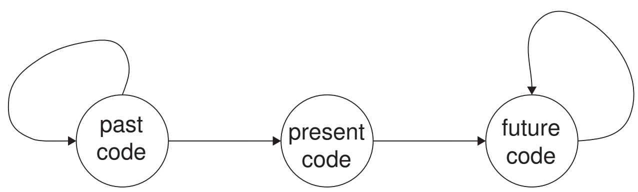

Figure 3.4: Past, present, and future code.

Any code, be it from the program, a library, or a plug-in, can be regarded as present code, which obviously influences the view on other code. This relation between the history of code and the taxonomy of code is depicted in figure 3.5. The top row shows the code taxonomy graph, partially unfolded; the three bottom rows show the code support graph, also unfolded. Depending on whether a library, the program, or a plug-in is considered as present code, the program will be regarded as future, present, or past code.

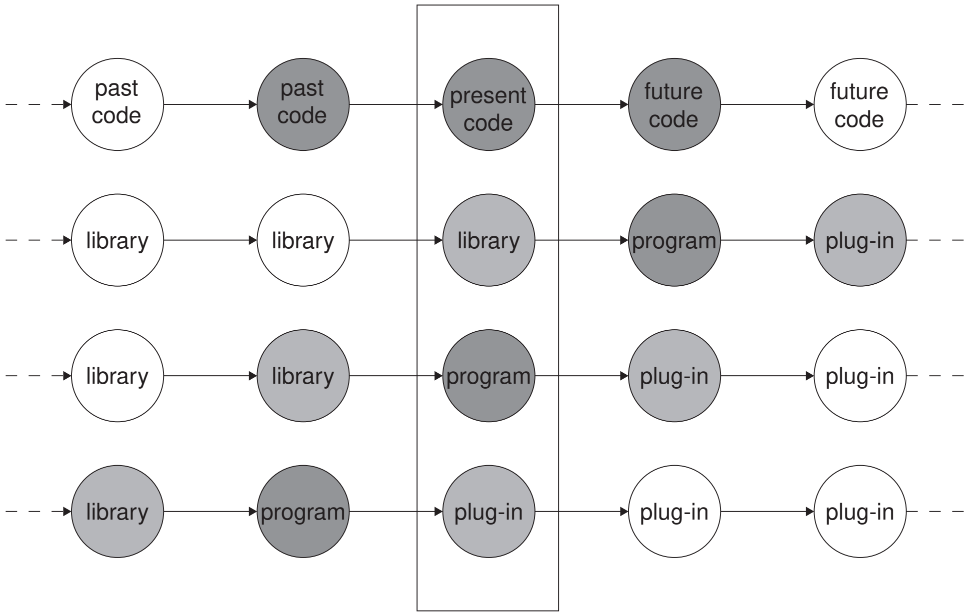

Figure 3.5: The notion of *present code* depends on the point of view.

3.3 Source boundary

During the development of a library, all future demands upon that library, by all programs that will be using it, cannot be known. In general, when developing present code, the demands by future code cannot be known. Hence, each forward traversal of an edge in the code taxonomy graph in figure 3.4 crosses a boundary of anticipation.

Similarly, each backward traversal of an edge crosses a boundary of modifiability: the developer of present code cannot adjust past code, like the plug-in developer cannot modify the program and the program developer can not modify the libraries. Such boundaries of modifiability are an example of *source boundaries*: a source boundary separates code with different levels of source code availability. A source boundary separates present code from past code.

The following levels of *source availability* can be distinguished:

1. Modifying source access.

This level of source availability is, by definition, possible only for present code, since only present code can be modified.

2. Full source access but without ability to modify.

   The sources can be read but not modified (modification is futile). This applies for instance to libraries as past code, as explained in section 3.1. This level can be referred to as *open source*; the programmer can read the source to learn from; a compiler can compile it. If modifications are desirable, the programmer can send patches to those who have modifying source access.

3. Availability of the interface only.

   The interface to the code is available (written in the same language as the source code); the implementation is only available as object code. This level can be considered as *closed source*. For example, it is the usual situation for libraries written in C, with the object code in `/usr/lib` and the interface to the library as header files in `/usr/include`.

4. No source access at all.

   Only object code or an executable is available. If any information is needed, such as an interface, it needs to be retrieved using reverse engineering techniques or a debugger.

It is clear that a source boundary influences reusability. In most programming languages the flexibility of reusing past code depends on the level of availability of the sources, even of past code.

Example

An example of sensitivity to source availability is Eiffel: in Eiffel, to create a subclass, the full source of all superclasses is required. This requirement is imposed by the inheritance semantics of Eiffel.

At the boundary between the program and a plug-in, the problem is worse. Dynamic Linking in Eiffel (DLE) [30] allows code to be dynamically loaded into a program. During compilation of this program, the compiler must know that dynamic loading will take place. This is because in the case of dynamic loading the compiler cannot run as a whole-program compiler and consequently cannot use its full optimization features. However, each time the application is recompiled, DLE requires the plug-ins to be also recompiled, thus requiring recompilation of all plug-ins when a new version of the program is released. Moreover, DLE allows only one plug-in to be loaded into a running program. Together, these restrictions make the use of plug-ins for an Eiffel program an uninteresting exercise.
□

When a linker creates a program by linking the object files and libraries, it can be aided, if necessary, by any other tool in the development environment. When a plug-in is linked into a running program, the development environment and all source code are absent; only the run-time environment (offered by the *run-time library*) is available. The run-time environment cannot deliver the same functionality as a development environment. As a consequence, the boundary between a library and a program is *friendlier* than the boundary between a program and a plug-in.

3.4 Language boundary

Reuse of past code depends on the programming languages that are involved. If past code was written in a language different from the present code, support by *glue code* is needed. Considering glue code as present code, it provides a means for future code to make use of past code. The future code is written in the *target language* and the past code was written in the *source language*. Glue code bridges a *programming language boundary*.

When glue code is needed, the possibilities for reuse of code are limited. As an example, consider the case of the source and target languages each being a different object-oriented programming language. The glue code will provide, in the target language, a proxy class for each class in the past code. Instances of a proxy class will wrap instances of the corresponding original class. However, the class hierarchies remain separate and subclassing a proxy class will not imply a new subclass in the past code: since the wrapping is unidirectional, newly introduced polymorphism in present code will not propagate back to past code.

This dissertation focuses on the issues of using a single programming language; the issues concerning programming language boundaries are not further considered. Dependence on past code on the other side of a programming language boundary can be expressed as a dependency on the glue code that is used to access that past code. Note that this exclusion coincides with the exclusion of the operating system from the code taxonomy: the kernel API library in section 3.1.4 is simply glue code.

3.5 Flexibility paradigms

Existing object oriented programming languages cater largely for planned reuse, as discussed in section 2.3. Everything in a design is planned and the sources are assumed to be as available and flexible as in present code. They do not discern between present and past code, ignore the difficulties in using past code, and do not take precautions for future code.

This problem has been studied before, especially in the context of the reuse of existing (past) code. This section gives an overview.

3.5.1 Type adaptation

Type adaptation [21] addresses unplanned reuse of *components*: a component is a collection of classes, and type adaptation allows the interface of a class or the classes to be adjusted. To this extent, type adaptation provides types to be added to the type hierarchy, operations to be added to types, and existing operations to be renamed. However, [21] imposes several restrictions:

- Though the type adaptation does not need access to the source code of the components, the components must be delivered in an intermediate format: this format does not need to be the source, but it should not be machine code. It would provide the necessary flexibility while still allowing a compiler to optimize after type adaptation.

- The new operations are not allowed access to the component's private details. While this allows the component to be replaced with a new version more easily—a public interface is less likely to change than some private implementation details—it does imply that a simple error by the component writer, like an omission in the public interface of an implementation detail that does not *need* to be private still limits the possibilities to reuse the component. (After all, the *need* for something being private is highly subjective.)

The requirement of an intermediate format makes the approach less interesting, or at least not directly applicable to existing languages. Type adaptation has not been applied to a real programming language or programming environment.

3.5.2 Extension hierarchies

In [31], the issue is addressed of extending existing classes instead of adding new classes. This is important when one wants to extend the behaviour of objects created by existing code, of existing objects (e.g., objects stored in a database), or instances of existing subclasses. The usual approach of editing the source code of the existing class is undesirable.

In [31] the concept of *extension hierarchies* is introduced. Such a hierarchy contains extensions to the base hierarchy, and is usually sparse. Extensions can be merged before application to the base hierarchy or they can be applied in sequence; conflicts are considered to be a problem and must be resolved. Issues involving an updated base hierarchy are considered too.

The ideas behind this work have been implemented as part of a C++ compiler from IBM [22] that provides extension hierarchies through *source composition*, a compile-time activity that requires all source code to be present. As a result, however, it can be applied only to present code.

Even though it does not provide flexibility beyond source code manipulation, this work represents a significant boost in the flexibility of C++. Unfortunately, it is available only as an unsupported extension of the particular compiler from IBM.

3.6 Extensibility

3.6.1 Extensibility operations

A programming language provides extensibility when present code can adjust behaviour that is defined in past code. For an object-oriented programming language, extensibility of past code implies extensibility of classes.

The level of source availability can be a parameter to the level of extensibility that is provided. When the only way to extend a class is to edit its source, extensibility it absent. When it can be extended without modifying its original source, some level of extensibility is present. When only the source code that defines the interface to a class is needed to extend the class, the level of extensibility is higher than when the full source of the implementation of all superclasses of the class is necessary for the class to be extensible.

The tab-to-space example in section 2.1 shows that the ability to *add* methods is necessary. To continue the example, suppose the developer of the standard library envisioned that a large number of text processing applications would use the library. This fact would be significant enough to justify the inclusion of tab-to-space conversion behaviour in the standard string class. Unfortunately, his implementation of the conversion behaviour would assume a tab stop every eighth column, being insufficient for some applications.

Suppose that in in our application, we need tab-to-space conversion with variable tab stops, which we can provide as *var-tab-to-space*. However, using a different name for this operation does not solve the problem, since the library-provided tab-to-space conversion is already invoked from past code. The only solution is to replace the existing tab-to-space implementation with our implementation of var-tab-to-space, i.e., to provide a new implementation for the existing operation.

The level of extensibility offered by an object-oriented programming language depends on the level at which it offers the following functionality for classes defined by past code, without imposing demands on past code, like requiring recompilation:

- Add methods.

  If a newly added method has the same method signature as an existing method, the new method replaces the existing method.

- Add state.

  Extra state information may be needed by newly added methods.

- Add a superclass.

  Adding a superclass to a class introduces additional methods and state for the instances of the class, and additional supertypes for them to conform to.

More intricate modifications to the inheritance hierarchy are possible than simply adding new superclasses, but they do not fit in a nice scheme as presented here. An example of such a modification is *class posing* in Objective-C (see section 3.7.2).

3.6.2 Extensibility time

In addition to the many ways in which code can be amended, i.e., the *how*, the moments in time at which such modifications are possible, i.e., the *when*, are another dimension of extensibility. The following moments can be discerned:

compile time The easiest of these situations: when a compiler has knowledge about any extensions, it can take precautions for it in the code that it generates.

link time Code has been generated without knowledge of any extensions, but the linker is still available as a tool to which decisions can be delegated.

run time The harshest of situations, where no compiler, no linker, nor any other tool is available. Extensibility has become the job of the dynamic linker and the run-time environment.

So far, the removal of methods and state from the instances of a class has not yet been mentioned. Since state can only be removed if all methods that depend on it have been removed, such removal is restricted to compile time, by a compiler knowing *all* methods accessing that state. Such knowledge is not available at link time or run time, hence state can only be removed at compile time.

Similar reasoning can be applied to methods: unused methods can possibly be removed at compile time. They can be ignored at any time.

3.7 Prior art in extensibility

Extensibility is available in various programming languages to a certain extent. For example, in Smalltalk the methods of a class are grouped in unrelated collections called *categories* [15]. A category can be added to a class, extending the class with the methods of the category. As another example, Oberon offers a complete operating environment which is extensible [18], though not in the sense defined in the previous section: instead, classes are rigid and programs developed in Oberon are plug-ins for the Oberon system.

This section discusses a few languages and the extensibility they provide.

3.7.1 C++

C++ code is not extensible, by design. It appears from the most recent edition of the C++ book [41] that everything that possibly can be done at compile time, should be done at compile time. Such an approach strongly reduces the flexibility of the resulting code since it is possible for one change in the source code to induce a myriad of changes in the object code.

An example of such far-reaching changes is offered by so-called *inline functions*: usually, a function invocation appears in object code as a function call; when an invocation is inlined, the object code for the invocation contains code of the function itself instead of a call. Function invocation inlining saves the overhead of the invocation, but when a function is changed *all* inlined invocations of that function must be recompiled for the change to fully take effect.

Being a popular language that has taken a long time to standardize, people have proposed many changes to C++. An interesting attempt at increasing the flexibility of C++ code was the suggested addition of *signatures* [2]. A signature is a collection of *method signatures*, i.e., declarations of, in C++ terminology, *member functions*. Every class that implements the necessary member functions *implicitly* conforms to the signature type. The signature type presents a new supertype that can be used as an abstraction from multiple classes that, without the signature type, are unrelated. The signature can be used to fit classes from different designs into one.

Example

Consider the following declarations:

```
signature foo
{
  void aap (void);
};
```

```
void f (foo&);
```

Furthermore, there is a class `bar` and the following function:

```
void g (bar& b)
{
  f (b);
}
```

Then the call to `f` with formal argument type `foo&` and actual argument type `bar&` is valid if, and only if, `bar` implements all member functions declared in the signature `foo`.
□

Signatures appear to provide a mechanism that is already provided by *pure virtual classes*, i.e., classes that contain declarations, and no implementations, of *virtual member functions* (C++ terminology for dynamically bound methods). Like a signature, such a class provides no implementation and is mostly used for the type it defines. The main difference between a pure virtual class and a signature is that classes must be inherited at compile time, whereas a class implicitly conforms to a signature, without recompilation being necessary. (Which is exactly why signatures are interesting, though apparently not interesting enough to be included in the ANSI C++ language definition.)

3.7.2 Objective-C

Objective-C is a combination of the C language with the object orientation of Smalltalk. Being modeled after Smalltalk, Objective-C has categories. A category can introduce new methods to a class and, depending on the particular implementation of Objective-C that is used, existing methods can be replaced [10, 36, 38]. For example, [36] allows methods to be replaced: methods in a category override methods in the class that it extends; dynamically loaded methods override methods that were present before loading.

Objective-C provides another extensibility mechanism called *class posing.* A class B can pose as its direct superclass A, which has the effect that every direct subclass of A actually inherits from B instead of A, with the exception of B itself. Posing is a run-time *trick* which is possible only because, in Objective-C, every method invocation is dynamically bound, even those to `super` (see section 4.5.4 on messaging `super`). Also class messaging is affected by posing: every message to the A class is actually passed to the posing class B.

When B poses as A, B must not introduce new instance variables. This restriction comes from the single-inheritance semantics of Objective-C: single inheritance implies that in addition to the state defined by a class, state originates from only one other source: the single superclass. This allows the state of each object to be administered as a straightforward `struct`, but it can not cater for the addition of state by B to the other subclasses of A, hence the restriction.

Example

As an example of the flexibility of Objective-C, though not using class posing, consider a situation where it is necessary to fix bugs in a class A defined in past code, but it is impossible to comply with the restrictions of categories or the rules of posing, for instance because a straight fix would need extra state in the instances of the class. When A does not have any subclasses and it is known that future code will not introduce any, for instance because the code when deployed remains under our control and we do not envision future code, a very nice hack can be performed.

In Objective-C, all classes inherit from the `Object` class. `Object` provides an *alloc* class method, which is inherited by every class and which is the designated way to allocate new objects. The hack is that it can be overridden to return something quite different from a fresh instance of A, as the category in the following example shows (in Objective-C syntax):

```
@implementation A (Foo)

+alloc
{
  return [MyFixedA alloc];
}

@end
```

In this case, one is concerned more with reusing past code that allocates instances of A than with reusing the class A itself.
□

3.7.3 Cecil

Cecil [7] is an object-oriented programming language that employs *multi dispatch*: in contrast to the more common single dispatch, where the method to be invoked is determined solely by the receiver of the message, with multi dispatch, the method is determined by the arguments. As a result, there is no explicit receiver and any combination of the arguments can be considered to be receiver (section 4.5.2 further discusses multi dispatch).

When multi dispatch is used, adding methods is easy since methods are not particularly bound to a single receiver or its class. Methods are dispatched on the arguments, irrespective of whether the argument classes involved exist in past or present code.

Cecil provides the extension declaration, which can extend an existing object, adding extra state or introducing additional superclasses. For this purpose, the source of the object does not need to be modified.

The designated Cecil compiler is Vortex [6], which is a whole-program compiler. A whole-program compiler decides statically whether the flexibility that is present in the program code is actually used at run time. For example, a message-send at a certain location in the code can be implemented by a direct method invocation if it can be shown that the class of the receiver does not vary at run time. The use of multi dispatch makes whole-program compilation almost mandatory if the resulting code is to exhibit some speed.

A whole-program compiler needs access to all source code. Furthermore, for a recompilation to not take too long, the compiler requires a delicate dependency analysis mechanism to enable incremental recompilation, where only those parts of the program are recompiled that are affected by a change in the source code.

The Cecil language provides ample extensibility that is also supported by the compiler, but any flexibility at run time is necessarily absent.

3.8 Résumé

The languages described in the previous section provide extensibility in some form or another. Of the languages surveyed, Cecil is the most extensible. However, due to the nature of the language, the Cecil compiler must remove as much flexibility from the code as possible, leaving no extensibility at run time.

The compile-time extensibility of Objective-C is maintained at run time, but the level of extensibility provided by the language is limited: Objective-C allows addition of methods, but the introduction of extra state and superclasses is not possible.

Full extensibility at compile, link, and run time does not appear to be offered by existing programming languages. Time and incentive to devise one!

Chapter 4

TOM: Design

This chapter discusses the design of the TOM programming language. The goal of TOM is to be an object-oriented programming language that provides flexibility of code as defined in chapter 3.

Code flexibility means extensibility of code at compile, link, and run time. Past, present, and future code are equal. Reuse is not restricted. The flexibility does not depend on the availability of source code.

4.1 Objectives

4.1.1 Application domain

TOM is a generic object-oriented programming language aimed at the implementation and maintenance of complex systems. Such systems feature the following properties:

execution speed During development, the size of code and its performance are largely irrelevant; a short turn-around time of the compile-link cycle is what is important. When the final product is to be deployed, code size and speed *are* of the utmost importance, and the run time required by a compiler to produce that code is irrelevant.

Fast compilation is easily provided by a simple compiler, without many optimizations. Fast code, or even faster special-purpose hardware, is delivered by a smart compiler, undoubtingly at the cost of very slow compilation.

restrained development time In today's world, early market presence is an important factor for the success of a product and, consequently, time to market should be short. Furthermore, software is a significant part of many products and it is becoming more complex with every new product that is developed. This makes writing the software both more important and a more difficult task to accomplish with every new product that is developed.

4.1.2 Target machine

TOM programs are not bound to a particular kind of hardware: nothing that can be written down in TOM has a meaning that depends on the particular compiler or target platform being used. The target can be an embedded system, a UNIX workstation, a Java Virtual Machine (JVM) [27], hardware, &c; the difference will not be apparent from looking at the source code.

TOM is intended to compare favourably to other object-oriented programming languages in use for the development of programs that run on personal computers. This includes languages that are used for similar goals or targets, such as Objective-C, Eiffel, C++, and Java.

4.1.3 Deployment

Deployment of code is the way in which code is used. When a program is run, irrespective of whether it contains bugs, the code of the program is used as envisioned by the program developer. However, when the program loads plug-ins, which were developed by other programmers, the program code has been past code to the developers of the plug-ins. It is very well possible that the developer of the program, i.e., the programmer of past code, has made assumptions about the program and its code, that clash with the needs of the programmers of the plug-ins and their code.

The problem is that the developer of present code can impossibly foresee how future code is going to use the fruit of his present labour. The developer makes certain assumptions about future code, and he can be wrong. The same problem also occurs with assumptions by library developers that appear not to be true for all programmers that want to use the library. Therefore, an objective governing the design of TOM is for the programmer of some code to not be able to annotate in code his assumptions about the deployment of that code.

A few examples illustrate the limitations induced by assumptions about the deployment of code.

Example

C++ provides the `virtual` keyword that can accompany member functions. Member functions that are `virtual` are dynamically bound; normal, non-`virtual`, member functions are statically bound. The common practice is that only when you expect a subclass to override a member function, the member function should be `virtual` [4].

However, there is another important difference between `virtual` and non-`virtual` member functions. Whereas `virtual` implies dynamic binding, thus enabling polymorphism, the absence of `virtual` *disables* polymorphism. As a result, if a programmer chooses to make a member function not `virtual`, he reduces the possibilities for useful redefinition by subclasses.
□

It is clear that if the developer follows some idea about the use of his code, he will be limiting the possibilities for reuse. Matters are worse when the envisioned use must be annotated in the code, like in the C++ example above. For a programming language to not restrict reuse, it must not provide or require such annotations.

Examples

The C++ `virtual` keyword has more reuse-influencing uses. In figure 4.1, a class D inherits from both B and C, each of which inherits from A. D inherits A twice: a case of *repeated inheritance*. When the base class A is inherited `virtual`ly by B and C, instances of D will carry the state introduced by A only once. However, when the inheritance of A is not annotated by `virtual`, the instances of D will carry the state of the base class A twice: once because of inheritance through B and once through C.

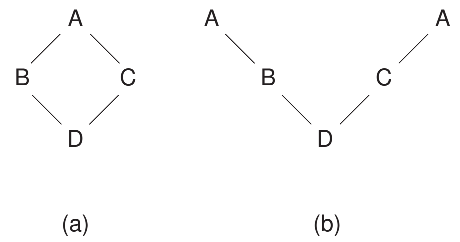

Figure 4.1: C++ repeated inheritance `virtual` (a) or not (b).

In this example, when D resides in present code and the other classes in past code, the example shows that decisions by past code about the layout of subclasses affect present code. The problem is that the present code can not undo or change those decisions, yet is affected by them.

In Java, the `final` qualifier, when applied to a class, means that subclassing that class is not possible. Java needs this restriction for the secure deployment of Java applets. For example, the Java `String` class is `final` [8] because a subclass might provide modifiable strings and strings that are modifiable during sensitive operations can introduce a breach of security. However, while the

`String` must be `final` for deployment as an applet, it proves an unnecessary restriction when used in a normal program.

The examples so far of annotated assumptions are all cases of modifying the semantics of certain constructs. An example of how assumptions dictate the time at which code is linked is given by Eiffel. In Dynamic Linking in Eiffel (DLE), a class that will be loaded at run time must inherit from *DYNAMIC* [30].
□

Programming languages employ other qualifiers than mentioned in these examples. For example, in many object-oriented languages, access to state and behaviour can be restricted by using `protected` and `private` qualifiers. However, the difference between for example a `private` method and a non-`virtual` method is that `private` is a mere access restriction whereas `virtual` changes the semantics. More importantly: in the light of extensibility, an access restriction is something that could be undone or otherwise circumvented, which is not true for a semantic change.

4.1.4 Extensibility

Extensibility is the *raison d'etre* of TOM.

Extensibility is the ability to modify features of a class, such as adding methods or state. When a certain feature influences code in one location, only one location must be updated when the feature is modified or replaced. When a feature influences code in one hundred locations, then one hundred locations must be updated to reflect changes in the feature. When all those locations reside in present code, location maintenance is a compile-time or compiler aspect. This situation applies to a whole-program compiler—which has access to all source code of the program it is compiling—or is, for instance, expressed in a `makefile` as a dependency of an object file on the implementation source file (`.c`) and any interface header files (`.h`) that it needs.

When the affected locations do not all reside in present code, problems arise. Examples are a change in a library feature that is used in several programs or when a program feature changes that is used by plug-ins. When static libraries are used and a program must be updated to use a new version of the library, with its new features, the program must be relinked to use the new library. Relinking is done by the developer of the program, who can recompile the program to incorporate those changes in the library that influence the program's object code. However, when dynamic libraries are used, the library code can change while the program remains the same: object code of the program will not be updated, even if it depends on the source code of a library feature that was changed, with possibly *catastrophic* consequences.

This problem is called the *fragile code* problem. A *line* of object code is always generated as the translation of a certain line of source code in some source file. Code is fragile if the object code depends on more than just that single line of source code. The more lines of source code influence that line of object code, the more fragile is the object code. In the other direction, the more lines of object code are influenced by a line of source code, the less easy it will be to adjust or replace that particular source code. This means that fragility hampers extensibility.

Examples

In C, the difference between interface and implementation, and the level up to which the implementation can be hidden, varies per language construct. For example, the `stdio` functionality can be offered perfectly well through the following definition of the `FILE` type in `stdio.h`:

```
typedef struct stdio_file_struct *FILE;
```

The exact contents of the `struct` is irrelevant for a compiler when invoking the `FILE` manipulating functions and can therefore be hidden from the user. Since code can not access the contents of the `struct`, it will not be fragile with respect to changes in the size of the `struct` or in the order of the `struct`'s fields or other modifications in the fields' type or offset from the base of the `struct`.

The following example shows a less strict distinction between interface and implementation. Consider a simple function, e.g., `printf`. We can read in the manual what it does and the C compiler reads in `stdio.h` how it can be invoked. However, in the object code, an invocation of `printf` will actually result in a subroutine call to the `printf` (or `_printf`) label: our invocation of some functionality through its interface has resulted in a unchangeable association with its implementation.

To circumvent this problem, C offers function pointers to separate function interface from function implementation, even without a syntactical difference in invocation, making them attractive to use. Furthermore, without using function pointers, linking with a different library can often be used as a link-time trick to override a certain library function with an alternative implementation. However, linker tricks are platform dependent. The precise effect on past code that also invokes the overridden function can not be known: it can range from the desired effect to no effect at all.
□

4.2 Design philosophy

Complementary to the design objectives of a language is the philosophy underlying the design. The philosophy underlying the design of TOM can be summarized as: *simplicity and freedom.* For instance:

- Employ a single mechanism for a single paradigm and variations thereof.

  For example, C++ signatures. (see section 3.7.1) introduced a new kind of types to which an object can conform: *signature types.* Objects that implement the methods declared by the signature type implicitly conform to that type. If the methods are declared by a class instead of a signature, the object must explicitly inherit the class to conform to the type defined by the class. Thus, apart from the implicit type conformance, signatures offer nothing that classes do not already offer. Therefore, signatures clash with the rule of a single mechanism for a single paradigm.

- Stay far from reasons of implementation.

  If a language feature can only be explained because of a reason that is concerned with compiler writing or other details of language implementation, the feature should not exist.

  To continue the previous example: even though the difference between signatures and certain uses of classes is the difference between implicit and explicit conformance, the difference is relevant, and consequently signatures are interesting, only because of how C++ compilers (must) implement C++.

  As another example, C++ non-`virtual` member functions do not exhibit polymorphism, whereas `virtual` member functions do. Yet non-`virtual` functions exist because of the implementation reason that when using a simple compiler, they are faster.

- Stay far from historical reasons.

  In many languages, things are done simply because they have always been done that way. An example is provided by the number of values that can be returned by a function. A mathematical function returns a point in a multi-dimensional space; in most programming languages, a function returns a single value, i.e., a point in the single-dimension space induced by its type. Exceptions exist, like Common Lisp where a function can return multiple values, but the caller must retrieve them separately if so desired [40]. At the same time, there is no good reason why not to allow multiple return values.

- Do not constrain the writer of future code.

  The writer of the superclass (past code) can in no way direct or restrict the possibilities of the writer of the subclass (future code). Put differently, a class is not closed; it is always amendable by a subclass or an extension.

- Do not constrain the writer of past code.

  The class must be amendable without making its use impossible. For example, a class in a shared library or in dynamically loaded code must be able to change, without affecting superclasses or subclasses. Put differently: it must always be possible to add behaviour and state to classes defined in past code.

4.3 Language basics

The remaining sections of this chapter discuss various aspects of the TOM programming language and how they support or advance flexibility of code through extensibility. As a start, the current section discusses a few language basics and gives examples of TOM code. Understanding these will aid in understanding the examples in the sections to follow.

Section 4.4 discusses the structure of objects, classes, the class hierarchy, and how objects can be extended and adjusted; section 4.5 explains methods. Section 4.6 discusses a few additional, relevant but otherwise unrelated, language topics.

Section 4.7 gives a short overview of features that are available at run time to further enhance the flexibility provided by the language; section 4.8 explains a few compile-time language features. Section 4.9 concludes with some remarks concerning missing features.

The syntax of TOM does not receive much attention, since concrete syntax is a secondary issue and it is only needed in this dissertation to make the examples legible. (It resembles that of C.)

4.3.1 Units

A TOM program uses TOM libraries. Each library is a *unit* and each program is another one. Each unit is implemented as a whole: all code in a unit is present code at the same time. A unit contains classes and provides a name space for their names.

Example

A `hello, world` example program in TOM consists of a unit with a single class. If this class is called `Hello`, it is contained in the TOM source file `hello.t`, and we call the program unit `hello`, then the *unit file* `hello.u`, which describes the unit, looks like this:

```
unit hello
{
  uses tom;

  file "hello.t"
  {
    class Hello;
  }
}
```

The clause `uses tom` expresses a dependency of this program unit on the library unit named `tom`. The `tom` unit is the TOM standard library; it is needed by every TOM program. Consequently, every unit depends on the `tom` unit.

The contents of the file `hello.t` is shown in an example below.
□

There is no difference between a library unit and a program unit, though usually the program unit is the unit that defines the *main method* (execution of a program starts with an invocation of the *main* method). Furthermore, the program unit usually depends on several library units, while none of the other units depend on the program unit.

Like the similarity between library unit and program unit, there is no difference between a program unit and the plug-in units that can be dynamically loaded. Usually a plug-in depends on the program unit, though that is not necessary. Since every unit ultimately depends on the `tom` unit, the program and plug-in share at least that unit.

Example

To complete the `hello, world` example, the file `hello.t`, which according to the unit file defines the `Hello` class, looks like this:

```
implementation class Hello

int
  main Array args
{
  [[[stdio out] print "hello, world"] nl];
  return 0;
}

end;

implementation instance Hello end;
```

To describe the meaning of the TOM code in this example, we start by observing that this file, `hello.t`, contains the implementation of the `Hello` class. Every class is fully defined by the state and behaviour of its class object and its instances. In our example program, we will not need instances of the `Hello` class, so their definition is rather empty. For the class object, no state is necessary and the only behaviour defined is the *main* method.

The name of the *main* method in the example is set in italics. To enhance readability for the eye that is not accustomed to TOM code, method name parts in the example code throughout this dissertation are typeset in an *italic font* as opposed to the `monospaced font` that is used for all other elements of code. This is for the purpose of readability only; typographic accents are neither used by nor necessary for a TOM compiler or programmer.

To continue the example, the *main* method returns something of type `int` and it accepts one argument of type `Array`. In this example, this argument is referred to by the name `args`.

Our *main* method contains three method invocations or message-sends. A message-send is written in square brackets `[...]`. The first expression within the brackets denotes the object being the receiver of the message; it is followed by alternating pairs of *method name part* and *argument*. When a method has no arguments, it has a single name part. The first method invocation in the example, of the *out* method, is such an argumentless invocation.

The *out* method is invoked of the receiver `stdio`. Since this code in `hello.t` is part of the unit `hello`, described by the unit file `hello.u`, which depends on the `tom` unit, a compiler knows that `stdio` is the name of a class and the *out* method invoked is a method of the `stdio` class.

The *out* method returns a *stream* object to which we *print* the constant string `"hello, world"`, which will subsequently appear on our screen. The stream objects defined by the standard library return themselves from a *print* method, and consequently, the third method invocation invokes the *nl* method of the same stream. The *nl* method emits a newline and flushes the stream.

As in many examples to follow, the meaning of those syntactic elements that are not discussed with the example, like the `return` and the `0`, is very similar to the meaning of the same constructs in C (or the similarity suffices for a correct interpretation of the example).
□

4.3.2 Basic types

TOM has two kinds of types: *basic types* and classes. The basic types are fully predefined; they can not be modified, amended, or extended by the programmer. Values with a basic type are always passed by value and operations on these values are statically bound. Contemporary CPUs are good at handling the basic types. The basic types are listed in table 4.1.

Table 4.1: Basic types

`byte`	8 bit unsigned int
`char`	16 bit unsigned int
`int`	32 bit signed int
`long`	64 bit signed int
`float`	single precision float
`double`	double precision float
`boolean`	values: *true* and *false*
`pointer`	generic pointer
`selector`	method signature
`void`	values: *void*
`id`	type of `self`
`dynamic`	anything

The numeric types `byte`, `int`, `long`, `float`, and `double` have the obvious meaning. The `char` type is a 16-bit unsigned integer, instead of the familiar 8 bits in most C implementations. The intended use of `char` is as character values from the 16-bit Unicode character classification [44].

The `pointer` is equivalent to the C *void pointer* (i.e., `void *`) and is mostly useful in glue code and in the implementation of classes that implement array-style collections or otherwise need pointers that do not point to objects.

Each type has a default value. The default value is implicitly assigned to new variables that are not explicitly assigned a value. Thus, the value of a variable is never undefined. This decision does not prevent all bugs and errors due to omitted initialization of a variable. It does however ensure repeatable behaviour, which is a great aid in debugging.

The default value of the numeric types is 0 or 0.0. The default value of a `boolean` is *false*. The default `pointer` is the invalid pointer (`NULL` in C); the default `selector` is the invalid selector. Object references default to the invalid reference, i.e., a reference to the invalid object. Any operation on that object fails (any message to it raises a `nil_receiver` condition, see section 4.6.1).

The invalid object reference is used frequently enough to have a name: `nil`. The default values of the `pointer` and `selector` do not have a name. However, the value of the following expression is the invalid pointer:

```
({pointer p; p;})
```

since the value of a *compound expression* is the value of the last expression in the compound.

The `void` type indicates the absence of a sensible value, similar to the type with the same name in C. The `void` type has only one value, which is noted as `void`. The `id` type is not really a type; it is the type of `self`. It is explained in detail in section 4.6.2. The `dynamic` type also is not a real type. It can occur as the formal argument type or return type of a method and it is used in implementing methods of which the argument type or return type, especially the number of elements of a tuple, can differ per invocation. An example of its use is given in section 4.5.3; tuples are presented in section 4.3.3.

Discussion

The notion and choice of basic types is not new; they are used in many programming languages. Notable exceptions are untyped languages like Lisp and Smalltalk. The notion of a machine independent range for integer types is also not new—Java uses the same strategy—but, surprisingly, not in widespread use: C, C++, and Eiffel do not define the range of the basic integer.

However, leaving the range of numeric types unspecified leads to assumptions by the programmer about the deployment of the code that he is writing. Even worse, these assumptions change over time. What starts as 'for 32 bits, use a `long`' becomes 'an `int` is 32 bits', and on contemporary machines, a `long` is becoming 64 bits wide.

Another possibly debatable point is how overflow of integer operations should be handled. Clearly, the untyped languages are *so slow* that an extra check for overflow does not cost a significant amount of time, while it adds the advantage of a silent switch to bignums. However, this flexibility comes at a high cost when the basic types and operations thereupon are meant to be handled directly by a CPU, and the basic types are meant to be low-cost and fast. In addition, not all CPUs can trap on overflow of integer operations. Therefore, TOM does not deviate from the approach taken by a lot of languages: arithmetic on integer values is modulo arithmetic.

Built-in operations

Operations on values of the basic types are expressed by operators. Operators constitute the only operations that are possible on values of the basic types and this also is the only use of operators. TOM employs the familiar C operators, with some differences (this list is not exhaustive):

- In C the arguments to the *short-circuit* boolean operators `&&` and `||` can be anything whereas they must be `boolean` in TOM.

- The result of the boolean-not operator (`!`) is `boolean`. The argument can be of any type. The result is *true* if the value of the argument is the default value of its type, *false* otherwise, which corresponds to its common usage in C. Thus, the following expressions are *true*.

  ```
  !0.0    !nil    !1 == !self
  ```

- The address-of and pointer dereference operators (`&` and unary `*`) have no use in a language without explicit pointers, and therefore they do not exist in TOM. TOM also does not employ the `,` (comma) as a sequence operator.

- The bitwise logic operators `&`, `|`, and `^`, bind stronger than the comparison operators, giving the expression `a & b == 0` the expected meaning.

- The operator `>>>` performs a logic shift right and is mostly useful on `int` and `long` arguments. The usual shift-right operator (`>>`) performs an arithmetic shift right on *signed* values.

- In C the indexing operator `[ ]` is a shorthand for pointer-arithmetic-and-dereference: `a[b]` is equal to `*(a + b)`. In TOM the meaning of the construct is different, the intention is the same, and it too can be explained as a syntactic shorthand:

  ```
  z = a[i];
  b[j] = y;
  ```

  are translated to

```
z = [a at i];
[b set y at j];
```

Note that the types of `i`, `j`, `y`, and `z` can be anything and `a` and `b` must be objects, making the indexing of associative arrays elegantly simple:

```
my_shell = environment["SHELL"];
```

4.3.3 Tuples

A tuple is a value that is made up of several components. For example,

```
(1, 3.1415, self)
```

is a tuple with as its type the tuple type

```
(int, float, id)
```

A tuple is an expression. Like every expression, it has a value and a type. However, tuple types are not *first-class types*: a variable can not have a tuple type as its type. (The reason for this is the same reason why expanded classes are dismissed, see section 4.4.1.)

The main use of tuples is in passing values to and from methods (see section 4.5). For example, in the following invocation

```
float i = ..., r = ..., a = atan2 (i, r);
```

the method *atan2* is invoked with one argument of type `(float, float)`. It returns a `float` value which is assigned to `a`.

The value of a tuple is computed by computing the value of each of its elements, from left to right. The value of the tuple is the tuple of the values thus computed.

In addition to being an expression, a tuple can also be used as the target of an assignment. This is only possible when each of its elements can be used as the target of an assignment. Obviously, simultaneous assignment behaves as expected:

```
(a, b) = (b, a);
```

(The values of `a` and `b` are swapped.)

4.4 Classes and objects

4.4.1 Classes

In addition to basic types, TOM employs classes, the types of objects. TOM employs an object model similar to Smalltalk [15]. As shown in figure 4.2, every object, instance and class objects alike, has a variable named `isa` being a reference to the object's class (*isa* is short for *is a*). Every object is an instance of its class and each class object is the sole instance of its meta class. To avoid infinite meta levels, all meta-class objects share one of them as a single meta-meta-class object.

Every class object contains information regarding all instances of the class: about the methods that they implement and the variables making up their state. The meta-class object contains the same kind of information concerning the class object. The meta-class object is implicitly defined by the inheritance mechanism; its behaviour or state can not be explicitly defined or extended.

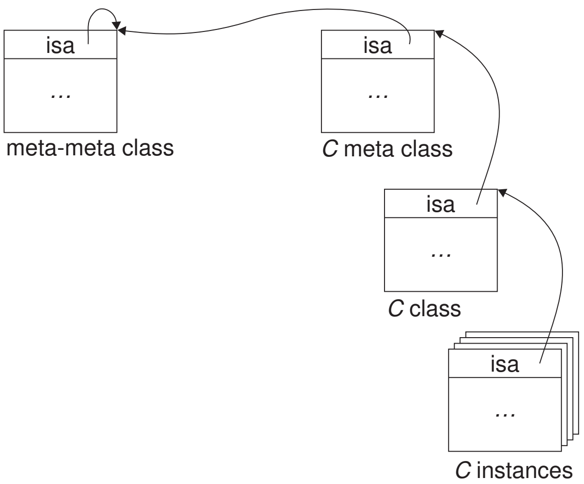

Figure 4.2: Instances, class objects, and meta class objects.

Every object definition actually defines *two* types: one denotes the instances of the class and one denotes the class object of the class.

Expanded classes and parameterized types

In the object-oriented programming paradigm an important classification of objects distinguishes objects that, when passed as an argument to a method or as a return value from a method, are passed *by value*, from objects that are passed *by reference*. When an object is passed by reference, the data making up its state remains at the same storage location and only a *reference* to the object is passed. Objects that are passed by value have their whole state passed: passing such an object as an argument in a method invocation actually passes a copy of the object. Consequently, the amount of data that is actually passed around is constant for pass by reference, whereas it depends on the type when passed by value.

Example

In Eiffel, objects that are manipulated by value are exactly those objects that are instances of an *expanded* class. Instances of normal, non-expanded, classes reside on the heap and are manipulated by reference. Expanded values have use beyond the archetypical Cartesian product types, like complex number and point-in-plane. Eiffel does not have basic types not being objects, like most other languages discussed in this dissertation have: in Eiffel, the basic numeric types are simply expanded classes.

In C++, the choice of by-value or by-reference is not made by the designer of a class like it is in Eiffel, but by the user of that class. This can undoubtingly be explained by C++'s C heritage: in C, every value can have its address taken and every pointer can be dereferenced. With good reasons: for example, to have a function return more than one value, all but one of the values must be passed by reference *to* the function. Therefore, creating a reference to (by taking the address of) a by-value instance is a normal operation, blurring the difference between expanded and non-expanded classes.
□

Instances of an expanded class are, by definition, passed by value. Program code that uses such an instance must have exact knowledge about the size of the state carried by that instance, much like a C compiler knows that the 32-bit general registers in the CPU can hold exactly one 32-bit `int`. This means that extensibility of an expanded class is limited: to add extra state requires recompilation of all client code and is therefore not possible. Furthermore, expanded values are mostly used for reasons of execution speed and all operations are therefore statically bound. Statically binding all operations not only saves execution time; it also saves state that is necessary to discern a class C from its superclass A, which is needed to support dynamic binding.

TOM does not have expanded classes, i.e., values of user-defined types are always manipulated by-reference. On the other hand, the significant speed advantage of expanded values and statically bound operations thereupon can not be ignored and TOM therefore employs expanded types with statically bound operations for those values that contemporary CPUs know how to handle, i.e., the numeric basic types presented in section 4.3.2.

An advantage of a fixed set of expanded types is the limited variation in the amount of state that has to be manipulated by code. For example, to provide an array abstraction to store all possible types, it suffices to create one array type for each of the basic types (with their varying storage requirements), and exactly one for all types of which the user-manipulated values are *reference to object*. Since the number of basic types is fixed and all user-defined types in future code (*future types*) are handled by *reference to object*, all possible requirements of future code can be handled with a fixed number of array types. This means that *parameterized types* are not necessary, other than for compile-time type checking. (A parameterized type can be regarded as a template for a type, which can be made concrete, at compile time, by supplying the type parameter, e.g., *array of integers*, *array of objects*.) Most importantly, when a new class is defined, it can be stored immediately in an array or any other kind of container defined by past code. It is not necessary to recompile the array or to perform partial compilation to specialize it to hold elements of the new type.

Garbage collection

A suitably high level of abstraction requires intimate knowledge about the lifetime of objects to be unnecessary. When such knowledge *is* necessary, the program code becomes fragile with respect to code changes which affect the lifetimes of objects.

For example, in C, *objects* are `struct`s that are allocated through `malloc`. When the lifetime of such an object has passed, its memory must be deallocated through `free`. Failure to do so correctly either leaks memory (the last pointer to an object has vanished but `free` has not yet been invoked) or accesses memory that should not be accessed (`free` has been invoked before the last pointer to the object has vanished). That this problem is real is shown by the popularity of *malloc debuggers*.

Availability of debugging tools does not solve or remove the problem of code requiring knowledge that violates encapsulation. Therefore, in TOM, knowledge about the lifetime of objects is not necessary and storage space of an object is reclaimed automatically after the object's lifetime has passed.

4.4.2 Object state

TOM provides the following kinds of object state:

instance variable An instance variable defines a value that differs per instance. Every instance variable defined or inherited by a class C adds to the state that is carried by instances of C.

The storage required for an instance variable is part of the instance's data (figure 1.1, page 3). Obviously, the number of storage locations required for an instance variable varies with the number of instances that exist in the running program.

class variable A class variable has a value that is the same for all instances of that class. When a class A introduces a class variable `cvar`, then the value of `cvar` will be identical for all instances of A. When a class C is a subclass of A then `cvar` has a different value for C than it has for A.

The storage required for a class variable is part of the data of the class object. The number of storage locations is therefore constant, except when new plug-ins are loaded into the running program.

static class variable A static class variable has a value that is the same for all instances of that class and its subclasses. Static class variables have a use similar to global variables in C.

The storage required for a static class variable is similar to the storage used by a global variable in C.

thread-local static class variable A thread-local static class variable (or *thread-local variable* for short) has a value that differs per thread and which, being a static class variable, has the same value for the declaring class, its subclasses, and all their instances. In a single-threading environment, a thread-local static class variable behaves like a static class variable.

An archetypical example of a thread-local variable is the `Thread` object representing the current thread.

A thread-local static class variable induces exactly one storage location for every thread in the running program and therefore the number of locations varies at run time with the number of threads.

At run time, every newly created storage location is set to the default value of the type of which values are stored in the location.

Example

A simplified example to show some concrete syntax of how every `Counter` has a `value`:

```
implementation class Counter end;

implementation instance Counter
{
  int value;
}
/* Method definitions go here...   */
end;
```

□

4.4.3 Inheritance

When a class A inherits from a class C, it inherits an interface and an implementation: instances of the subclass A can be manipulated by code that expects to manipulate the superclass C; the methods that the subclass uses to respond to messages are those inherited from the superclass.

Because of the strong association between the class and the instance, inheritance needs only be written down once in the source, since a mandatory repetition would be rather useless. For example:

```
implementation class Foo: State end;

implementation instance Foo end;
```

In this otherwise empty class definition, the class object `Foo` will be like the `State` class object, and the `Foo` instances will be like `State` instances. (Repeating the `Foo: State` at the `instance` is allowed but useless.)

Behavioural inheritance

The distinction between class objects and instances is useful but sometimes it is undesirable. In such situations, both the class object and the instances must conform to the same type. This mechanism is offered through *behavioural inheritance*, for example:

```
implementation class Aap: instance (Noot) end;

implementation instance Aap: instance (Noot) end;
```

In this example, the class object of `Aap` explicitly inherits the *instance* of `Noot`, as do the instances of `Aap`. With respect to the type `Noot`, the instances *and* the class object of `Aap` are now identical, i.e., they both conform to the type `Noot`.

The usefulness of behavioural inheritance lies in conformance to its type, the operations offered by the type, and the methods that the instance implements. The class of the behaviourally inherited instance is irrelevant.

The most visible use of behavioural inheritance is in the instance of the `All` standard class. It defines and declares various useful methods that apply to all objects, both class objects and instances, for example the following method which simply returns the receiving object:

```
id
  self
{
  return self;
}
```

Behavioural inheritance is akin to Objective-C *protocols* and Java *interfaces*. Like those mechanisms, behavioural inheritance groups method declarations and it introduces a type to which instances can conform. Unlike those mechanisms, it does not introduce a new mechanism and it introduces a type to which also class objects can conform.

Multiple inheritance

TOM allows multiple inheritance. Multiple inheritance introduces the possibility of repeated inheritance (see figure 4.1 on page 29). In TOM, the state that a repeatedly inherited class introduces is present in the subclasses once, as depicted in figure 4.1a.

Multiple inheritance also introduces the possibility of conflicting methods, e.g., when two methods of the same signature are inherited from different superclasses. TOM does not automatically resolve such conflicts; they must be resolved by overriding the method in the inheriting class. The overriding method can, for instance, select the desired overridden method from a specific superclass (see section 4.5.4).

Important classes

Several classes can be recognized in any TOM program, much like the standard classes in other languages. For example, in Smalltalk, the `Object` class resides at the root of the inheritance tree. TOM employs the following special types c.q. classes:

`Top` The implicit supertype of all object types. This type is not very useful, as it does not define any behaviour and can not be extended.

`Any` The implicit subtype of every object type: when used as the return type of a method that can return any object, the caller never needs to cast the value that is returned. For example, the following method is the only object retrieval method of the `ObjectArray` class (which offers a read-only array abstraction that stores object references) that actually directly retrieves an object:

```
Any
  at int index;
```

`All` The conventional supertype of all object types. All classes *should* either inherit from `State` (see below) or both class object and instances *should* inherit from the `instance (All)`. The instance `All` defines all kinds of behaviour that is useful for *all* objects, both class objects and instances.

Being the supertype of all objects, `All` can be used as the type of a formal argument, allowing any object type to be passed as an actual argument, without needing a cast. This is used, for example, by the only method of the `MutableObjectArray` class (which offers a read-write array abstraction that stores object references) that actually directly modifies the array:

```
void
  set All object
   at int index;
```

`State` Every class must inherit from `State` for the instances to be allocatable. `State` is also the class that defines the `isa` object variables and that provides the designated way to create new instances, namely through the *alloc* class method.

The `instance (All)` is the conventional supertype of all objects, a fact that is visible in the definition of the `State` class:

```
implementation class State: instance (All)
...
end;

implementation instance State: instance (All)
...
end;
```

The usefulness of the `Top` and `Any` types is restricted to compile time: they do not represent real objects that can be allocated or extended. The pervasive presence of `All` enables the addition of behaviour to *all* objects, not discriminating between instances and class objects, simply by extending the `All` instance (extensions are explained in section 4.4.5). Similarly, behaviour can be added to all classes *or* all instances, simply by extending the `State` class *or* the `State` instance.

4.4.4 Encapsulation

Encapsulation denotes the hiding of implementation details. This applies to object code as much as to source code. An object's state is a prime example of an implementation detail. When code directly accesses an object's variables, it becomes fragile with respect to changes to those variables. Therefore, TOM allows direct access to the state of an object only to the object itself. When the object is a class object, direct access is also granted to its instances.

As a result, access to the state of object other than `self` is never direct, neither in source code, nor in object code. This implies that to access some object's state, it must provide methods to do so. To prevent the tedious task of creating many little methods to access instance variables, like

```
int
  foo
{
  return foo;
}
```

the qualifier `public` exists. It has the effect of defining such an *accessor method*. Thus, the following declaration of the instance variable `foo`

```
public int foo;
```

grants access to the object variable `foo` through a method named *foo*, which can be used thus:

```
int f = [x foo];
```

Similarly, the qualifier `mutable`, when applied to the object variable `foo`, defines the following *modifier method*:

```
void
  set_foo int f
{
  foo = f;
}
```

and the object variable `foo` of an object `x` can be assigned thus:

```
[x set_foo 42];
```

Discussion

Object-oriented programming languages like C++ and Objective-C employ the qualifiers `public`, `private` and `protected`. An instance variable that is `public` can be read and written directly by methods belonging to any instance of any class, instead of only the instance to which the variable belongs. When an instance variable is `protected`, it can be directly manipulated by methods of the declaring class and subclasses; when it is `private`, only methods of the declaring class can access it.

Obviously, TOM gives a rather different meaning to the `public` qualifier, though the intention is the same—enable access from other objects. Furthermore, access being `protected` is natural for TOM and `private` is useless, since, given the extensibility in TOM, one can always define an extension to retrieve or modify a `private` variable.

4.4.5 Extensions

A class is defined by its main definition and its extensions. The main definition is often referred to as the *main extension*. All extensions, apart from the main extension, have a name that allows the extensions of the same class to be distinguished; they are collectively known as the *named extensions* of the class.

As an example, adding a method to a class A can be done simply by defining the method in an extension:

```
implementation class A extension Foo end;

implementation instance A extension Foo

int
  age
{
  return 5;
}

end;
```

In this example, the extension `Foo` of A defines a method '`int` *age*'. If this method is already defined for A, then this new definition overrides the previous one.

A method definition in a named extension always overrides a method with the same signature in the main extension. Similarly, an extension that is loaded at run time has precedence over the extensions that were present before the extension was loaded. Extensions that are not ordered by different load times, are ordered according to the dependencies between their containing units, such that an extension by past code can be overridden by future code.

Apart from adding and replacing methods, an extension can also add variables and superclasses. In this respect, all extensions, named or not, are equal.

Discussion

Named extensions can be added to a class at compile, link, and run time. Though the real use of extensions is at link and run time, even at compile time they offer an interesting advantage: they enable the definition of a class to not be contained in a single source file. The class definition can be distributed over multiple source files. Similar functionality is offered by the implementation of extension hierarchies for C++, as discussed in section 3.5.2.

4.4.6 Class posing

TOM provides class posing, inspired by class posing in Objective-C (discussed in section 3.7.2), with one important difference: posing is part of the language instead of a trick at run time. Figure 4.3 shows the effect on the class hierarchy of a class B posing as its superclass A. In addition to changing the superclass of the other subclasses of A, any reference (sic) to A will actually reference B, both in a type and in sending messages to the class object.

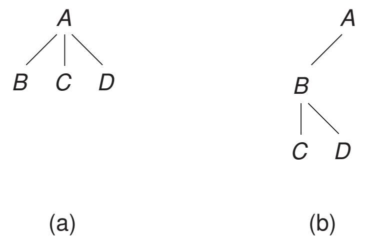

Figure 4.3: (a) B inherits A; (b) B poses as A.

The advantage of posing as A over extending A is simple: whereas an extension can *replace* methods of the class it extends, a subclass can *override* methods of the class it inherits (irrespective of whether the subclass poses as the superclass or not). A method that has been replaced is no longer available, whereas a method that is overridden can still be invoked, by *messaging super* (see section 4.5.4).

Being part of the language instead of a run-time trick, knowledge about classes being posed is available at compile time, which means that the types of objects and methods can be deduced correctly. This saves a lot of explicit casting, as the example below shows.

Note that for the modification of the inheritance hierarchy by class posing, it is irrelevant whether the other subclasses of A are defined in the same unit (present code) or a different unit (past or future code).

Example

Suppose we take a closer look at the classes A and `B` of figure 4.3b (ignoring the other subclasses for the moment). The source code of `B` could look like what follows, indicating that `B` inherits from and poses as A, plus that `B` adds method '`int` *age*', that we assume is not implemented by A.

```
implementation class B: posing A end;

implementation instance B

int
  age
{
  return 5;
}

end;
```

Since posing is a mechanism that provides additional extensibility, allowing the superclass A to be adjusted, A will usually be defined in past code, i.e., in a different unit than the unit containing B. While compiling the unit containing A, the compiler will not have knowledge of B, and many a method will return an A. While compiling the unit containing B, the compiler knows about B posing as A and it will correctly type and accept an invocation of the *age* method of an instance of A, even though A itself does not provide that method:

```
A a = [foo next_a];
int i = [a age];
```

Had the fact that B poses as A not been known, then the invocation of *age* would not have been possible, since the type of `a` is A, and A does not provide in implementation of a method '`int` *age*'. Instead, a cast would have been needed to convert the type of object `a` to B, which does provide an implementation of the requested method:

```
int i = [B (a) age];
```

□

4.5 Methods

4.5.1 Method definition

Methods are the unit of behaviour. A method is defined by its name, the argument and return types, and its body. The TOM method name syntax resembles that of Objective-Cand Smalltalk. Either a method has no arguments and a simple name or it has one or more arguments, each preceded by a *method name part*. Every argument has a type and a name. Table 4.2 shows a few example method declarations.

Table 4.2: Examples of method declarations.

declaration	meaning
`double` *pi*;	*pi* is an argumentless method that returns a `double`.
`int` *main* `Array args;`	*main* accepts an `Array` argument named `args` and returns an `int`.

`(int, int)`
  divmod `int a`
    by `int b;`
	divmod by is a method with two
	arguments `a` and `b` of type `int`;
	it returns a tuple of two `int`s.

The value that is returned from a method can be a tuple and is therefore not restricted to be a single value of a particular type as is common in many programming languages. Also, returning multiple values is not different from returning a single value, which it is in Common Lisp [40].

Tuples have more uses than as a return type. To avoid the excess method name parts that often occur in Objective-C and Smalltalk, an argument can also be a tuple:

```
(int, int)
  divmod (int, int) (a, b)
{
  return (a / b, a % b);
}
```

A method belongs to particular objects, either the instances of a class or the class' class object. A method is invoked as a result of sending a message to such an object. The receiver and selector of the message are available to the method body as the implicit arguments `id self` and `selector cmd` respectively.

Returning from a method

This section explains how the return value of a method can be set. It serves as an illustration of the design of TOM. This aspect of the language does not influence extensibility.

A method body is a compound expression (enclosed in braces, as in { ... }). The value of that compound expression is irrelevant. To return a certain value, e.g., 42, from a method that returns an `int`, the following suffices:

```
return 42;
```

Apart from setting the value that will be returned to 42, `return` has the side effect of terminating execution of the method. Since that side-effect is not always desirable, TOM offers a *return-value assignment*, the unary `=`, as in

```
= 42;
```

which does not exhibit the side effect of terminating the method. This is useful in situations where the return value is known before all resources are released, locks unlocked, &c. For example, instead of

```
int v = [my_resource value];
[my_lock unlock];
return v;
```

one can use

```
= [my_resource value];
[my_lock unlock];
```

and more exciting variations thereof.

A third mechanism of setting the return value involves *named return values*. In the *method heading*, these names are defined by a tuple of identifiers following the return type; a single return value can be named by a singleton tuple. Each identifier in the named return values tuple either names an argument or is the name of a local variable that is implicitly declared with the indicated return type. For example:

```
(int, int) (quotient, remainder)
  divmod (int, int) (a, b)
{
  quotient = a / b;
  remainder = a % b;
}
```

In addition, when `return` is used without any arguments, execution of the method is finished and the return value is not affected.

The real use of named return values (and their reason for being) is in method conditions, which are discussed in section 4.8.2.

4.5.2 Method invocation

As explained in chapter 1, the conceptual model underlying a method invocation is that of sending a message. A message has a receiver and it carries a selector plus any arguments that are needed. It is up to the receiver of the message to properly respond. In terms of code, it is the class of the receiving object that determines which code will be executed in response to the message.

The scheme described here is known as *single dispatch*: only the class of the receiver determines which method will be invoked. A language like Cecil employs *multi dispatch*: a message-send does not have a receiver *per se*: the method to be invoked depends on the classes of *all* object arguments.

Example

As an example of multi dispatch, consider a `Circle` and `Rectangle` that can draw themselves on a `Display` (e.g., draw a certain shape on a screen) and, in addition, the `Rectangle` can do smart things on a `ParticularDisplay` (and the `ParticularDisplay` is a subclass of `Display`). This is expressed by the following methods (this example code is not valid in a particular language):

```
void draw (Circle, Display);

void draw (Rectangle, Display);

void draw (Rectangle, ParticularDisplay);
```

Thus, to draw a `Circle`, the first method will be invoked, irrespective of whether it must draw on a mere `Display` or on a `ParticularDisplay`. On the other hand, when drawing a `Rectangle`, the method to be invoked will depend on whether the `Rectangle` will draw on a `Display` (second method) or a `ParticularDisplay` (third method).
□

An important disadvantage of multi-dispatch method binding is that code for an invocation actually inspects the classes of the objects involved, thus making the code fragile with respect to changes in the classes of the objects involved or in the available methods. A conceptual disadvantage of inspecting the actual class of the arguments is that the distinction between interface and implementation of the objects becomes blurred.

The fragility of multi-dispatch method binding is unsuited for run-time extensibility. Therefore, TOM employs single-dispatch dynamic method binding.

Example

The dispatch flexibility that is offered by multi dispatch can also be obtained when using single dispatch. As an example, we can rephrase the `Rectangle` drawing example in TOM, while safely ignoring the `Circle` since the shape (either `Circle` or `Rectangle`) is the receiver of the *draw* message. The *draw* method of the `Rectangle` can tell the argument `Display` what to do:

```
// in Rectangle:
void
  draw Display d
{
  [d drawRectangle self];
}
```

and the `Display` and `ParticularDisplay` reply as follows:

```
// in Display:
void
  drawRectangle Rectangle rect
{
  [rect drawOnDisplay self];
}

// in ParticularDisplay:
void
  drawRectangle Rectangle rect
{
  [rect drawOnParticularDisplay self];
}
```

with the obvious implementation by `Rectangle` of the methods *drawOnDisplay* and *drawOnParticularDisplay*.

The advantage of this setup when compared with multi dispatch is the extensibility: adding a new kind of `Display` does not invalidate any invocation of the *draw* method that has already been compiled. This is, of course, an important aspect when using plug-ins, and a prerequisite for run-time extensibility.
□

4.5.3 Method overloading

Methods are distinguished by their signature: when two methods differ in name, return type, or type of an argument, they are considered to be different methods. This is called *method overloading*: the method name is used for multiple methods. For example, the following method declarations all denote different methods.

```
void
  foo;

int
  foo;

id
  foo int bar;
```

There is, however, one exception: methods that only differ in the type of an object argument or return value are considered equal. If methods were discerned on object types, it would offer not much more than implicit additional parts in a method name. If methods were discerned on object class, we would arrive at multi dispatch, which was already discussed and dismissed in section 4.5.2.

Ambiguities in method invocations are resolved at compile time through a few simple rules. As an example of such a rule, an actual argument type of `int` matches a formal argument type of `long` better than it matches a `float`. True ambiguities are reported as an error. Since overloading is a compile time issue, method-overloading ambiguities do not exist at run time.

Discussion

Method overloading eases reuse. When past code contains, for example, a method with the signature '`void` *display*', then it can not claim exclusive use of the word *display* as a method name. Overloading enables using that name again in an extension, for example in a method '`Display` *display*' (using the word *display* as a noun instead of a verb). However, method overloading obviously does not solve the general problem of clashing method names when using past code from multiple sources.

Method overloading should be used sparingly, since overzealous application easily leads to a strong reduction of code readability. On the other hand, however, there are situations where it greatly enhances readability, as the example below shows.

Example

The `OutputStream` class in the `tom` standard library unit provides several methods to print various types of values to the stream. These methods translate the value into a sequence of bytes readable by humans; these bytes are subsequently written to the stream. An attempt to avoid overloading, thus explicitly mentioning the argument types in the method name, could result in the following code:

```
[out printByte 'H'];
[out printDouble 1.234d56];
[out printLong -1L];
[out printObject nil];
```

This is not only rather ugly, but also requires a change of the method name when the type of the value is changed. In addition, one must know the exact type of an expression when invoking the method, which is unfortunate for such a frequently used action as printing.

In reality, the `OutputStream` class overloads its print methods, which are all called *print*. They return `self`, thus enabling *invocation stacking*, as in:

```
[[out print "The value of pi resembles "] print 3.14];
```

The most useful of the *print* methods accepts any argument:

```
id
  print dynamic args;
```

Since the formal argument `args` has the type `dynamic`, the type of an actual argument can be anything, most notably a tuple, as this example shows:

```
[out print ("The value of e resembles ", 2.71)];
```

In the invocation of a method with a `dynamic` argument, the selector that is passed in the message-send will carry the actual type of the argument. As an example of constant selectors, the selector that corresponds to the above example is, in TOM syntax (a selector constant denotes a method signature and resembles a method declaration without mentioning the argument names):

```
selector (OutputStream print (String, float))
```

Since methods are not discerned on the exact type of object arguments, the following constant denotes the same selector:

```
selector (All print (Any, float))
```

Obviously, the '*print* `dynamic`' method performs an invocation of the proper *print* method for each of the values in the tuple that is passed as an argument. In fact, the overloading-resolution rules of a TOM compiler force an invocation like

```
[out print 1.6e-19]
```

to invoke the method '*print* `float`' and not '*print* `dynamic`'.
□

4.5.4 Messaging `super`

A subclass can override a method that it inherits from a superclass. When the corresponding message is sent to an instance of the superclass, the method invoked will be different from the method invoked when the message is sent to an instance of the subclass. From within the overriding method (in the subclass), the overridden method (in the superclass) can still be invoked by sending a message to `self` while acting to be an instance of the superclass. This mechanism is provided by *messaging super*.

The following example is a method defined for instances of `Subclass`, making them always return twice the *age* that would have been returned without this method:

```
// in Subclass:
int
  age
{
  = 2 * [super age];
}
```

Since a class can have multiple superclasses, a message to `super` can be ambiguous. Such an ambiguity can be removed by explicitly indicating which superclass is to deliver the desired response, for example:

```
// in Subclass:
int
  age
{
  = 2 * [super (Superclass) age];
}
```

Ambiguities when referencing `super` are only observed at compile time; at run time, the message to `super` is directed to a certain direct superclass, and ambiguities with respect to which particular superclass is to respond do not exist and can not be introduced by dynamic loading. Of course, messages to `super` are dynamically bound just like normal method invocations. Furthermore, a message to `super` is not restricted to be an invocation of a method that is overridden by the current method, though such use is the normal use. For the invocation of other methods, `self` as the receiver is more suitable and, more importantly, open to implementation by subclasses.

4.6 Miscellanea

This section groups the discussion of a few unrelated language features, which are relevant to the issue of extensibility.

4.6.1 Conditions

A condition flags an unusual circumstance. A condition is an object that can be told to *signal* itself; this triggers the condition-signaling mechanism: the condition is offered to every *condition handler* that has *a priori* indicated interest in that particular kind of condition. A condition handler is an expression in a method; it has a body and is active during the evaluation of its body.

A condition handler is an expression that returns an object. When invoked, it can take one of the following actions:

1. Return the condition object. The signaling mechanism will continue to search for matching handlers.

2. Return an object, which will subsequently be returned from the *signal* method invocation activating the condition-signaling mechanism. This action terminates the signaling mechanism.

3. Perform a non-local return to the method invocation containing the handler or its callers (direct or ancestral). This terminates the signaling mechanism.

A condition can be told to *raise* instead of *signal*, in which case the signaling mechanism will only stop on a non-local return. A condition is usually raised by code that can not usefully continue execution.

Discussion

Conditions in TOM serve the same purpose as exceptions in C++ and Java [41, 17], and conditions in Common Lisp [40]. They are not as baroque as conditions in Common Lisp, yet fix a problem of exceptions (see below) big enough to justify the name *conditions*.

An exception in C++ and Java is *thrown*, to be *caught* by handlers. When a matching handler is found, it is executed, but only after the stack has been unwound to the context of the handler. This compares roughly with TOM conditions that would only be raised and never signaled.

In situations where continuing execution after signaling a condition makes sense, exceptions are not an adequate tool to indicate problems, since after an exception is thrown, there is nothing to return to. Unfortunately, these situations are common, for instance every method that returns an object: upon failure to create the object, it can ask a handler what to do next and the handler can suggest a replacement object, for instance `nil`.

Java introduces another problem for its exceptions. Java discerns checked and unchecked exceptions. Each checked exception that can be thrown during execution of a particular method must be declared as such in the *throw clause* of that method, which is part of its declaration. If a method is implemented that has been declared elsewhere, for instance in a superclass, then the exceptions that can be thrown must not differ. This forces local exception handling upon the method, which is contrary to the objective of exceptions: they enable non-local error handling.

Checked exceptions must be mentioned in a method's throw clause, a fact which is checked by a compiler. When thus enforced they restrict extensibility and negatively influence the validity of source code for no good reason. That checked exceptions are unwieldy is implicitly acknowledged by the creators of Java: they invented unchecked exceptions to flag exceptional circumstances that are triggered by the program but not actively thrown by the program code, for instance `OutOfMemoryError` and `NullPointerException` exceptions.

4.6.2 The `id` type

The `id` type has been presented in section 4.3.2 on basic types, though it is not really a distinct type. The `id` type denotes the *current type*. In the context of a method definition, the current type is the *formal* type of `self`, i.e., the class or instance containing the method definition. However, in the context of a method invocation, `id` denotes the *actual* type of the receiver of the message, as the following example explains.

Example

Use of the `id` type is best illustrated by the object allocation mechanism used in TOM. A new object is created by invoking the *alloc* method of its class, followed by an invocation of the object's *initializer*. The *de-facto* initializer has no arguments and is defined by `State` thus:

```
id
  init
{
  = self;
}
```

So, this default initialization method does nothing.

Within the method body of this *init* method, which is defined by `State`, it does not matter whether the return type is `State` or `id`, since the two are equal. Things are different for an invocation of this method, and the types change when taking subclasses into account. Suppose we have a class `Shell` which is a subclass of `State`:

```
Shell s2, s1 = ...
s2 = [s1 init];
```

The *init* method invoked is inherited from `State`. Consider this example if the *init* method would return `State` instead of `id`. In that case, the assignment to `s2` would not be possible—without a type cast—because of the types being incompatible. Luckily, the type returned by *init* is `id`, being the type of the receiver, the type of `s1` in this case. The assignment is thus correct.

By definition, new objects are allocated through the *alloc* method defined by the `State` class. This method, when invoked, returns a newly allocated instance of the class to which the *alloc* method was sent. This *alloc* method provides the only way to create new instances. Thus, if instances of a class will be created, the class must be a subclass of `State`.

The *alloc* class method is declared thus:

```
instance (id)
  alloc;
```

The `id` type *changes* with the receiver. With the allocation of instances, the type of the object returned is the type of the instances of the class that was the receiver. Hence the type of the value returned by `alloc` is `instance (id)`, i.e., an instance of the receiving class.

Object allocation and initialization methods are mostly invoked in one breath, thus:

```
Shell sh = [[Shell alloc] init];
```

Having to type two method invocations to create a new instance can become cumbersome. Therefore, a class can provide a single method to perform both allocation and initialization. As an example, a `Sea` class of objects that hold one or more `Shell` objects could provide the following *allocator* class method:

```
instance (id)
  withShell Shell sh
{
  = [[self alloc] initWithShell sh];
}
```

In addition to a default initializer, `State` also provides a corresponding default allocator:

```
instance (id)
  new
{
  = [[self alloc] init];
}
```

As the following code shows, using the allocators makes the code shorter without reducing readability:

```
Sea one = [[Sea alloc] initWithShell [[Shell alloc] init]];
Sea two = [Sea withShell [Shell new]];
```

□

Example

Shifting an object type from a class to its instances can of course also be reversed, as used by `State`, which defines the following instance variable:

```
class (id) isa;
```

Thus, `isa` is a reference to the class object of the *current* object. While defined only once, in the definition of `State`, its type `class (id)` is correct in all subclasses.

□

4.7 Run-time flexibility

The extensibility features of TOM are discussed in the preceding sections. Those features are provided by the language and maintained at run time, aided by the run-time environment. Some additional features of that environment can not be left unmentioned, for they are instrumental in the run-time flexibility that TOM provides.

4.7.1 Computed method invocation

In the example message-sends in preceding sections, the selector is always fixed by code. The ability to parameterize the selector of a message is provided by the *perform with* method (and related methods), defined by the `All` instance:

```
extern dynamic
  perform selector sel
    with dynamic arguments;
```

When invoked, *perform with* will send a message to the receiver of the *perform with* method, with the selector `sel` and as arguments what is passed to the `arguments`. This is a run-time exercise: any typing error in the arguments or return type can not, in general, be caught at compile time, but is flagged at run time.

As an example of its use, the effect of the following method invocations is the same:

```
int a1 = [b multiply 6 by 9];

int a2 = [b perform selector (int multiply int by int)
            with (6, 9)];
```

The difference is that in the *perform with* case, the selector is an argument. It could just as well have been `selector (int` *divide* `int` *by* `int)`.

4.7.2 Postponed method invocation

The *perform with* method moves the boundary between code and data, by retrieving the selector of the message to be sent from data instead of the usual code. This boundary can be moved further. Obviously, moving the boundary defers to run time decisions that otherwise a compiler would make, thus negatively influencing execution time. Nevertheless, the flexibility thus gained is considerable.

An `Invocation` is an object that contains a description of a method invocation: the selector, arguments, and receiver. To continue the multiplication example from the previous section, the following is yet another way of asking an object `b` to *multiply* 6 *by* 9:

```
int a = [[[Invocation of selector (int multiply int by int)
                      to: b
                      with (6, 9)]
           result]
          components];
```

In this case, an `Invocation` object is constructed and asked for its *result*, which causes the invocation to *fire* first, actually performing the method invocation. The *result* method returns an `InvocationResult` object, from which the actual values that were returned from the invocation can be retrieved through the `dynamic`-typed *components* method.

An interesting feature of the `Invocation` objects is their ability to be curried, i.e., already carry the values for some of the arguments, but not all. The remaining arguments can be provided later, either explicitly or through a method invocation that provides the missing arguments and method name parts, as the following example shows.

Example

Consider the following multiply-accumulate method of some object:

```
int
  addAfter int d
   multiply int a
         by int c
{
  = a * c + d;
}
```

Furthermore, suppose that some object `b` implements this method, and that we have the following invocation `inv`:

```
Invocation inv
  = [Invocation for selector (int addAfter int
                                  multiply int by int)
                to: b with -12];
```

The invocation `inv` is not complete: the selector dictates 3 integer arguments, whereas only one is provided. Any attempt to *fire* the invocation or to ask for its *result* will fail. However, the invocation can be completed by sending it a message with the missing arguments and corresponding method name parts. To complete the example, the value of `a` after the following initialization

```
int a = [inv multiply 6 by 9];
```

is 42.
□

4.7.3 Forwarding

TOM binds all methods dynamically. In addition, it passes the selector of the message as the implicit argument `cmd` to every method invocation. This enables a mechanism to handle the case when an object receives a message it does not understand. This mechanism is called *forwarding*.

When a message is forwarded, the receiver is asked for an object to which to forward the message, using the following method that every object should inherit from the instance `All` :

```
All
  forwardDelegate selector sel
{
  = self
}
```

When the object returned by *forwardDelegate* is different from `self`, the message is forwarded to that object. Otherwise, the message is packed into an `Invocation` object and passed to the receiver using the following method, also inherited by every object from the instance `All`:

```
InvocationResult
  forwardInvocation Invocation invocation;
```

The default implementation of this method raises a condition.

The forwarding mechanism provides a solid basis for an implementation within the language of what Java calls Remote Method Invocation and what is known as Distributed Objects in Objective-C.

4.7.4 Introspection

TOM provides run-time introspection: every unit (available as an instance of the `Unit` class) can be queried for its classes (the class objects) and extensions (identified by instances of the `Extension` class). Each class can be asked for its extensions, and each extension (like each class) for its methods, variables, and superclasses. The value of an object variable can be read and set by name. Methods defined by a particular extension can be invoked.

Introspection is a powerful flexibility mechanism, as it allows code to query (and *reason about*) its building blocks.

4.8 Compile-time features

Not everything in a language designed for extensibility and flexibility of code is necessarily flexible or a run-time matter. Some compile-time features interact with extensibility, warranting discussion in this and the following sections.

An important compile-time feature is the possibility to give names to constants. This can be as elegant as a `const int` in C++ or as textual as a `#define` in C.

TOM offers the `const` declaration, which is allowed anywhere where a variable declaration is allowed, but which, given the nature of giving names to numbers, usually is employed at the class level. The value of a `const` is a constant expression, like for instance the definition of `TRUE` and `FALSE` in the instance `All`:

```
const TRUE = 0 == 0;
const FALSE = !TRUE;
```

Another compile-time issue is the typing of object variables. For a value of one of the basic types, the type is not only important for the operations that can be performed on that value, but also because it dictates the size of the memory location needed to store the value. For a variable that references an object, the storage size is independent of the type of object to which it points. A notion of the type is only important for deducing which message to send for a given method invocation.

All method invocations are dynamically bound and trigger the forwarding mechanism when a method with the given signature is not implemented by the receiver. As a result, strict typing at compile time is useless: many a program can be written of which the validness can not be confirmed by static type checking. The strict separation between the interface and implementation of objects, that is maintained at run time, will catch any errors.

This attitude towards typing explains why TOM allows a subclass or extension to redeclare what was defined in a superclass or extension. For example, when the following method exists:

```
// in some class:
All
  theObject;
```

an extension or subclass can redeclare it thus:

```
// in some subclass:
redeclare Any
  theObject;
```

Under the method overloading rules (section 4.5.3), these declarations concern the same method signature. Taking the meaning of the `All` and `Any` types into account (section 4.4.3), the redeclaration changes the meaning of the method from 'some object is returned about which nothing special is known' to 'any object is returned; if you know which one it is, you are probably right.'

Redeclaration is most often used for instance variables. For example, a generic `Parser` object has an instance variable to reference the generic `Lexer` object being used to lex the input. When the `Parser` is subclassed to obtain a `TOMParser`, that parser will probably employ a `TOMLexer`, a fact that can be expressed by a redeclaration.

4.8.1 Method-arguments default value

It is not uncommon for a method to have various parameters that influence its operation. The parameters can be the state of an object or simply arguments to the method. It is also not uncommon for some of the parameters to have the same value in most of the invocations. For example, a method to retrieve a numeric value from a `String` object may have arguments to indicate that the number may or may not be preceded by a sign or that C-style base-modifying prefixes (`0x` for hexadecimal and `0` for octal) should be considered. It will be tedious to have to specify those arguments for every invocation.

One way around the *boring arguments* is to provide methods which lack those arguments and which invoke the real method with the appropriate mix of default and actual arguments. However, when doing this exhaustively for every combination of default and actual arguments, the number of methods to be provided quickly explodes. This leads to arbitrary omissions, which is undesirable.

For these circumstances, TOM provides default argument values. For example, the `String` number-retrieval method could be declared thus (the colon in the method name parts with default argument values is conventional):

```
int
  integerValue (int, int) (start, length)
    allowSign: boolean signs = YES
 allowCBases: boolean bases = YES;
```

and the following invocations would be possible (specifying `-1` as the `length` indicates the remainder of the string):

```
a1 = [s integerValue (0, -1)];
a2 = [s integerValue (10, -1) allowCBases: NO];
a3 = [s integerValue (19, 67) allowSign: YES allowCBases: YES];
```

When a subclass overrides a method that provides default argument values, those defaults remain in effect if the subclass does not specify default values.

Discussion

The default argument values are used at compile-time, to be precise: while the invocations are compiled. At run time they are no longer visible. When compared with a mechanism like adding extra methods which substitute default values, it has the disadvantage that the default values can not be changed by an extension. However, such modifications are a bad idea: the programmer writing a method invocation knows the values of any unspecified arguments and decides not to specify them because their values suit. He would be surprised when those defaults change. And his code would start to fail.

The first argument to a method can not be optional. This prevents many ambiguities that otherwise would be possible and surprising.

4.8.2 Method conditions

TOM provides method preconditions and method postconditions, inspired by Eiffel [29]. A method precondition is a boolean expression that can be evaluated upon every entry to the method and which is supposed to never fail. If the condition evaluates to `FALSE`, the program is deemed incorrect. Similarly, a method postcondition is evaluated just before returning from the method.

As an example, the following declaration in an imaginary array class exemplifies a precondition that requires the `index` to be within bounds.

```
Any
  at int index
pre
  index >= 0 && index < [self length];
```

A precondition can be checked upon every entry to the method. When a precondition fails, the following call is executed:

```
[self preconditionFailed cmd];
```

All objects inherit this method from the instance `All`; the default implementation raises a condition, but this method can, of course, be overridden.

Method conditions also apply to methods that override the method containing the condition. Thus, if a subclass or extension overrides a method, the new method will also *inherit* the method conditions.

Method postconditions require two additional language mechanisms. One is the ability to refer by name to the value that is returned by the method. An example is the following method by the `All` instance, which tests the receiver being the `other` object (the *eq* method is the method equivalent of the equality operator), and of which the postcondition suggests that the method should not be overridden:

```
boolean (result)
  eq All other
post
  (self == other) == result
{
  = self == other;
}
```

The second language feature made necessary by postconditions is the ability to refer to a value that was computed upon entering the method. This is provided by the unary `old` operator. As an example, consider the following method from a (single threaded) semaphore unlock operation:

```
void
  enableGC
pre
  gc_inhibit > 0
post
  gc_inhibit == old gc_inhibit - 1
{
  gc_inhibit -= 1;
}
```

Method conditions increase the code size and, when checked, can cost considerable run time. A compiler is free to ignore method conditions and a human can instruct it to do so. Method conditions should not contain side effects.

Discussion

Method conditions in Eiffel are method-centric, as expressed by [29, page 341]: "If you promise to call [my method] `r` with [its] pre[condition] satisfied then I, in return, promise to deliver a final state in which [its] post[condition] is satisfied." Consider a method that overrides a method with precondition `pre` and postcondition `post`, and declares a precondition `pre_sub` and `post_sub`. From the method-centric point of view, polymorphism dictates that the overriding method can not demand more from client code but it is free to promise more. Put differently, the effective precondition will be the weaker `pre || sub_pre` and effective postcondition the stronger `post && sub_post`.

In TOM, method conditions are deemed to be applicable not just to the method but to the whole object. For example, a precondition must be able to express "before this method is invoked, the receiver must have been properly initialized," irrespective of any preconditions for the same method of the superclass. Therefore, the rule of widening preconditions and narrowing post-conditions is not enforced. Instead a precondition can narrow using `pre &&` and a postcondition can widen using `post ||`. An additional advantage is that inherited conditions can be fully overruled, thus aiding extensibility.

In addition to method conditions, Eiffel employs *class invariants*: boolean conditions that must hold upon entry to or exit from a method that is invoked as the result of a message-send to an object different from `self`. The availability of class invariants might explain the strict semantics of method conditions. However, since class invariants are inherited by subclasses, they reduce the freedom of subclasses. In addition, implementation of class invariants requires consideration of the class of either the sender of the message (in a method) or the receiver of the message (in a method invocation). This, too, hampers extensibility.

4.9 Missing features

TOM is a young language. As such it lacks some features. Some of these features are not yet defined but expected to be included in a future version to TOM.

One of those missing features is a means for indicating that a certain method, class, or variable is available for backwards compatibility only: TOM misses the `obsolete` keyword as provided by Eiffel.

A facility to give different names to existing types is needed, as in

```
typedef int pid_t;
```

A `typedef` would be allowed everywhere where a class, instance, or local variable declaration is allowed. The type name introduced by the `typedef` would have similar scoping rules. Note that the type names introduced by a `typedef` do not introduce new types, and hence would not affect method overloading.

Another interesting change would be to allow the last method name part to be argumentless irrespective of whether it is the single name part (already allowed) or not (not yet allowed). This is especially interesting for curried invocations, allowing them to be used for an argumentless message.

The feature missed most is probably the first to be added in the near future: blocks. A block—the name is from Smalltalk; blocks are available in many languages under many names—is a compound expression of which the evaluation can be postponed. A reference to it can be passed around, much like one can now manipulate `Invocation` objects. The advantage of blocks over `Invocation` objects is that an `Invocation` must invoke a method and needs a receiver of a certain class, whereas a block is itself the receiver, and the method is implicit.

4.10 Résumé

Extensibility in TOM is provided by the language. It is supported at compile time and link time by the development tools and at run time by the run-time environment. Additional flexibility is provided by the run-time environment, aided by the language.

The requirement of run-time flexibility of code has introduced flexibility in the organization of source code, at compile time. Since a class can be extended, its complete definition need not be contained in a single file.

Chapter 5

TOM: Implementation

This chapter describes an implementation of the TOM programming language, as developed by the author and designated *the TOM reference implementation*. In this chapter, the name TOM-1 refers to this implementation. It includes a compiler and a run-time library.

Obviously, an important goal of TOM-1 is to provide an implementation of the TOM programming language that can serve as a test bed for the extensibility of TOM. As a result, flexibility of code is more important for the design and implementation of TOM-1 than is the speed of compilation or execution speed of the resulting code.

5.1 Source boundary

In TOM-1, the level of source availability (see section 3.3 on page 17) has little influence. Present code (level 1) needs no explanation; with open source (level 2) access, every feature offered by the language is available.

Closed source access (level 3) to a unit means that the unit file (`.u`) is available, and that for every TOM source file (`.t`) in the unit, an interface file (`.j`) is available. The interface file usually contains everything the implementation file contained at the time the interface was created, except for method bodies and `extern` qualifications. Most notably, an interface file usually includes the definition of constants, method argument default values, and method pre- and postconditions. Closed source access to past code still implies that every feature offered by the language is available.

Binary access (level 4 source access) means that no sources are available. Luckily, some information can be retrieved from the binary, aided by run-time introspection, in an attempt to alleviate the absence of source. This is accomplished by retrieving the information that is already present in a running program because it is necessary for providing run-time flexibility.

It is possible to write a plug-in which is loaded into the running binary that has been produced by TOM-1, to retrieve the following information, for every desired unit:

- the units upon which this unit depends;
- the names of all classes and extensions introduced by this unit;
- for every extension or class the superclasses introduced by that extension or class;
- for every extension or class, the type and name of any object variables that it introduces, including, for each class variable, whether it is static, not static, or thread-local; and
- for every extension or class, the selector of each method that it defines.

This information, retrieved from a binary, strongly resembles the interface of the original closed source, i.e., the contents of all `.j` files of the desired unit. Most importantly, it contains enough information to allow the implementation of extensions and subclasses.

Unfortunately, the interface thus retrieved excludes information on method-arguments default values, any preconditions and postconditions that may have been defined, and any constants that were defined. Put differently, information about all compile-time features has been lost when retrieving information from a binary.

5.2 Extensibility

TOM-1 does not provide the *full* extensibility defined by the TOM language. These restrictions are imposed by the TOM-1 *implementation*; it is possible to work on TOM-1 so as to remove these restrictions. They are:

- The addition of state through dynamic loading, through a named extension or class posing, must not affect live objects, that have been allocated and not yet reclaimed by the garbage collector. Thus, if a class has live instances, addition of instance variables is not supported.
- The same restriction applies to a class, being the sole instance of its meta class. Unfortunately, class objects are preallocated by TOM-1 *at compile time*. As a result, the addition of non-static state is not allowed at link or run time. Fortunately, non-static class variables are not frequently used.

5.3 Compiler

The TOM-1 compiler, named TOMC, is a simple compiler that processes source files one at a time. It emits a C source file (`.c`) for every TOM source file (`.t`). This does not result in the fastest possible code but it keeps the portability of TOM-1 high: porting TOM-1 to a new platform typically takes a few hours, time which is mostly spent on the run-time environment and the *perform* methods.

The C files generated by TOMC must be compiled by the GNU C compiler [38], for they depend on GNU CC-specific extensions to the C language. This demand on the C compiler does not impose a severe restriction on the portability of TOM, since GNU CC is available on every significant platform.

This section discusses various design and implementation issues concerning the TOM-1 compiler and its support of the TOM programming language.

5.3.1 Code annotations

TOM-1 provides three kinds of source code annotations. Each kind serves its own particular purpose.

1. The *normal* comment is the C-style comment:

   ```
   /* This is a normal comment, C-style.  */
   ```

   This C-style comment is pure comment: an annotation of program code, to be read by the human reader. It does not have any meaning to any tool, most notably a compiler.

2. The *abnormal* comment is the C++-style comment:

   ```
   // This is an abnormal comment, C++-style.
   ```

   It is abnormal in the sense that the TOMC compiler will issue a warning for every occurrence of an abnormal comment. The intended use of abnormal comments is for comments which discuss or indicate problem situations: they can be considered an institutionalized version of the informal `???` and `XXX` strings in comments.

3. A generic annotation mechanism, using SGML-like tags [16], for example:

```
<foo> This is an SGML-style code annotation.  </foo>

<bar> This is one too.  </bar>
```

SGML-style annotations are intended to be significant to tools. Most notably, the TOMC compiler regards them as whitespace, except for `<c>`...`</c>` (see below). Annotations currently in use are:

`<copyright> ...</copyright>` Every TOM source file in the TOM-1 distribution starts with a copyright notice enclosed in these tags.

`<c> ...</c>` C glue code, see the next section (5.3.2).

`<0> ...</0>` Conventional TOM variant of the idiomatic C construction:

```
#if 0
...
#endif
```

which skips the '...'.

`<doc> ...</doc>` These documentation comments are extracted by the document extraction tool `tm` (see section 6.7). They are used to document classes, object variables, and methods.

5.3.2 Interfacing with C

For any programming language, it is important to be able to access functionality written in a different programming language. Such access is provided by glue code. The designated way to interface TOM code with code in another programming language, is to implement methods of TOM objects in the other language. For this purpose, the C language is an obvious choice.

A TOM method is flagged as having an implementation in another language by the `extern` qualifier:

```
extern double
  atan2 (double, double) (y, x);
```

Obviously, an `extern` method does not have a body in TOM code.

For methods employing the `dynamic` type, i.e., methods that accept or return a variable number of values of any type, an external implementation is mandatory, since the language does not yet define how to create or dissect `dynamic`-typed values. For example, the '`id` *print* `dynamic`' method, described on page 56, is implemented externally only for that reason.

Though external implementations are mandatory for methods employing the `dynamic` type, they otherwise have several disadvantages:

- For a method implementation in hand-written C, there is no compiler to generate code for inherited method conditions, and therefore no checking of such method conditions. This omission applies to all compiler features: method-arguments default values are also not available to method invocations that are written in C.

- The external method must be completely written in C; there is no mechanism to include TOM code.

- The notation of TOM method invocations in C can be pretty hefty. The C language distinguishes, in function invocations, arguments of which the formal type is defined from those with an undefined type. In the latter case, a mechanism known as type promotion converts the arguments before passing them to the called function. On 32-bit architectures, this mechanism mostly affects `float` arguments, which are promoted to `double`, being twice in size. Since TOM methods always have known argument types, such type changes cause problems. These can be remedied but require explicit typing of the method invocations. For example, to invoke an argumentless method, the C type of the method being invoked is:

  ```
  tom_object (*)(tom_object, selector)
  ```

  Even when macros are used to remedy this verbosity, this notation remains elaborate.

TOM-1 provides an alternative to external methods: it is possible to include literal C code in TOM code. Obviously, this facility depends on the TOM-1 compiler emitting C code, on the TOM-1 garbage collector being conservative, &c. However it makes writing glue code so much easier, that the advantages outweigh the disadvantages, for now.

As an example, the following class is glue code for the `atan2` function provided by the C math library.

```
<c>
#include <math.h>
</c>
```

```
implementation class Math

double (result)
  atan2 (double, double) (b, a)
{
  <c>
    result = atan2 (b, a);
  </c>
}

end;

implementation instance Math end;
```

Note how the value that is returned from the method is declared to have the name `result`, and that it can be assigned from C code.

With this definition, TOM code can invoke the `atan2` function of the C library, simply by invoking this *atan2* method.

5.4 Run-time environment

This section discusses some implementation issues concerning the run-time environment that are directly or indirectly relevant for extensibility or the implementation of the language.

5.4.1 Method binding

As explained in section 4.1.4, any association in code between an operation and an implementation, increases code fragility and negatively influences extensibility. Therefore, in TOM-1 all method invocations are dynamically bound.

Numerous approaches are available for the efficient implementation of dynamic method binding, see for example [5] and [6]. TOM-1 method binding uses a mechanism that is also used by GNU Objective-C , described in [42].

The single dispatch method binding problem can be described as a mapping

$$class \times selector \mapsto method \tag{5.1}$$

where the *class* is the class of the receiver of the message. (The *class* of a class object is the meta class.)

An efficient implementation of this mapping is made less than trivial because dynamic loading can introduce additional selectors, new or already known. As an example of a possible approach, NeXT's implementation of Objective-C [36] ensures that every selector is a unique string, aided by the linker and the dynamic linker. However, depending on linker features is not portable.

The solution used by TOM-1 accepts that every selector can have one or more *descriptors*. The selector administration, which starts at link time or run time, and which is continued at run time when plug-ins are loaded, associates with every selector a unique *selector identity*. The identity is stored in every descriptor for that selector. This way, every selector in a running program is identified by an integer number that is unique for the selector. These identities reside in a closed naming $[0..n)$.

When a program has n distinct selectors and loading a plug-in introduces m selector descriptors, then for the new n, dubbed n', the following equation holds:

$$n \leq n' \leq n + m \tag{5.2}$$

The integer selector names can be used as an index into a two-level array as depicted in figure 5.1. The first level index selects a *bucket*; at the second level, a entry in the bucket points at the method. This scheme allows buckets to be shared between a class and its subclasses, saving considerably in memory usage, though at the expense of an extra indirection for each lookup.

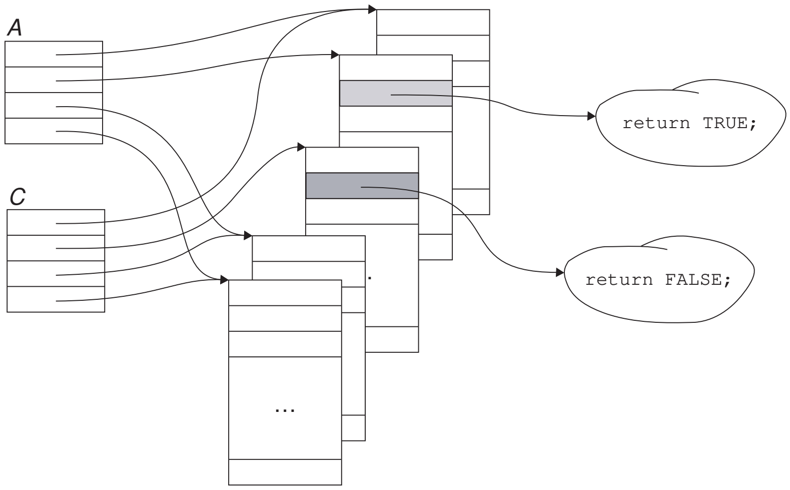

Figure 5.1: Method dispatch tables (of classes A and C) using a 2-level array.

Using this scheme, a method invocation is as expensive as a few indirections, which cache well for frequent use, and a computed function call:

```
tom_object receiver = ...;
tom_bucket b = receiver->isa->dispatch->buckets[sel / N];
void (*) () method = b->methods[sel % N];
```

with `N` being the size of the buckets. A smaller `N` increases the possibility for bucket sharing but also increases the overhead. In the current implementation, `N` is 16.

The run-time overhead of this method binding scheme depends on the kind of application and the way the overhead is measured. A rough estimate, valid for most applications, dictates that dynamic method binding of all method invocations induces approximately 10% CPU time overhead.

Concluding, the TOM-1 method binding mechanism is efficient, portable, and easy to use from hand-written C code.

5.4.2 State binding

The state carried by each object originates from its class definition, any extensions defined for it, and the state inherited from its superclasses. Multiple extensions of a class can define additional state of an object, and state can be inherited from multiple superclasses. As a result, object variables do not necessarily reside at a constant offset from the start of an object.

The offset from any object variable to another object variable defined in the same extension (main or named) is constant, irrespective of the class in which the variable resides. In addition, we have observed that for a reasonably-sized program, the number of state-defining extensions is approximately equal to the number of classes. This can be explained from the fact that classes tend to introduce, with respect to superclasses, additional instance variables; non-static class variables are not used; and extensions usually only define methods.

Given this information and the requirement of run-time extensibility, the following scheme is used to access object variables: every extension that introduces additional object variables is assigned a number, the *extension identity*. Every class has an extension offset table and when a method needs access to a particular object variable, it uses the table to retrieve the offset from `self` to the start of the first variable in the extension containing that object variable. Since all these ingredients are constant, this computation needs to be done once per method and extension being accessed.

The memory overhead of this scheme with respect to, for instance, the object layout generated by C++ compilers, is difficult to measure, accounting for only a few instructions and some local variables in methods accessing object variables. State binding is done on a per-method basis, incurring little overhead for individual accesses, for example within a loop. Furthermore, it does have the advantage of a constant, small, memory overhead per object. Compared to C++, the TOM-1 overhead is only larger than that of a C++ object without virtual member functions, without a virtual superclass carrying state, and without a repeatedly-inherited superclass carrying state.

5.4.3 Library options

Past code supports future code and present code depends on past code. All code depends on its clients to be properly initialized: in this respect, past code depends on future code. However, this dependency should not imply that every program must start with initialization of the libraries on which it depends. Such *musts* are to be avoided: the smallest valid program should be as empty as possible, even if many libraries are used.

Initialization needs parameterization. If past code is initialized automatically (i.e., beyond control of the future code), the initialization should still be parameterizable.

For instance, the C library is initialized automatically before the `main` function is invoked. This is obvious, since it is the C library that actually calls `main`. The initialization of the C library is implicit and not parameterized.

The X library, which provides the connectivity from a program to an X window server, is written in C. Every X program is required to invoke a particular function to let the X library initialize itself. As part of this initialization, the command line, passed as an argument to the invocation, is scrutinized and those options that the X library understands are removed. The initialization of the X library is explicit and parameterized by command line options.

In TOM-1, initialization of code other than the program code is implemented through *load* methods (see section 5.4.4). Parameterization thereof is handled by library options. Library options are command line arguments that start with a colon, '`:`', allowing them to be discerned from the normal options that, at least on UNIX, start with a '`-`'. The syntactic distinction enables unknown library options to be skipped by program code, allowing ':'-options to be passed to code that is dynamically loaded.

Example

The garbage collector in TOM-1 has several parameters which influence a program's memory usage and the efficiency thereof. These parameters can be given a value different from their default value using library options (section 5.4.5).

Some other interesting library options are:

`:cc-pre` Instruct method preconditions to be checked. The code to check method preconditions is usually included in the compiler output, but they are not actually checked, unless `:cc-pre` is provided.

`:cc-post` Like `:cc-pre`, but applying to the method postconditions.

`:extend=`*objectfile* Dynamically load the code in the file named *objectfile* before invoking the *main* method.

□

5.4.4 *load* methods

A *load* method is a special method: every *load* method in a class or extension, is invoked automatically when it is loaded. All *load* methods in a program and the libraries upon which it depends are invoked when the program is started, before the *main* method is invoked. The *load* methods allow initialization of code without needing to depend on future code. Furthermore, they are instrumental in the implementation of library options.

```
void
  load MutableArray args;
```

The arguments to the program are available as the `args`, ready to be modified: the *load* method can remove from the `args` array every argument that it recognizes and handles. The arguments that are removed are not passed to the *main* method.

The *load* methods are partially ordered: unordered within a unit, the *load* methods of a particular unit are invoked after the *load* methods in the units upon which it depends.

Example

The TAG *graphical user interface* (GUI) library provides a GUI API to programs which is independent of the underlying window system. A TAG program is developed and compiled for the TAG API.

At compile time, link time, or run time, one or more *concrete implementation of TAG* (CIT) units can be added to the program, about which neither TAG nor the program have prior knowledge. Each of these CITs employs a *load* method to register itself as a CIT with TAG.

Various CITs exist and multiple CITs coexist within a program. The user may indicate a preferred CIT through a library option. Of course, the CIT to be used thus can be dynamically loaded using `:extend`.

For example, the CIT to connect to an X server is named `x`; to open the X display `crypton:0`, the option `:display=x:crypton:0` can be used. As a

special hack, to soothe the unexpecting user, the *load* method of the `x` CIT recognizes and handles the customary `-display` command line option, as in `-display crypton:0`, and the `DISPLAY` environment variable. Which only shows the flexibility offered by the *load* methods.
□

5.4.5 Garbage collection

Automated garbage collection is important to have in an object oriented environment. The opposite technique, manual storage management, hampers encapsulation: having to know when to free the memory that is occupied by an object is having to know too much about the object and its clients. With the extensibility of TOM, manual storage management becomes even less of an option. TOM-1 employs a time-constrained incremental write-barrier mark & sweep garbage collector (GC) based on the concurrent GC described in [11].

Various library options are available to control some parameters that affect the GC operation. `:gc-pth` sets the object allocation threshold which triggers a run of the garbage collector for `:gc-ptl` milliseconds, `:gc-stat` requests a memory status report just before the program exits, &c.

5.4.6 Debugging support

While debugging, it can be desirable to set a breakpoint on messaging a particular object, messaging with a particular selector, or messaging a particular object with a particular selector. The latter is more or less easy if you know the method that will be invoked as a result. However, the first two are not easy to set since they involve a possibly very large number of methods.

All method dispatches in TOM-1 are guided through a single lookup function, which accepts an object and a selector and returns the method to be invoked. It is of course possible to set a breakpoint on this function and make it conditional on the selector or the receiver. However, contemporary debuggers require a context switch to the debugger to evaluate the condition, thus reducing the execution speed by several orders or magnitude. The same problem occurs in the case when both the receiver and selector are known and a breakpoint is set on a particular method, but the first few thousand occurrences of the breakpoint must be skipped: each occurrence triggers a switch to the debugger, greatly influencing execution speed.

In these circumstances avoiding the context switches can turn this major debugging problem into a piece of debugging cake. TOM-1 actively supports this: condition evaluation—*does the receiver match? does the selector match?*—can be performed by the run-time library, incurring very small overhead on every method dispatch when compared with a normal run without a debugger. On a match, the library invokes a particular function, and the debugger needs only to set a breakpoint on that function.

In the current implementation, the method lookup function performs the check for matching selectors or receivers and, before that, whether the check is necessary. The overhead is one test of a global variable per method lookup, incurred in every run of every program. On the other hand, there are circumstances where it is instrumental in making a problem debuggable.

5.5 Availability

Just like freedom is important in the design philosophy of TOM, freedom is important for TOM-1, the implementation of TOM discussed in this chapter.

A programming language has reached an important goal when it is being used as such. For example, a compiler and environment for the Self programming language [43] has long been available for free, but with only the Sun SPARC as a target [35] and without availability of sources, implying that everybody who did not have access to such a machine was not in a position to use Self.

As another example, Eiffel has long been a language for which a compiler or environment was not freely available. This situation was only recently remedied by the GNU SmallEiffel compiler [9]. Even Bertrand Meyer, the man behind Eiffel, observes that "Sather[...] has the benefit of a public-domain implementation" [29, §35.6] (Sather is a language that started as an Eiffel derivative).

When using TOM-1, the programmers of present code can not annotate assumptions in their code that may affect the programmers of future code negatively; past code is amendable by future code. This open attitude is reflected in the licensing of TOM-1. TOM-1 is available under the least restrictive open source license available: the TOM compiler and tools are distributed under the terms of the GNU General Public License [12]; the libraries are distributed under the GNU Library General Public License [13].

This means that programs developed using TOM-1 can be used for commercial applications, and that any bug fixed in the TOM-1 tools, or changes made to the libraries, should be fed back to the official distribution. However, the extensibility of TOM code makes the developer almost independent of what bugs or limitations the libraries contain: he can fix them in his own programs!

Chapter 6

Reflection

The first chapters of this dissertation argue that flexibility through extensibility is an answer to the problem of limited reuse in object-oriented programming languages. Later chapters present the TOM programming language that has been developed to implement that answer. In this chapter we will survey the benefits offered by the flexibility of code as provided by TOM.

6.1 Conditional extensibility

TOM provides extensibility at compile time, link time, and run time. The run-time extensibility is unique for TOM, being the reason for its development.

Extensibility at compile and link time, i.e., during the development of a project, can be regarded as being largely a feature of the development tools, i.e., the compiler and linker. As such it is not restricted to a particular language, as is indeed shown by the IBM VisualAge C++ compiler [22] providing C++ with the extensibility of subject oriented programming [19]. This line of reasoning can also be reversed: if compile-time and link-time flexibility can be added to a rigid language like C++, then run-time flexibility can be removed from code written in a flexible language like TOM.

Removing flexibility from code increases its execution speed and can result in a reduction of its size. These features can be important for, for instance, embedded systems, where processors must be cheap, hence slow, and the ROM small, hence full. Systems with such strict constraints usually provide no means or incentive for exploiting the run time flexibility, and any flexibility will be viewed as unnecessary and an undesirable overhead.

An example of removing flexibility is the removal, where possible, of dynamically bound method invocations, i.e., replacing dynamic binding by static binding. One step further, a small statically bound method can be *inlined* to save to overhead of the call.

Developing a complex application always involves decisions on the right amount of flexibility. Popular programming languages like C++ do not exhibit much flexibility and adding the proper amount of flexibility to a program is a difficult job. On the other hand, an important advantage of flexibility removal is that it can be automated: one can write a compiler that removes flexibility from a program, as shown by whole-program compilers such as [6]. Comparing the two approaches, the choice is between repeated *ad hoc* manual addition of flexibility to every program and automated removal of the generically available flexibility. TOM enables the latter.

6.1.1 Extensibility time

Removing run-time extensibility is a deployment decision.

The creator of a program knows how the program will be deployed and he is therefore the only person eligible to remove extensibility from the code. Library developers, on the other hand, do not have such knowledge and, consequently, binary distributed libraries must be fully flexible. Unless the linker, which is responsible for combining object code from libraries and the program, performs link-time optimizations, removing flexibility is a compile-time operation. A compiler operates on source code, and the more source code is available, the better the result can be. The best result is therefore obtained by a whole-program compiler.

When considering to assign extensibility removal to the linker—for instance in an attempt to enable binary distribution of libraries—several disadvantages show. Most importantly, the linker is a tool that is platform specific and not language specific. When a language requires specific linker functionality, the portability of the language is restricted. In addition, even the more obvious, non-language specific, linker optimizations such as function inlining and function call interface specialization are CPU specific and would require considerable effort for each additional CPU that is to be supported. The economics of linker modification are even worse for dynamic, run-time, linkers.

Concluding, developing a language to require extra functionality at link time signifies an unfortunate design decision. The reduction of code flexibility is best done at compile time. Consequently, flexibility of code is best manipulated by a whole-program compiler.

6.1.2 Extensibility scope

Extensibility removal is not a binary decision.

Flexibility need not be a *yes or no* question. Full flexibility can be maintained in some parts of the code, while it is removed from other parts. The flexible parts can, for instance, be identified by classes that need to remain fully extensible. The flexibility demand then applies to those classes and their subclasses.

Classes that are usefully kept flexible are situated along boundaries between sub-systems in a complete system, e.g., between a program and its input/output functionality, a program and its plug-ins, or a kernel and its device drivers.

6.1.3 Extensibility range

Non-extensibility during deployment does not preclude extensibility during development.

Extensibility is an aid at development time: extensible code is less fragile than non-extensible code. As a result, small changes in the source code induce little changes in the object code and recompilation can be fast. On the other hand, when a small source-code change *can* induce many object-code changes, one may express to the build tool the dependency of those object files on that source file or header file. With such dependencies, many a small change will cause recompilation of many source files. Without such dependencies it is possible that a recompile necessary after a change is erroneously omitted, causing bugs that are hard to find.

Of course, extensibility at development time is more interesting for the test and debug opportunities that it provides, as will be discussed in the next section. On the other hand, in a large project involving many programmers and a source code management system, the advantage of not needing to acquire a write lock on a particular source file, just to experiment with a slight modification to a class contained therein, should not be dismissed.

6.2 Varying software versions

This section and the sections to follow discuss software testing and testing possibilities offered by the code being extensible. This section describes differences between the released version of a program and its test versions, and why test versions are important. The next section introduces an example software project (*the elevator*) and uses it to explain what software testing entails. The two sections following discuss the possibilities.

6.2.1 Software test versions

It is common for the source code of a program to contain code that is only conditionally compiled. As a result, when compiled for the purpose of debugging or testing, the program will be different from the released version of the program.

As an example, in a program that is written in C, conditions can be asserted at run time using the `assert` macro from the include file `<assert.h>`, for example:

```
assert (1 + 1 == 2);
```

will result in code equivalent to something like:

```
do
  {
    if (!(1 + 1 == 2))
      {
        fprintf (stderr, "assertion failed:  1 + 1 == 2\n");
        abort ();
      }
  } while (0);
```

Using assertions, the programmer states his assumptions governing the code, even the most trivial ones (though usually less trivial than in the example). The idea is that when some bug is introduced during development, some assertion will fail and reveal the bug early. The assertions in a program thus aid in debugging and testing. Part of a *test plan* can be, for instance, that no assertion shall fail during any test.

Obviously, when time has come to compile the released code, those assertions are no longer necessary. This not only saves space; it is easier for a customer to accept a program occasionally crashing, than it is to explain to the customer that letting the program provide incomprehensible explanations of a problem, like

```
Assertion failed:  1 + 1 == 2
Bus error, core dumped
```

is easier than it is to prevent the problem in the first place.

When compiling for actual release, all code of our example C program is recompiled with the `NDEBUG` macro defined. As a result, the code equivalent of the `assert` example will be

```
;
```

which is a null operation. It is normal for a released program executable to contain few or no sanity checks. This is positive, since not checking saves execution time and memory space.

6.2.2 Regression testing

An important aspect in software testing is repeatability: a bug can only be usefully examined if the circumstances leading up to the exposure of the bug can be reproduced. When a bug is repeatable, a new test case X can be developed for it, and part of the specification for the next release of the software will be 'test case X must pass.' Until the bug is fixed, the software will fail this specification.

It is widely known that software is fragile: fixing a bug can introduce new bugs, not infrequently even bugs that were previously fixed. It is therefore almost mandatory to not only test that our new program passes test case X, but also test cases $1, 2, \ldots, X-1$. Only if none of the bugs that were fixed reoccur, we can be sure that the quality of the program did not regress. The collection of test cases for previous bugs is called the *regression test suite*, which is used during *regression testing* [3].

6.2.3 The economy of building test cases

Suppose we are working on Project L, which spans many released versions during many years. At some time, many years into the project, one of the customers reports a grave bug that we must be certain of avoiding in the future. By the test plan of Project L, every valid bug that is reported must be covered by a test case in the project's regression test suite, so we try to understand the problem and reproduce it on the current version. Unfortunately, we fail to reproduce it, so we try it on a freshly-built test version of the program. We fail to reproduce on the test version too, but not being able to trigger the bug does not help us, since according to the test plan, we need a test case. Trying to trigger the bug at the customer's site, using the customer's version of the program, can be rather impractical. We end up needing a test version of the binary running at the customer's site.

Obtaining a test version is easy if a project releases to the public about once a year or even less frequently, since in that case we need to keep only a limited number of test versions stand-by to address this problem. Unfortunately, Project L has neither millions of anonymous users nor exactly one user. Like many software projects, there are a few customers, each of whom receives some version of the software that is tailored to his situation and which is updated on demand. In such cases, retrieving the right version of the sources, to compile and build a test version, can be troublesome and time consuming. The situation is aggravated if the customer can include third party software into our product. Recompiling a test version of the whole system as it is running at the customer's site has become next to impossible.

6.2.4 The economy of building test versions

Apparently, the economy of building a test case for a bug that a customer reported depends on the economy of building a test version of the program allowing us to scrutinize the problem. When code is extensible, we can design the test version of the program to consist of the released version of the program with a *test extension.* When we need to have a test version of the program that runs at the customer's site, we can simply retrieve the customer's binary and run it with the test extension. When the interface between the program and test extension does not change often, we can use the most recent version of the applicable test extension.

It is of course possible to develop or use multiple test extensions for a program, for instance one for each distinct subsystem. A test extension requires extensibility of only certain classes. The flexibility in other classes may be removed, but the less flexibility is available throughout a program, the less possibility you have of changing your mind in the future, about what needs to be extensible or not.

6.3 A testing example

As an example, we have implemented software to control a system of elevators. As shown in figure 6.1, the system consists of a number of elevators that service a number of floors. On each floor, a prospective passenger can issue a request to travel up or down, and within each elevator carriage, he can request to visit any floor. Furthermore, each floor has three position switches for each elevator: one to indicate that the carriage is near and above the floor; a similar switch below the floor, and one exactly on it. When a swiftly moving elevator carriage hits a position switch near a floor, there still is enough time to slow down and stop at that floor.

In the stylized world of our example, our expensive hardware is connected to a single network through which all communication takes place. When for example the state of a switch changes, the switch posts a message on the network, and when the motor that drives an elevator needs to start, it receives a message from the network. As depicted in figure 6.2, in the software that controls the elevator system, *device drivers* abstract the control application from the actual hardware.

Testing a software system is a process of different stages (see [3] for a thorough introduction to software testing):

unit testing This is the first step in testing. Unit testing tests the functionality of the building blocks of the system. Good examples of such building blocks are single classes or other units of code developed by a

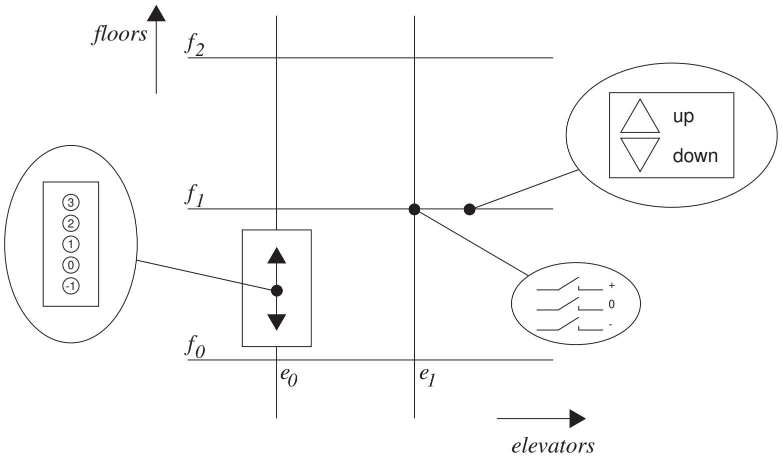

Figure 6.1: An elevator system and its various inputs.

single developer. Strategies for unit testing try to ensure that all code in the unit makes more-or-less sense, for example by trying to execute all statements at least once; trying all conditions both ways; or trying all paths through the code.

As an analogy, if we were directing a play, unit testing would involve each actor practicing his lines in front of a mirror.

integration testing The next step in testing, integration testing tests the various building blocks to interact as expected and specified.

For the play, we practice the scenes, making sure each actor does what he is supposed to do with the other actors present and active (enter left; dialogue; exit right).

system-level testing System-level testing determines whether the system actually provides the functionality it is to provide. Various strategies exist for devising system-level tests. To name one important example, *transaction testing* tests whether transactions are properly handled by the system. A transaction can be anything that enters the system, is processed, and possibly leaves. In our elevator, a passenger trip can be viewed as a transaction. The passenger enters the system with the first request ('I want to go up') and exits the system when the doors open at the destination floor (and the passenger leaves the carriage).

In our play, we would be practicing the whole play or, in a sub-system test, a complete act, to see whether everything feels good to the actors and director, and whether the message is clear to the public.

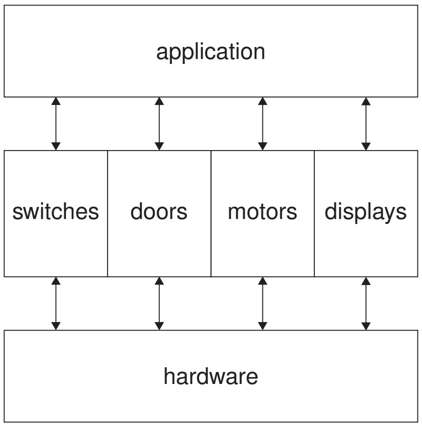

Figure 6.2: Application communicates with hardware through device drivers.

When testing an elevator at the system level, we test whether the software behaves correctly *in all circumstances*, or in a suitably large number of circumstances for us to feel confident. In this case, *correctly* means *according to specification*. It is not difficult to specify the behaviour of the example elevator system: transport people safely and efficiently. The software controls the transport; the software behaves correctly if it ensures that the specification is not violated, i.e., when its actions in response to its input stimuli are correct.

The problem with testing elevator software is that it takes humans, time, and an elevator. Regression test runs recur frequently, which makes the involvement of humans undesirable and long test runs expensive; the involvement of humans implies that repeatability is impossible; &c. A solution that is often opted for is *record and play back* (depicted in figure 6.3): record the stimuli of a test once (6.3a) and play them back as often as desired later (6.3b). The advantages are obvious.

Though used extensively, record-and-playback approaches to software testing have several disadvantages. Two of these are significant for our goals (for more information, see [3]):

1. The software must be *prepared* for running with record-and-playback. The testing is part of the design and the testing functionality is part of the source. This means that, e.g., third-party libraries, or other code that was not part of the design, can not be tested with the same approach.

2. A session must have been *recorded* before it can be *played back*. When hardware and software are developed concurrently, this implies that system-level or subsystem-level testing of the software can not start until hardware is available.

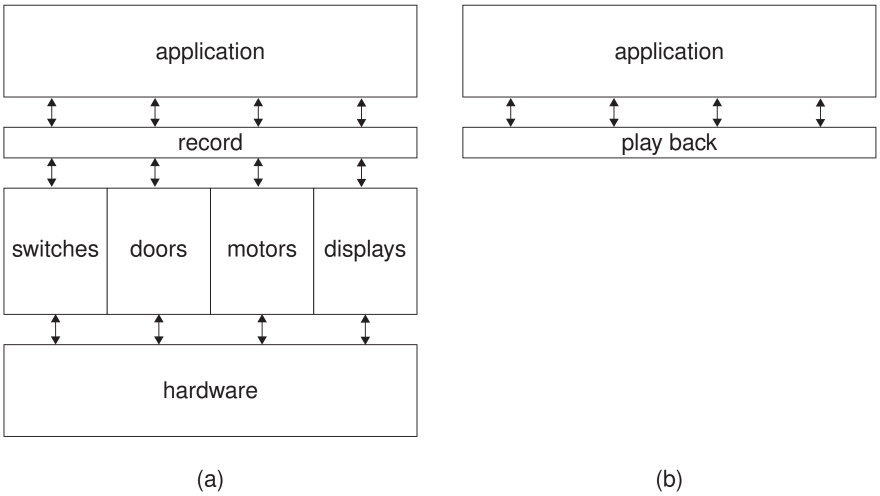

Figure 6.3: Events recorded once (a) can be played back later repeatedly (b).

Though record and play back is the only example of system-level testing that is presented here, the problems that are presented with it are universal for system-level testing. The next section shows how extensible code enables unplanned preparation of software for the purpose of testing.

6.4 Unplanned testing

The previous section discusses how the elevator software can be tested when the testing concept of the application was part of its design. The true strength of code extensibility is that even if testing was not planned, a test extension can still be developed [32]. This is important when employing a library that is provided by a third party, with different goals, a different design, and an unknown test plan. With extensible code, it is possible to develop and apply a test extension independently afterwards.

This section describes an example of the development of a test extension for the elevator from section 6.3. In fact, when we developed the elevator software, the development of the test extension was unplanned, on purpose. Put differently: in the design of the elevator software, we did not anticipate the development of a test extension. Of course we had the advantage that we could look at and learn from the full source of the elevator while developing the extension. Note however, that not having the sources is not a problem: in a language that provides run-time extensibility and flexibility, all important information, such as class names and method signatures, is available at run time and does not necessarily need to be retrieved from the source. In the development of a test extension, aided by a debugger and the extensibility, that information is easily retrieved.

The remainder of this section describes an approach for developing a test extension, focused on particular elements of the system: the switches. The approach applies equally well to the other elements of the system; the focus keeps the example clear.

In the software that controls the system, every tangible entity in the elevator system is proxied by an object. An event in the hardware is sent as a message over the network and delivered to the corresponding proxy object. For example, for each switch, there is an instance of a `Switch` class. When a switch changes state, ultimately the following method of the corresponding `Switch` is invoked:

```
void
  switch boolean new_state;
```

When considering only the switches, while testing the system with a record-and-playback approach, a play back involves invoking the *switch* method at appropriate times, as previously recorded. Underlying the testing process is the assumption that the software will, in the real world, respond correctly to the switches, when it, during testing, responds correctly to invocations of the *switch* method. We can abstract from the detail that a play back must be preceded by a record, and pose that, as far as the switches are concerned, we can test the system by appropriate invocations of the *switch* method. This can, of course, be abstracted to any method of any object.

The basic idea is that for the purpose of testing the software, we only need to invoke the right methods at the right time (and observe the outcome of the tests as usual). Which methods are right depends on the system, the part of the system that is tested, and the desired or required accuracy of the tests. Which time is right depends on the system, the part of the system that is tested, and the desired or necessary accuracy of the tests. This concept enables a certain approach to testing; it is not a recipe to all-encompassing success.

As an example: to test a calculator, the speed at which the, real or testing, user types is largely irrelevant until we start testing the calculator's responsiveness and speed. It is possible to use the same testing approach for the speed tests and functional tests, but if that approach is too slow, functional testing can be done in a way that abstracts from time, for example by not taking the time waiting for user input into account or by running on a faster machine. The next section is a concrete example of this approach.

Elevator test extension

To continue the elevator example, the software responds correctly to the switches by correctly responding to the invocations of the *switch* methods. This is true for all methods along the boundaries of what *appear* to be the device drivers—since our testing is unplanned we have no knowledge about the design and we do not actually know the device drivers in the code.

When the system was written in a language like TOM, the methods along the interface to the device drivers can be replaced by a test extension, as is depicted in figure 6.4. When the test extension is loaded, it replaces the device drivers with *something* that does not depend on the hardware. What exactly the test extension provides depends on the level of detail that is desired in the tests, or that is needed by the software under test.

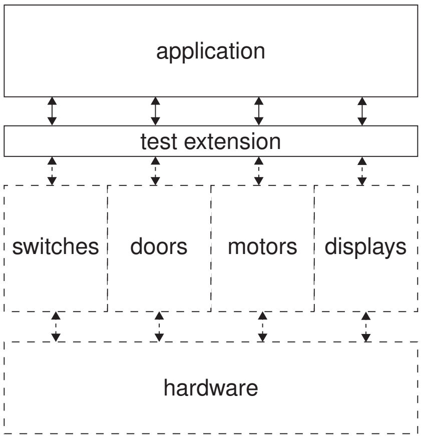

Figure 6.4: The test extension obviates the device drivers.

As an example of the abstraction from time, useful system-level tests can be performed by feeding the system a trace of input events and checking the output events. Time-independent trace tests neglect the issues that real time has on the system. This can be seen as a disadvantage, as bugs can be time dependent, but also as an advantage: all bugs that are not time dependent can be tested without the *burden* of real time. Being time independent, the software can run on the fastest machine available, further speeding up the tests. If all sources are available, it could even be recompiled to run on even faster machines.

Experiment

We have implemented a test extension for the elevator control software to perform transaction testing on the elevator scheduler. The extension removes any dependencies of the elevator software on the elevator hardware, as depicted in figure 6.4 [33]. The extension as implemented takes over the device drivers and situates them in a simulated world that is inhabited by simulated passengers. As implemented, the world is made up of the following players:

elevators All elevators are identical, they travel at constant speed, and exhibit infinite acceleration and deceleration. They can carry any number of passengers and take constant time to load and unload.

floors All floors are identical. They are equally spaced and each floor can be visited by all elevators.

passengers All passengers are equal. They have a 50% chance of wanting to travel to floor 0 when not already on that floor. Otherwise, they have a 50% chance of wanting to travel a distance that is larger than a quarter of the number of floors. When arriving at a floor, a passenger will wait for a random time, uniformly distributed between 5 and 100 seconds. With this behaviour, a passenger in the simulated world is equivalent to multiple persons in the real world.

switches The guidance switches at every floor in every elevator shaft are also present in the simulated world.

The elevator guidance switches are small, though they are large enough for the floor switch to be turned *on* by the elevator and stay *on* while the elevator stops at the floor.

motor Every elevator is powered by a motor. In the test extension, the motor device driver actually maintains the simulation of the moving elevator cage. When instructed to start moving, it repeatedly generates events to inform the guidance switches of being manipulated.

The simulated world is parameterized by the number of elevators, floors, and passengers. Furthermore, the physical dimensions and constraints of the system are defined at compile time—of the test extension. They are listed in table 6.1.

Table 6.1: Simulated dimensions.

description	value	unit
distance between floors	3	m
guidance switch distance from floor	0.25	m
switch length	0.01	m
elevator speed	1.2	m/s

The non-determinism inherent to a real-time system like an elevator system implies that a string of events can not be replayed to show exactly the same outcome. This is true in the real world and it simplifies the simulated world. As a result, generating test cases for the purpose of replay is not a possible testing approach. On the other hand, not being able to simulate and re-simulate events precisely introduces much freedom with respect to time: the freedom to run on a faster computer and the freedom to otherwise influence time.

The testing approach provided by the elevator test extension is that of generating random scenarios. While playing these scenarios, the system is checked to not violate safety, deadlock, liveness, and other rules. This actually imposes a strong demand on the simulation, in the sense that enough *interesting cases* must occur often enough during tests, in an attempt to avoid testing only trivial cases.

In our simple simulated world, we can not influence the occurrence of interesting events; we have not even defined what makes events interesting. Instead, we choose to test such a large number of cases that the interesting ones will probably occur too.

Running a large number of test cases requires many transactions to be processed by the system, i.e., many passenger travels. Unfortunately, testing many transactions implies a long testing time. Fortunately, we are in a position that the exact notion of time does not matter, and we can run the tests on faster machines.

The next step in maximizing the number of transactions that can be processed in a test run, stems from the fact that an elevator control system is idle most of the time. It needs only to respond to events, and when an elevator is not moving, nothing interesting will happen to it until it starts moving. Moreover, in our simple simulated world, nothing can happen to an elevator between hitting guidance switches. In test runs, such periods of time during which nothing can happen, can be skipped.

Decoupling simulated time from the real time has a dramatic effect on the run time of experiments. For example, with 20 persons inhabiting a simulated world that contains a single elevator to service 20 floors, a day in the experiment has actually elapsed in 33 seconds outside the experiment. Put differently: a day with $12.5 \cdot 10^3$ passenger travels is simulated in little over half a minute. More figures, with a varying number of elevators and passengers, and a constant 20 floors, are shown in figure 6.5.

As figure 6.5 shows, the run time of an experiment depends on the number of elevators, much more than on the number of persons. This stems from the fact that in this event driven system, handling the events is what costs time, and most events are generated by the elevator guidance switches. The persons

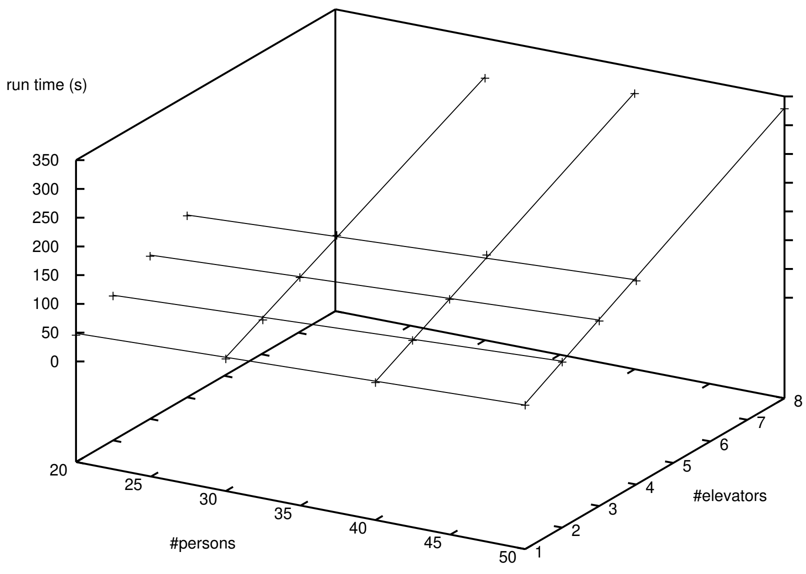

Figure 6.5: Run time of an 86400-second rush-hour experiment.

only make the elevators move and more persons means more passengers to share an elevator (and more prospective passengers to wait for an elevator).

Figure 6.6 shows the number of times a passenger successfully traveled. For every 10 extra persons in the simulated world, the number of passenger travels increases approximately 25%, whereas it only increases about 10% for every extra elevator. This can be explained by the abstractions of the testing model. It has been optimized for simplicity while still exercising that part of the software that governs the system-level transactions, the elevator scheduler. This has resulted in a model with the infinitely large elevator cages, constant 1-second load/unload time, and the fact that, even if the number of elevators increases and every elevator is so busy that it will stop at every floor, most of the time the elevator will be moving at 1.2m/s between the floors that are 3m apart.

The model offered by the test extension is of course tailored for the testing goal. It is possible to increase the accuracy of the model, increasing the run time but also adding possibilities for more checks, for instance, safety checks concerning the elevator doors. The model that we employed suffices for testing the elevator scheduler. In fact, by using the test extension, we found a bug in the scheduler. Testing many transactions appeared to be important, since the bug showed up only after $3 \cdot 10^4$ seconds of simulated time and more than 10^4 transactions.

The discovered bug appeared as a passenger having to wait half an hour for an elevator. The bug was that the elevator had visited the floor while the

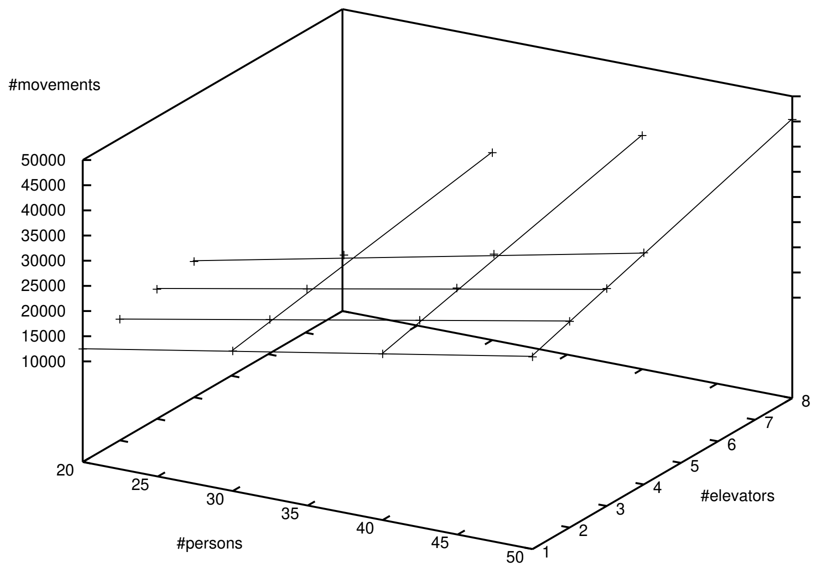

Figure 6.6: Passenger movements in an 86400-second rush-hour experiment.

request was outstanding, only to unload a passenger and continue travel in the other direction than the direction of the request. The scheduler erroneously marked the request as having been serviced.

After extensive testing, all passenger waiting and travel times can be observed to be *normal*, in that none is significantly longer than the others. We are now confident that the scheduler is functioning correctly.

Library extension

In the preceding sections, the separation of simulated time and real time is taken for granted. In the context of an event-driven simulated world, such a feature is indeed a non-issue. However, in this case, the simulation is maintained by a plug-in that is loaded into a program for the control of a real-world elevator. Neither the program has been prepared for this extension, nor the libraries used.

So, how can the simulation actually work? The answer will not be a surprise: the test extension simply extends, apart from the device drivers in the program, also the libraries, in two places:

1. The `Date` class in the `tom` library unit provides a notion of time as `double` values with respect to some beginning of time. This *epoch* is usually the start of the program, enabling the program to maintain the time close to its own starting time with very high accuracy, and

still discern milliseconds centuries from now. Since all time stamps are relative to the epoch, increasing the epoch advances time. The extension to the `tom` unit adds a method to the `Date` class to allow modification of the epoch.

2. The `RunLoop` class from the `too` library unit is used by the elevator software as the event dispatcher; it provides events from file descriptors and timers that can be set. The extension to the `too` unit replaces the method of `RunLoop` that implements the central dispatch loop. The modified method can be requested to not wait idly, advancing time every time it would otherwise wait, thus firing timers continuously.

6.5 Hardware/software co-development

A major problem in the development of hardware/software systems is that the hardware is often finished earlier than the software. It is possible to remedy this problem if it is possible to start earlier with development of the software. The testing approach presented in the previous section, which enables software testing without the hardware, can obviously also be used as a development approach, enabling software development without availability of the hardware. Test extensions should be developed in conjunction with the software, as early availability of test extensions enables early testing of the software system.

A common setup for hardware/software projects reflects the duality of the system in the organization: the hardware group develops the hardware and the software group develops the software. These groups are destined to not understand each other. In the proposed development approach, a third group is required, to develop the project's device drivers and testing extensions. The members of this group will need to understand both the hardware and software disciplines and can act as mediators between the software engineers and hardware engineers, understanding the needs of each group both as consumers and producers.

6.6 Hardware/software co-design

A popular trend in the research into hardware/software co-design is to hunt for the best language in which to describe complete systems. These languages are always programming languages, since a program can be executed to provide a simulation of the system being described and many programmers are available for the widely known languages.

Currently, Java is popular because it is machine independent and because the language provides multi-threading constructs (e.g., [20]). However, language dependence on certain run-time facilities, such as the dependence of the Java language on multi-threading, does not make the language more suitable than languages which do not depend on such facilities. In both cases, the run-time libraries provide support for those facilities.

It can be argued that TOM is the better system specification language, since it is not biased towards execution on any specific machine. Even Java has a bias: its much-touted machine independence is nothing more than a dependence on the Java Virtual Machine (JVM) [27]. On the other hand, any language can be used as a specification language: even a suitably extended subset of C is usable as a hardware description language [26]. In essence, the success of a system specification language depends on the semantics that the tools give to utterings in the language, and the availability of tools that allow sensible application.

6.7 Applications

TOM is a programming language. To research the real applicability and usefulness of the possibilities that it offers for software testing, many more experiments must be performed than the single elevator experiment described in section 6.4. The experiments should be performed by many different persons in many different projects. Before these people—you, reader, could be one of them—can be convinced to perform such an experiment, they must be convinced of the positive outcome of the project, even if the particular testing experiments would not meet high expectations.

For people to undertake the venture of starting to use a new programming language, they must be convinced that it will be worth it. The alternative, to invent something that attracts wide attention and to which the language is secondary, is not easily done. Examples of such successes are UNIX introducing C and executable content in web pages which was enabled by including Java in the web browser.

In addition to the elevator testing experiment, the following projects, great and small, were implemented in TOM. Some are real applications in real use. Some have been implemented up to proof of concept.

Libraries

mu The *meta unit* consists of parsers for TOM source and unit files and a collection of skeletal classes to represent TOM language constructs. The parsers are generated by `gp` (see below).

TAG The *TOM Abstract GUI* library unit provides a NeXTSTEP-like API featuring a unified imaging model, without relying on PostScript like NeXTSTEP does [1]. The flexibility of the TAG architecture is briefly described on page 80. Concrete implementations of this abstract GUI exist for X11 displays and PostScript printers.

tdbc The *TOM Data-Base Connectivity* library unit provides a simple and quite popular tabular database abstraction that is also available for Python (see `http://python.org`), Java (`http://java.sun.com`), &c. Currently `tdbc` has only one back-end, supporting the PostgreSQL database engine (see `http://www.postgresql.org`).

tomgtk A glue code library to enable the development of Gtk+ and GNOME programs in TOM (see `http://gtk.org` and `http://gnome.org`). The glue code takes care of various issues, making automatic and implicit in TOM code what is manually explicit in, e.g., C code. The two most important of these issues are storage management and run-time type checking.

Programs

gp *Generates parsers*: for a given grammar, `gp` emits a class that implements a recursive descent parser. Supporting the `gp` parsers is the `gps` library unit which, among other things, defines a skeletal parser and lexers.

tm *Meta TOM* is the TOM documentation extractor: it generates class documentation from TOM source files, aided by `<doc>` ... `</doc>` comments that a friendly programmer adds to every method, instance variable, and class. `tm` employs `mu`.

tesla A TOM compiler under development, `tesla` is the first TOM compiler to be written in TOM. Designed to be a whole-program compiler, it currently provides the same functionality as the TOM-1 compiler: everything is fully flexible and extensible. Flexibility removal is the next big thing to be added. `tesla` employs the same parsers from `mu` as used by `tm`.

An interesting implementation aspect of `tesla` is its use of a class hierarchy to represent host and target platforms. Only the platform-describing classes that are relevant for the compiler to be built are actually compiled into the program. (They could be dynamically loaded.)

bit *Builds interfaces for TAG*, it resembles the NeXTSTEP *Interface Builder*. `bit` is a tool to define the user interface of programs that employ TAG.

`bit` can be in one of two modes: either the user interface is being built by creating and manipulating user-interface elements, or the interface is being tested with the user-interface elements behaving as they normally do. These modes are added by `bit` to all TAG user-interface elements through the extension of two TAG classes. The TAG library does not contain provisions for this dual behaviour exposed by `bit`.

Illustrate It! *It!* is a vector-based single-image drawing application in the spirit of Adobe Illustrator and Altsys Virtuoso. *It* uses TAG and employs plug-ins to provide its functionality. All tools are provided by plug-ins, even the grid is a plug-in. The program itself is not much more than a framework for single-page documents with objects as content. Expected extensions (sic): *Present It!*.

Vault A tool to manipulate circuit descriptions in VHDL. Vault is described in the next section.

Miscellaneous

see An application-specific database for an AltaVista-like web search service (`http://www.altavista.digital.com`), accompanied by a spider and a web server.

dnews A usenet news server that employs a PostgreSQL database to store news articles and that communicates with the client through NNTP.

$\delta\eta\mu\omega\iota$ A collection of *CGI scripts* that run from a web server to maintain a database of news items with user comments and quality weighing of items and comments through a voting system. Similar web-site concepts are increasingly popular and can be seen in successful action on, for example, Slashdot (`http://slashdot.org`). $\delta\eta\mu\omega\iota$ employs `tdbc`.

mod_TOM A module for the Apache web server allowing Apache's functionality to be extended in TOM, similar to the way in which mod_PERL allows Apache to be extended in Perl (see `http://www.apache.org` and `http://www.perl.org`).

The extensibility of code that TOM provides allows for example the `mu` unit of TOM parsers to step lightly over issues concerning its use. It does not matter whether the `mu` classes will be subclassed before use or simply extended. In either case, extensibility enables easy reuse.

This observation is generally true for library units written in TOM. TOM libraries allow a large degree of freedom to their users. Additions and slight modifications to a class are easily done, without the need for subclassing or other complexity increasing constructs.

6.8 Vault

Vault is a tool that has been developed to perform clock gating on register-transfer level (RTL) circuit descriptions in VHDL. Clock gating is a technique to reduce the power consumption of synchronous digital sequential circuits. This power reduction is achieved by suppressing the clock signal to flipflops at moments when the new value is identical to the current value. Since generating a gated clock from the normal clock signal requires an additional flipflop, to save power, multiple flipflops must be clocked by the same gated clock. Flipflops governed by the same clock reside in the same *clock domain*. The complexity of clock gating lies in the process of determining the clock domains.

The operation of Vault was previously described in [34]. This section explains its architecture.

Figure 6.7 depicts the unit architecture of Vault. Every box denotes one of the units making up the program and an arrow denotes a dependency. Italic names denote pre-existing library units; the other units were developed to create Vault. Furthermore, the grey units and dependencies do not involve extensions.

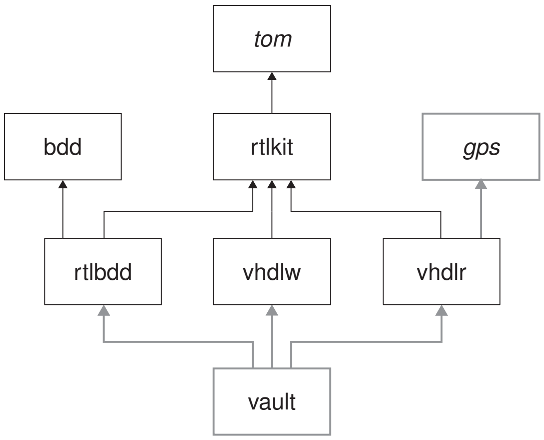

Figure 6.7: The units architecture of Vault.

Central to Vault is the `rtlkit` library unit which consists of a large number of classes to represent circuit descriptions and instantiations thereof, e.g., N-bits adders and a circuit employing several 8-bit adder instances. An instantiation can be created from a description, and *vice versa*. Though the vocabulary employed in `rtlkit` matches the vocabulary of VHDL, `rtlkit` is not specifically targeted at VHDL. In fact, `rtlkit` can not read or write descriptions in *any* language, though it is designed to be general enough to be usable for both VHDL and Verilog descriptions.

The `rtlkit` unit extends the `OutputStream` class of the `tom` unit to maintain a level of indentation. This enables easy creation of properly indented and readable output.

The `vhdlw` unit extends the `rtlkit` classes with the ability to emit themselves in VHDL. Other units could be developed to add output in additional languages. Nothing in the design of `rtlkit` needs to cater for this kind of output, since this functionality is added directly to the affected classes instead of being added in the form of code that operates on those classes.

The `vhdlr` unit adds to `rtlkit` the ability to read VHDL code. `vhdlr` employs `gps` for its definitions of an abstract parser and lexer. The `vhdlr` parsers are however not created by `gp`; most of `vhdlr` is glue code to the actual parser that comes as C code.

The `bdd` unit is glue code to a BDD library written in C [25]. The `rtlbdd` unit employs the `bdd` unit to extend the `rtlkit` class that represents nets in an instantiation with the ability to reason about their boolean function. `rtlbdd` also extends the `bdd` unit to directly reference from each BDD variable the net to which it corresponds.

The `vault` unit simply is the main program that parses the command line and invokes the required functionality as offered by the library units. This functionality can be summarized as follows: read the VHDL source files (`vhdlr`), instantiate the top entity (`rtlkit`), compute the next-state values of all clocked flip-flops (`rtlbdd`), compute the clock domains and modify the affected instantiations (`rtlbdd`), create new architectures and entities for those modified instantiations (`rtlkit`), and emit the new descriptions and a configuration to use them (`vhdlw`).

An advantage of the library-oriented setup to Vault is that each part is reusable without the burden of the unnecessary parts. `rtlkit` is the core library for RT-level circuit descriptions. `vhdlr` is only needed for applications that need to read VHDL code. Similarly, `vhdlw` is only needed for VHDL output. `rtlbdd` is only needed when information is required about the boolean functions on the nets. This setup would not have been possible without the extensibility of code as provided by TOM.

Chapter 7

Conclusions

7.1 Achievements

This dissertation describes flexibility of program code through the extensibility of classes. An extensible class is open to modification, most notably through the addition of methods, object variables, and superclasses. The extensibility of classes provides significant advantages to the developers of past, present, and future code:

future Extensibility allows the developer of future code to adjust the classes from past code. Such changes can range from an occasional method addition to complex extensions to the original design, for example to add facilities for automated software testing.

present Extensibility enables an organization of classes that is not dictated by the boundaries of source files. A source file can contain any number of classes and the definition of a class can be distributed over any number of source files.

past Extensibility allows the developer of past code to apply changes to an evolving library without requiring recompilation of all future code. On a computer system employing shared libraries, this means that an incompatible version increment is necessary far less often than without extensibility.

Extensibility reduces the demand on the library developer to deliver a complete, feature-full, library of widely reusable components. Future code can extend the library at will as necessary, allowing the library developer to focus on the design of the structure instead of the features.

An important concept in object-oriented programming is *encapsulation*, i.e., the hiding of implementation details. Encapsulation is usually enforced at compile time, directed by source code annotations. In support of extensibility, TOM provides mechanisms to break the encapsulation defined in source code, which is possible only by strict encapsulation in object code and at run time. In addition, this allows classes to be extended even if their source code is not available.

Run-time encapsulation requires a level of indirection to every access of a method or object variable. In the reference implementation of TOM described in this dissertation, the computational overhead is limited, by precomputation and caching, to few indirections upon every access, without requiring much memory. Exact figures are not available since exact measurements are difficult to conduct. The computational overhead is in the order of 10% for method binding and less than 1% for variable binding.

The flexibility that is offered by extensibility offers a significant advantage over the flexibility inherent to many frameworks and libraries: whereas the *addition* of flexibility requires human creativity, the *removal* of flexibility can be performed by a compiler. The different requirements of compilation during program development (high speed) and compilation of production code before deployment (high quality) can be accommodated by maintaining full flexibility during development and removing as much flexibility as possible and desired upon deployment.

7.2 Future work

Research into the flexibility of code, using TOM as a vehicle, may continue in several directions.

- Finish the development of `tesla`, the TOM compiler written in TOM, to complete it as a whole-program compiler. Whole-program compilation of TOM code would enable the use of TOM in resource-constrained applications. Furthermore, it could show the usefulness of partial flexibility of code and its automatic derivation from full flexibility as opposed to the common *ad hoc* manual addition.

- Once TOM is an accepted object-oriented programming language and used by real people in real projects, the true usefulness of flexibility of code through extensibility can be quantitatively measured. Possible measurements include the number of extensions employed in a program with respect to various other quantities or, more difficult, the design or implementation time saved by using them.

As with all programming languages, the most important goal for TOM is to be used by many programmers. In pursuit of that goal, the availability of a good tutorial on programming in TOM would be a fine first step.

Glossary

block What is known as a *block* in some languages is referred to as a compound expression in TOM. A block, from Smalltalk, is a compound expression of which the evaluation is postponed: the block becomes an object of which a method is invoked to trigger evaluation.

cast (verb) To change the type of a value. Most casts are performed implicitly, since explicit casting is nasty. For example, if A is a subclass of B, which in turn is a subclass of C, then an expression with type B can be implicitly cast to C but must be explicitly cast to A.

category A collection of methods that can be added to a certain class. The *category* is a concept used by Smalltalk and Objective-C.

class A class defines a type along with its operations, and provides an implementation of that type and the operations. Instances of the class will fit the type.

class messaging Sending messages to a class object.

class object In programming languages that provide class objects, every class is represented by its class object.

class method A method of the class object. In a language without an explicit object denoting a class, the concept can still be present as a method that does not depend on the state of a particular instance.

class variable Conceptually, an object variable that belongs to a class object. Also in languages without class objects, the term is often used to denote a global variable that is declared in the scope of a class, hence available only to the instances of that class.

client A person or entity that uses or enjoys certain services. Mostly used from the perspective of the provider of the services.

client code Used in the context of a class A, client code is code that uses A, possibly as a subclass but most often from an unrelated source.

code Something that can be executed. Popular appearances are *source code*, which is written by a human, and *machine code*, which can be read by a machine.

compile (verb) To translate utterings in a source language to utterings in a target language, while preserving functionality. The target language can be readable—many compilers translate to C—but usually it is machine code.

compiler A program that compiles, usually reading source files and writing object files.

compile time While the compiler is running, i.e., during the translation of source code to object code.

conformance A type B conforming to a type A implies that all operations that are supported by type A are also supported by B. (The reverse is not true.)

curry (verb) To provide a function with partial arguments, the result of which is a new function with less arguments.

declare (verb) To inform the outside world of the existence of something, without providing an implementation of it.

deferred class A class that does not implement all methods that are present in its interface. A class is deferred if it declares some methods without defining them, or if it inherits from a deferred class, without implementing all deferred methods.

deferred method A method that causes a class to be a deferred class. A deferred method has been declared but not defined.

define To provide an implementation. Often, a definition also serves as a declaration.

dereference Literally, to *remove a reference*. Thus when considering a pointer, dereferencing it means to consider the value at which the pointer points.

development time The period of time during which something is developed. This ranges from a few minutes for a trivial example to multiple years for Very Large Programs.

dispatch See *method dispatch*.

dynamic library A shared library which is dynamically linked to a process as part of starting the program.

dynamic linking To link at run time. This most often refers to combining the code of a process with that of a plug-in, but is also applicable at process startup, when associations of the program with the dynamic libraries are resolved.

dynamically At run time. Opposite of statically.

encapsulation Hiding something, e.g., object variables or implementation details.

exception An exceptional situation, or an object employed for the indication of such a situation. Usually associated with the unwinding of a stack.

expanded class Instances of an expanded class are always passed by value, as opposed to the *normal* by-reference passing.

extensibility The possibility of being extended. In this dissertation, extensibility denotes the possibility to modify the behaviour of objects.

flexibility Something is flexible if it can be easily extended, or applied in circumstances not envisioned by the designer (not if it *bends well*).

glue code Code that enables code in one language to use use functionality that is offered by code written in another language.

implementation In general, an implementation fulfills the promises made by an interface. In object-oriented programming terminology, the implementation of a class consists of its methods definitions and variables.

implementation file A source file that contains definitions. An implementation file provides the implementation of, e.g., a class.

implementation inheritance Inheritance by a class, which inherits both interface and implementation from the superclass.

inherit To acquire certain features, such as object variables, methods, and a position in the inheritance hierarchy.

inheritance hierarchy A *directed acyclic graph* (DAG) in which each node is a class and an edge from the class depicted by the source node inherits from the class of the destination node.

inline display To display something within a document, instead of displaying only a reference.

instance An object, with the explicit exclusion of class objects. Any number of instances can be created at run time.

instance variable A variable is a name of a storage location. With an instance variable, the storage location is contained in an instance, i.e., the instance variable is part of the object's state.

interface In general, the outside appearance. In object-oriented programming terminology, the interface of a class is defined by its behaviour, i.e., by a declarations of the available methods. As such, an interface and a type are very much alike.

interface file A source file that contains declarations. An interface file provides the interface of, e.g., a class.

interface inheritance Inheritance of only interface, as opposed to implementation inheritance. Usually unqualified *inheritance* is not interface inheritance.

kind Referring to some, often weak, classification in a context where using the words *type* or *class* may cause confusion. For example, *kinds of objects* certainly does not refer to their classes.

library A collection of object files, accompanied by interface files. A library provides functionality that can be used by many programs.

link (verb) To combine object files, possibly retrieving them from libraries in order to resolve undefined references. This activity is performed by the linker to combine libraries and object files into an executable.

linker A program that links.

load (verb) A synonym for link.

member C++ terminology for anything that is part of a class. A *member function* is a method, a *member variable* an instance variable, unless it is `static`, in which case it is a static class variable.

message A conceptual entity, used in *sending a message to an object*. The message contains argument values and an indication of the method signature of the method that is to be invoked. (The class of the receiver of the message determines which method actually matches.)

messaging `super` Sending messages to `self` while acting to be an instance of a direct superclass instead of the real class of `self`. Frequently used to invoke the original method from an overriding method in a subclass.

method A piece of code associated with a class. A method is an implementation of a certain operation. Apart from the class association, a method is much like a function in a programming language like C.

method activation The execution context of a method that has not finished yet, usually discernible as a stack frame.

method dispatch The mechanism underlying the invocation of a method as a result of sending a message to an object.

method overriding Defining a method that would otherwise be inherited.

method signature The *outside* of a method, i.e., its name, argument types, and return type.

multi dispatch Determining which method to invoke based on all arguments of the invocation. The term *message-send* no longer applies since a single receiver object can not be discerned.

multiple inheritance Inheritance that allows multiple direct superclasses. Opposite of single inheritance.

non-local return A return from a method, but not necessarily returning to the method caller. A non-local return can return from *any* outstanding method invocation, instead of only the most recent one (i.e., it can return *to* any outstanding method invocation instead of only the caller of the current method).

object A piece of data that is encapsulated by methods. The only way to interact with the object is to invoke its methods; the data is not directly accessible.

object code Machine instructions in an object file.

object file A partial program, the object file contains machine code with partially unresolved subroutine calls. See link.

operation A type offers certain operations that can be performed on values of the type. A type is implemented by a class and the operations by the class' methods.

platform An executable program can only run on one particular platform, which is defined by the combination of the hardware and the operating system.

plug-in An extension of a program, a plug-in is object code that can be dynamically loaded into a program. A plug-in is usually developed, maintained, and upgraded independently of the program.

polymorphism A mechanism underlying method invocations that obeys the actual class of the receiver of the message-send. When the object is an instance of a subclass of the class assumed by the caller, polymorphism ensures that the method dictated by the subclass is invoked.

process A running program.

proper subclass A proper subclass of a class A is any subclass of A, except A itself.

proper superclass A proper superclass of a class A is any superclass of A, except A itself.

receiver The object that receives a message. It is the `self` in the method that is invoked as a result.

resend Within a method of a subclass, a resend passes the method invocation to the method of a superclass. This is a less flexible variation on messaging super.

run time While the program is running. Frequently used as opposite of compile time.

selector Part of a message-send, the selector dictates the name, argument types, and return type of the method that is invoked in response to the message. See also *method signature*.

`self` In many programming languages, the variable referring to the receiver of the message which caused the invocation of the method. This actually is an implicit argument.

scope Visibility, most often of the names of types and variables.

shared library A library of which the object code is not duplicated in every program but shared between them. The cleanest implementation is provided by dynamic libraries.

shared object Different name for a dynamic library.

signature See *method signature*.

single dispatch Determining which method to invoke based on the receiver of the message-send.

single inheritance Inheritance that allows only a single direct superclass. Opposite of multiple inheritance.

source code Input to a compiler; usually written by humans.

stack The stack of a CPU.

stack variable A variable of which the storage location resides on the stack. When the variable goes out of scope, for instance because the method exits, the storage location is destroyed.

state The state of an object denotes the data carried in all object variables of the object.

static Usually used to mean that something is defined or known statically, i.e., at compile time.

statically At compile time. Opposite of dynamically.

storage location A location, of unspecified size, at some address in memory.

subclass (noun) A subclass of a class A is A itself or any class that inherits from A, directly or indirectly. See also *proper subclass*.

subclass (verb) To subclass a class A is to create a class B as a subclass of A, and to use B instead of A.

subtype A type A is a subtype of type X, when values of type A can also be considered as values of type X. Compare with *subclass*.

super A name for `self` acting to be an instance of a superclass. This can only be used when messaging super, i.e., as the receiver of a message.

superclass A superclass of a class A is A itself or any class from which A inherits, directly or indirectly. See *proper superclass*.

test extension An extension of a program that, like a plug-in, is loaded at run-time. The goal of a test extension is to aid in testing the program.

thread A thread of control in the address space of a process. Multiple threads can run in a process. Each thread has its own CPU state and accompanying stack. On a multi-CPU machine, threads may run concurrently.

type A constraint on the operations that are supported by values of that type.

unit testing Testing a unit of code for its basic functionality. Prime example of such a unit is a class.

variable A storage location with a name and a type.

value Something that resides at a storage location and has a certain type.

virtual A C++ keyword that allows the programmer to specify that the member function to which it is applied exhibits polymorphism.

whole-program compiler A compiler that considers all source code of the program that it compiles. Opposite of a *normal* compiler for which the unit of compilation is a single source file.

wrapper Something that wraps around something else, to give it a different appearance or interface. When certain objects, usually defined in past code, do not conform to a type to which they should conform, each of those objects can be wrapped by a wrapper object which does conform to the desired type.

Bibliography

[1] Adobe Systems. *Programming the Display PostScript system with NeXTstep.* Addison-Wesley, 1992.

[2] Gerald Baumgarter and Vincent F. Russo. Signatures: A language extension for improving type abstraction and subtype polymorphism in C++. *Software Practice and Experience*, 25(8):863–889, August 1995.

[3] Boris Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, second edition, 1990.

[4] Grady Booch. *Object-Oriented Analysis and Design, with Applications.* Benjamin/Cummings, second edition, 1994.

[5] Craig Chambers. *The design and implementation of the SELF compiler, an optimizing compiler for object-oriented programming languages.* PhD thesis, 1992.

[6] Craig Chambers, Jeffrey Daen, and David Grove. Whole-program compilation of object-oriented languages. Technical Report 96-06-02, University of Washington, Seattle, WA, June 1996.

[7] Graig Chambers. The Cecil language—specification and rationale. Technical report, University of Washington, Seattle, WA, March 1997.

[8] Patrick Chan, Rosanna Lee, and Douglas Kramer. *The Java class libraries*, volume 1. Addison-Wesley, second edition, 1998.

[9] Dominique Colnet and Suzanne Collin. SmallEiffel the GNU Eiffel compiler. `http://smalleiffel.loria.fr`.

[10] Brad Cox. *Object-oriented programming: an evolutionary approach.* Addison-Wesley, second edition, 1991.

[11] Edsger W. Dijkstra, Leslie Lamport, A.J. Martin, C.S. Scholten, and E.F.M. Steffens. On the fly garbage collection: an excercise in cooperation. *Communications of the ACM*, 21(11):966–975, November 1978.

[12] Free Software Foundation. *GNU General Public License*, June 1991. Version 2.

[13] Free Software Foundation. *GNU Library General Public License*, June 1991. Version 2.

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

[15] Adele Goldberg and David Robson. *Smalltalk-80: the language*. Addison-Wesley, 1989.

[16] Charles F. Goldfarb. *The SGML Handbook*. Oxford University Press, October 1990.

[17] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996. Version 1.0.

[18] Hanspeter Mössenböck. Extensibility in the Oberon system. *Nordic Journal of Computing*, 1(1):77–93, 1994.

[19] William Harrison and Harold Ossher. Subject-oriented programming (a critique of pure objects). *ACM SIGPLAN Notices*, pages 411–428, 1993.

[20] Rachid Helaihel and Kunle Olukotun. Java as a specification language for hardware-software systems. In *International Conference on Computer-Aided Design*, San Jose, CA, 1997.

[21] Urs Hölzle. Integrating independently-developed components in object-oriented languages. In *Proceedings of the European Conference on Object Oriented Programming*, Kaiserslautern, July 1993. Springer Verlag.

[22] Support for subject-oriented programming in C++. `http://www.research.ibm.com/sop/sopmont.htm`.

[23] *5th International Conference on Software Reuse*. IEEE Computer Society, June 1998. `http://www.cs.vt.edu/icsr5/`.

[24] *IEEE Standard Portable Operating System Interface for Computer Environments (POSIX, IEEE Std 1003.1-1988)*. IEEE, 1988.

[25] Geert L.J.M. Janssen. The Eindhoven BDD Package. `ftp://ftp.ics.ele.tue.nl/pub/users/geert/bdd.tar.gz`.

[26] David Ku. Hardware-C – a language for hardware design. Technical report. Version 2.0.

[27] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[28] Bertrand Meyer. *Reusable Software: the base object-oriented component libraries*. Prentice-Hall, 1994.

[29] Bertrand Meyer. *Object-oriented software construction*. Prentice-Hall, second edition, 1995.

[30] Bertrand Meyer and Éric Bezault. Dynamic linking in Eiffel. Technical Report TR-EI-49/DL, Interactive Software Engineering, February 1996.

[31] Harold Ossher and William Harrison. Combination of inheritance hierarchies. *ACM SIGPLAN Notices*, 27(10):25–40, 1992.

[32] Pieter J. Schoenmakers and Jochen A.G. Jess. Facilities for testing control software. In *IEEE High Level Design Validation and Test workshop 1997*, November 1997.

[33] Pieter J. Schoenmakers and Jochen A.G. Jess. Testing software suffering from hardware. In *ProRISC/IEEE workshop on Circuits, Systems, and Signal Processing 1997*, November 1997.

[34] Pieter J. Schoenmakers and J. Frans M. Theeuwen. Clock gating on RT-level VHDL. In *International Workshop on Logic Synthesis 1998*, pages 387–391, June 1998.

[35] The Self project. `http://www.sunlabs.com/research/self/`.

[36] NeXT Software. *Object-oriented programming and the Objective-C language*. 1993.

[37] *1997 Symposium on Software Reusability*, May 1997. `http://www.lfs-owego.com/~ssr97/`.

[38] Richard M. Stallman. *Using and Porting GNU CC*. Free Software Foundation, 1995.

[39] Richard M. Stallman. *The Emacs Editor*. Free Software Foundation, 13th edition, 1998.

[40] Guy L. Steele. *Common Lisp the language*. Digital Press, second edition, 1990.

[41] Bjarne Stroustrup. *The C++ programming language*. Addison-Wesley, third edition, 1997.

[42] Kresten Krab Thorup. Optimizing message lookup in dynamic object-oriented languages with sparse arrays. In *FreeSoft 1993*, Moscow, Russia, March 1993.

[43] David Ungar and Randall B. Smith. Self: The power of simplicity. *ACM SIGPLAN Notices*, 22(12):227–241, December 1987.

[44] Unicode Consortium. *The Unicode Standard, Version 2.0.* Addison-Wesley, September 1996.

[45] N. Wirth and J. Gutknecht. The Oberon system. *Software Practice and Experience*, 19(9):857–893, September 1989.

[46] *8th Annual Workshop on Institutionalizing Software Reuse*, March 1997. `http://www.umcs.maine.edu/~ftp/wisr/wisr8/wisr8.html`.

Index

Pagenumbers in **bold** reference a definition of the term;
italic pagenumbers reference an entry in the glossary.

Biography

Pieter Schoenmakers was born on 8 May 1967 in Breda, the Netherlands. He attended grammar school *het Stedelijk Gymnasium* in Breda before starting for a degree in Electrical Engineering at Eindhoven University of Technology in August 1985. After 4 years he took a 6-month break to work as a programmer in the MACH group at Acorn Computer in Cambridge, England. Too many interesting projects, part-time diversions, and service in the Dutch army later, he graduated on 16 December 1993. In the next year, he sold computers, did consulting work, and ended up developing audiotex software in Amsterdam. On 1 January 1995 he started as a Ph.D student in the Design Automation Group of the department of Electrical Engineering at the Eindhoven University of Technology, the result of which being the dissertation you are reading.

Pieter's research interests include everything that people (developers, really) need to use or create computers and derivatives: languages, compilers, libraries, operating systems, networking software, and computer hardware. Pieter's professional interests include every challenge for which his knowledge, experience, and understanding will be valuable tools.

The future is uncertain, as it should be.

Stellingen bij het proefschrift

Supporting the Evolution of Software

door Pieter J. Schoenmakers

1. Types zijn generalisaties en bijgevolg te strikt.
2. Het ontwerpen van een programmeertaal met één specifieke compiler-implementatie in gedachte gaat voorbij aan het feit dat tijdens de ontwikkeling van een programma vooral de compiler snel moet zijn terwijl na de ontwikkeling nog slechts de snelheid van de gecompileerde code telt.
3. Het nut van het keyword `const` wordt in de C++ *standard template library* geïllustreerd door het bestaan van zowel `iterator` types als ook `const_iterator` types.
4. Een programma is een executeerbare specificatie.
5. In een ontwikkelomgeving met meerdere programmeurs en *exclusive write locks* staat het exclusieve recht een file te modificeren op gespannen voet met de universele plicht bugs te fixen.
6. Een leeftijdsgrens aan de verkoop van tabak, en het daarmee impliciet gedogen van roken in bepaalde leeftijdscategorieën, zou beter een bovengrens zijn.
7. In een sorry-democratie geldt niet langer: Een man, een man, een woord, een woord.

   [*Peper komt na fout terug op politie-CAO*, de Volkskrant, 28 januari 1999]
8. De mogelijkheid een binnenkant te bekijken
is een uitnodiging de buitenkant te begrijpen.
9. Specialisten—de tegenhangers van generalisten—doen zichzelf en hun werkgever tekort.
10. De softwarecrisis wordt veroorzaakt doordat de mensen met de juiste capaciteiten de kunst van het programmeren verloochenen.
11. 'Bezint eer ge begint,' maar vooral 'reflecteer daarna weer.'
12. Twijfel aan uw gereedschap!