Reusability in the Smalltalk-80 Programming System

L. Peter Deutsch

Xerox PARC, Software Concepts Group

The Smalltalk-80 interactive programming system supports three different kinds of re-use. Smalltalk-80 is designed to be portable to a variety of machines with modest effort, allowing re-use of the *complete system* across *hardware* environments; the Smalltalk-80 language includes a powerful form of data abstraction allowing re-use of *algorithms* across a variety of *data structures*; built-in facilities of the Smalltalk-80 system include many building blocks and conventions for implementing interactive applications (re-use of a *framework* across *applications*).

INTRODUCTION

In everyday problem-solving, we can re-use tools, procedures, or approaches that have worked in the past, adapting them to the situation at hand. Solving problems in the computer setting is often a much more frustrating process, to a large extent because adaptive re-use is often difficult or impossible.

The phrase "tools, procedures, or approaches" alerts us to the fact that there are several qualitatively different ways to re-use past experience. A physical tool designed for one purpose may be usable for another (e.g. a hammer to drive nails or to flatten out a dent in a sheet of metal.) A procedure designed to solve one particular problem may be adaptable to another (e.g. a recipe for broccoli used to cook cauliflower with small changes in the spices and cooking time.) An approach may be applicable across a wide range of problems (e.g. the "divide and conquer" paradigm used to organize a group of people addressing envelopes, or a program processing a graphical image.) What is being re-used may be a physical object, a mental or physical process, or a combination..

The Smalltalk-80 system is designed to facilitate re-use of both code and design, at scale levels ranging from entire systems to individual algorithms and data structures, and across a range of both hardware (which supports the system from "below") and applications (which employ the system from "above"). The following sections spell out those aspects of the Smalltalk-80 design and implementation which particularly support reusability.

Proceedings
Workshop on Reusability in Programming
7-9 September 1983
Newport, RI

ALGORITHM REUSABILITY ACROSS DATA STRUCTURES

The greatest obstacle to the re-use of algorithms is their dependence on the detailed (concrete) implementation of the data they manipulate, rather than only on its necessary logical (abstract) properties. The traditional way around this problem is to pass to the algorithm parameters that describe the data, either data parameters (e.g. the offset and size of a key within a record) or procedural parameters (e.g. an ordering function for sorting). This parametrization can occur either at run time or at compile time (the latter through the use of in-line procedures or macros, or simply through constant folding optimizations.) A more recent extension of this idea is the use of *data abstraction*, in which some or all of these parameters are packaged up with a data type, and the application of a procedure to objects of a particular type automatically substitutes into the procedure the correct (data or procedure) parameters for that type.

Smalltalk-80 strongly encourages use of abstraction in two ways: by restricting the ability to refer to the implementation of a data type to a set of procedures associated with it by its implementor, and by providing abstract as well as concrete data types arranged in a hierarchy which can be used for specialization or extension. Both of these ideas originated in the Simula language: in Smalltalk-80, however, they are applied to *every* object in the system, rather than being an addition to the Algol world.

In Smalltalk-80, the fundamental unit of organization is the *class*, which consists of a description of a data object plus a set of named procedures that have access to this description. Every object in the system is described by (is an *instance* of) some class. (This includes "primitive" objects such as integers and strings, and "internal" objects such as compiled procedures, processes, and classes themselves.) The description of a data object consists of a set of names for its parts, e.g. a class representing a lookup table might have two parts named **keys** and **values**: Smalltalk-80 does not currently use type declarations, so the description of a part just consists of the name. Only the procedures associated with a class can refer to the parts directly: any access to an object from outside its class must invoke a procedure defined in the class. In fact, the fundamental (and only) procedure call operation in Smalltalk-80 treats all objects as abstract: invoking the procedure named **op** with operands **obj1** ... **objN** means looking up the *name* **op** in the procedure dictionary associated with the class of **obj1**, and then executing the procedure body with the parts of **obj1** directly accessible by name and the formal parameters bound to obj2 ... objN. The distinguished operand obj1 is called the *receiver* of the invocation, since it has the responsibility for deciding which implementation of op to use. Thus every operation is potentially generic -- implemented by more than one kind of object; also, since there are no language constructs for directly accessing the internal implementation of an object from outside, one class of objects can masquerade as another.

Classes are arranged in a hierarchy (formerly a tree, now a lattice) with the property that if class A is a subclass of class B, then all operations implemented in B are recognized by instances of A and have the same meaning (unless reimplemented in A, or in some class between B and A in the hierarchy). As a consequence, algorithms can be implemented once at higher or more abstract levels of the hierarchy, and are automatically available at lower levels. For example, some common searching algorithms are implemented in an abstract class called Collection that is a common superclass of Set, Array, and Dictionary; some mathematical notions like absolute value are implemented in the abstract class Number and available automatically to Integer and Float. Thus when one builds a new class for some application two important benefits arise:

- new classes that are similar to old classes can be built with an effort that is proportional to the degree of dissimilarity;

- default implementations of common operations are available with no effort, and can be tuned later without changing any client programs (since the implementation in the subclass will take precedence over the implementation in the superclass).

An important programming technique in Smalltalk-80, similar to the *inner* mechanism in Simula, is to create abstract algorithms that rely on concrete subclasses to provide some missing parts. For example, here is the procedure that implements linear searching in the abstract Collection class (refer to the Appendix for a summary of Smalltalk-80 syntax):

```
includes: anElement
    self do:
        [:eachElement |
        eachElement = anElement
            ifTrue: [↑true]].


    ↑false
```

The basic iteration message do:, which generates each elements of the collection in turn, cannot be implemented at the abstract level. Its definition in class Collection is

```
do: aBlock
    "Invoke aBlock with each element of this
    collection."
    self subclassResponsibility
```

which generates a run-time error if invoked without an overriding definition in the concrete subclass, and which serves as documentation to subclass implementors. In this way, abstract classes can include not only re-usable algorithms but a rudimentary specification of what is required of their concrete subclasses.

The class hierarchy tends to soften the distinction between design and implementation at the algorithm level. Algorithms which depend on very few properties of their operands can be written as executable code in highly abstract classes. However, a strong separate design element remains in the assignment of functional roles to classes, the definition of the protocols (operations and their meanings) for different classes, and the construction of the class hierarchy itself.

FRAMEWORK REUSABILITY ACROSS APPLICATIONS

In the previous section, we concentrated on the fact that an algorithm could be implemented in an abstract class and re-used with more than one concrete subclass. A different way of looking at this arrangement is that a collection of abstract classes, and their associated algorithms, constitute a kind of *framework* into which particular applications can insert their own specialized code by constructing concrete subclasses that work together. The framework consists of the abstract classes, the operations they implement, and the expectations placed on the concrete subclasses as described in the previous section.

The Smalltalk-80 user interface is the most outstanding example of this framework viewpoint. It is based on a uniform model of "viewing" objects. To interact with an object (called a *model*), three components are required:

- a *view* object must exist that knows how to convert some interesting aspect(s) of the model to visible form.

- a *controller* object must exist that knows how to interpret user-initiated events (button clicks, keystrokes, movement of the pointing device) as selection and editing commands in the functional space provided by the view.

- the model itself must provide interfaces that allow the view to access and update the aspects being viewed. Since Smalltalk does not allow the internal structure of an object to be accessed directly from the outside, these "aspects" may be as simple as individual state variables, or arbitrarily complex characteristics mediated by accessing and updating procedures.

For example, there is an abstract class View that provides much of the mechanism for handling screen clipping, coordinate conversion, automatic updating of the screen when the model's state changes, and so on. Concrete subclasses are expected to provide an implementation for the operation displayView, which redisplays the representation of the selected information from the model within the appropriate area on the display screen. For simple concrete views, implementing this single operation is sufficient.

The same view object, if properly designed, can be used with many different kinds of model objects, and the same model can be viewed in many different ways. Smalltalk-80 provides several kinds of views and associated controllers that have been successfully re-used across many different applications, and that in fact form a kind of standard "user interface builder's kit" that writers of new applications naturally employ rather than building their own from scratch. Here are some examples of such view/controller structures:

- Pop-up menu: a list of alternatives that appears on the screen when appropriate (a button being depressed, or a query command being issued that returns its result in this form), allowing the user to select one or none of the alternatives.

- List view: a list that remains on the screen, can be scrolled within a clipping region, and whose selected element is displayed in some other view on the screen.

- Text view: a region that contains editable text, with a standard command repertoire (copy, cut, paste, search, undo, etc.) that can be extended by an individual application (e.g. 'accept' for compiling code, 'put' for storing a document).

- Switch/button: a region that initiates some action when a button is depressed while the cursor is within it, or that simply retains its on/off state.

- Inspector: a combination of a list view and a text view which views a list of state variables of a model. The variables can be viewed one at a time in the text view, and altered by typing in a new value.

The subclassing mechanism described in the previous major section plays a crucial role in making views re-usable. For example, the code view used by the source code browser is a subclass of the standard system text view, adding only procedures for three commands: 'accept' which invokes the compiler, 'explain' which provides an explanation of source code constructs or names, and 'format' which reformats the source code with standard indentation and spacing.

SYSTEM REUSABILITY ACROSS HARDWARE

Smalltalk-80 adopts the usual approach to software reusability across variant hardware: it is based on an ideal *virtual machine* (VM) into whose instruction set the system is compiled. VM instruction sets can be mapped onto real hardware in two ways:

- A retargetable compiler can translate the VM instruction set into the hardware instruction set. In this case the VM instruction set is just one of several intermediate representations used by the compiler, and may well not be visible to programmers. This is the approach taken in the Unix system.

- An interpreter, written in the hardware instruction set, can interpret the VM code at run time. This is the approach taken in the Pascal P-system, and also by Smalltalk-80.

The Smalltalk-80 VM is a simple stack-oriented machine similar in structure to the Pascal P-system machine, but different in several vital respects:

- There are no compile-time type declarations: all objects are tagged at run time with their type. All objects (except small integers) are represented by pointers.

- Storage reclamation is automatic. Implementors may choose to use reference counting, garbage collection, or any other suitable technique. (Since Smalltalk-80 is designed as a highly interactive system, most implementors have chosen reference counting, since it produces the shortest unexpected delays in interactive response.)

- All data structures used by the system in its operation are covered by the type system, and are visible to the programmer. This includes procedure activation records, processes, the process scheduler itself, the objects that represent types, etc. As a result, programmers writing entirely in the Smalltalk-80 language can implement (indeed, have implemented) debuggers, new control structures, scheduling policies, substantially different source languages, etc.

- The VM includes a complete I/O system, based on virtual I/O devices that correspond directly to customary hardware. This is discussed in more detail below.

- The VM also includes a process scheduler, based on semaphores and very cheap processes. The I/O system uses semaphores and processes exclusively: there is no notion of an "interrupt" other than signalling a semaphore which enables a high-priority process to run.

In addition to the VM instruction set, the Smalltalk-80 VM includes about 100 "primitive procedures". In many systems, a facility may appear to the programmer either as a special language construct or as a procedure, and may be implemented either as a VM instruction or as a library procedure, with no correlation between the two. In Smalltalk-80, language constructs (of which there are very few) are implemented with VM instructions, and all other capabilities are implemented as primitive procedures, invoked with exactly the same syntax as user-defined procedures. This even includes such things as arithmetic, storage allocation, access to parts of objects (such as fields of a record), and many control structures.

Starting with this modest VM, the Smalltalk-80 system provides a complete interactive programming system, including a compiler, decompiler, code editor, display-oriented debugger, and structured code browser and query facility; application building-blocks including graphics, arbitrary precision integer, and Ethernet communication packages; and finished applications such as a graphics editor and a version manager. The total size of the Smalltalk-80 system is about 900K bytes of object code and data, and about 1.3M bytes of source code. All of the source code and internal documentation for the system is available on-line.

Implementing the VM typically takes about 40K bytes of object code regardless of the level at which it is done (this figure is constant, to within a factor of 2, across implementations in microcode, assembly language, and C.) A good implementor can create a straightforward VM implementation in a couple of months, since an executable version of the VM specification has been published and need only be transcribed from "machine-oriented Smalltalk" into some low-level language. On the other hand, a sophisticated implementor has a good deal of leeway in how to implement certain critical parts of the system, such as storage reclamation or message sending (described earlier): implementations on the same hardware have differed by as much as a factor of 5 in performance.

The VM I/O system

The Smalltalk-80 I/O system incorporates the following devices:

- An "undecoded" keyboard, in which every transition of a key up or down produces an event.

- A pointing device which can be sampled to produce an X/Y position.

- Three buttons, whose transitions produce events.

- A black-and-white bitmap display of reasonable size (512 x 512 pixels is the bare minimum, and at least 600 x 800 is recommended.)

- A timer, which can be set to cause an event after a specified number of milliseconds.

- A calendar clock, which gives the date and time.

An I/O "event" means that the VM implementation produces a signal on a Smalltalk-80 semaphore object, and, if relevant, saves any associated input data in an fixed-size internal buffer, which a Smalltalk-80 program reads with a primitive procedure.

In addition to these standard devices, Smalltalk-80 includes a file system which can be interfaced either to a disk at the physical record level, or to an underlying operating system such as Unix. The file system exploits the class hierarchy by providing abstract File and FilePage classes which knows how to manage files implemented by fixed-size physical records (including buffering, allocating and deallocating pages, etc.): a Smalltalk-80 system running on top of Unix would discard most of this logic, while a system managing a physical disk would only need to provide a subclass that transferred individual records and kept track of free records. Some Smalltalk-80 systems also include support for the Ethernet communication network, again providing all the software (customarily found in an operating system) to implement higher-level protocols starting from the physical packet level: remote files are supported in exactly the same way as local disk files.

UNSOLVED PROBLEMS

Even though the Smalltalk-80 approach to reusability has succeeded substantially in all three of the areas mentioned above, there are substantial unsolved problems in each of them.

The class approach to abstraction, the subclass mechanism for extension and specialization, and the use of a single structuring mechanism for all data and procedures in the system have worked extremely well. However, problems arise in a number of areas. As a number of other researchers have observed, a tree structure for class inheritance imposes severe restrictions. Smalltalk-80 recently adopted a limited form of lattice inheritance (multiple superclasses), but we have insufficient experience to evaluate its worth. More complex inheritance mechanisms, such as the Flavors system from MIT Lisp, seem to destroy some of the explanatory simplicity which we consider among Smalltalk-80's greatest virtues. Another problem is that determining the actual procedure on the basis of the class of only one operand leads to awkward mechanisms for handling a few naturally operand-symmetric functions like arithmetic type-coercion. Type declarations, which we believe are a net hindrance in the exploratory environment in which Smalltalk-80 has grown up, may be important for both efficiency and documentation in other contexts.

Designing good frameworks is even trickier than designing good procedural interfaces in general. We are not entirely satisfied even in the area of the user interface, which is where we have devoted the greatest attention to the subject.

The effort required to produce a reasonably efficient implementation of the VM is substantial -- on the order of a man-year for an experienced designer. We are attacking this problem primarily by publishing as much of our (and others') implementation experience as possible, initially as a collection of papers by recent implementors of the system (Krasner, 1983). Even the best implementations to date are much less efficient than machine-oriented language systems such as Unix/C. Much of the inefficiency is due to the "late binding" or "total abstraction" philosophy of the Smalltalk-80 language. Patterson (1983) reports on some interesting work that shifts the balance between abstraction and efficiency for Smalltalk-80.

APPENDIX: Smalltalk-80 Language Syntax

At the level of individual procedures, Smalltalk-80 has a fairly conventional statement and expression syntax. We will summarize some parts of the syntax informally rather than giving actual syntax equations: "railroad diagrams" for the complete syntax appear in the back endpaper of (Goldberg & Robson, 1983).

Variables in Smalltalk-80 have the usual identifier syntax:

```
alpha  endOfFile  OrderedCollection
applePieNumber3
```

By convention, embedded capitalization separates words. The names of global (statically allocated, shared) variables begin with a capital letter; names of local (instance, procedure argument, or procedure temporary) variables begin with a small letter. There are a few pseudo-variables that refer to aspects of the current execution context: the most important one is **self**, which refers to the receiver of the currently executing invocation.

Smalltalk provides syntax for literal constant numbers, symbols, characters, strings, and arrays, as shown in these examples:

```
13  -5000000079  16rFFFF  3.14159
#aSymbol  $t  'This is a "quoted" string
#(1 2 3 (hello there) 'yes')
```

Expression syntax is somewhat less conventional, so we present it more formally. We assume the obvious definitions of the constructs empty, identifier, and literal-constant (defined above). We omit a few language constructs not used in the main body of the paper and not essential to understanding the language.

```
special-character ::= + | / | \ | * | ~ | < | > | = |
      @ | % | | | & | ? | !
variable-name ::= identifier
unary-operation ::= identifier
binary-operation ::= - | special-character-operation
special-character-operation ::=
      special-character |
      special-character-operation special-character
keyword ::= identifier:

primary ::= variable-name | literal | block |
      ( expression )
unary-object ::= primary | unary-expression
unary-expression ::= unary-object unary-operation
binary-object ::= unary-object | binary-expression
binary-expression ::=
      binary-object binary-operation unary-object
keyword-expression ::= binary-object |
      binary-object keyword-part
keyword-part ::= keyword binary-object |
      keyword-part keyword binary-object
compound-expression ::= unary-expression |
      binary-expression |
      keyword-expression
expression ::= variable-name ← expression |
      primary | compound-expression
```

As the above equations show, Smalltalk-80 admits three different kinds of operation names: unary postfix (named by identifiers), binary infix (named by sequences of special characters), and keyword (named by a sequence of keywords alternating with operands, where a keyword is an identifier followed by a colon). In each case, the interpretation is exactly the same: the operation is interpreted relative to the class of the receiver (the object preceding the operation name, or the first keyword.) Here are some examples of compound-expressions:

```
someCollection first
aPoint x + aPoint y
target at: j put: (source at: j + delta)
```

Statement and procedure body syntax are more conventional:

```
statements ::= ↑ expression | expression |
      expression . statements
formal-name ::= variable-name
block ::= [ block-formals block-body ]
block-formals ::= block-formal-names | | empty
block-formal-names ::= : formal-name |
      block-formal-names : formal-name
block-body ::= statements | empty
temporaries ::= | temporary-variable-names |
temporary-variable-names ::= empty |
      temporary-variable-names variable-name
formal-pattern ::= unary-operation |
      binary-operation formal-name |
      keyword-pattern
keyword-pattern ::= keyword formal-name |
      keyword-pattern keyword formal-name
procedure ::= formal-pattern |
      formal-pattern statements |
      formal-pattern temporaries statements
```

The ↑ notation means return a value from the current procedure.

Blocks are the Smalltalk-80 equivalent of the Lisp FUNARG or Algol 60 call by name. A block is a piece of code, possibly with formal parameters, which will be executed at a later time by invoking the operation **value** (if parameterless) or **value: param1 ... value: paramN** if it takes parameters. Like a FUNARG or by-name parameter, a block shares (by reference) the dynamic environment that existed at the time the block was created; like a full FUNARG, but unlike a by-name parameter, a block can be passed "upward" as well as "downward". This often allows Smalltalk-80 programmers to create non-hierarchical control structures without directly accessing the objects that implement control in the system (although the latter approach is also available.)

Note that the formal-pattern of a procedure looks exactly like a unary, binary, or keyword expression without a receiver.

REFERENCES

Goldberg, A., et al. 12 articles on the Smalltalk-80 language and system. Byte magazine, August 1981.

Goldberg, A. & Robson, D. Smalltalk-80: The Language and its Implementation. Reading, MA: Addison-Wesley, 1983.

Krasner, G. (Ed.) Smalltalk-80: Bits of History, Words of Advice. Reading, MA: Addison-Wesley, 1983 (forthcoming).

Patterson, David A. (Ed.) Proceedings of CS 292R: Smalltalk on a RISC, Architectural Investigations. Berkeley, CA: Computer Science Division, University of California, April 1983.