Rosetta Smalltalk:
A Conversational, Extensible Microcomputer Language

Scott K. Warren
Dennis Abbe

Rosetta
5925 Kirby, Suite 215
Houston, Texas 77005

ABSTRACT

Rosetta Smalltalk is a personal information handling environment for low-cost microcomputers based on the work of the Learning Research Group at Xerox PARC. Our prototype runs on two different Z-80 based personal computers. The major goals of the system are to support a lively interactive style of working and to provide an open-ended medium in which personalized tools may easily be constructed. Rather than write monolithic programs, the user extends the language with new objects and syntax. He then solves his problems by interacting with his extensions at the keyboard. Multiple independent CRT windows permit several partially completed interactions to be displayed at once.

All facilities in Rosetta Smalltalk are represented by *objects*, which are instances of Simula-like classes. Objects are not operated on directly, but are sent *messages* requesting them to perform actions and return replies. The language is extended by creating new classes and by adding new messages to existing classes.

Key Words and Phrases: abstract data types, conversational computing, extensibility, hypertext, message sending, modularity, object oriented programming, personal computing, windows.

CR Categories: 4.0, 4.20, 4.22, 4.34

INTRODUCTION

Rosetta Smalltalk is a conversational, extensible environment for doing personal information handling on today's low-cost microcomputers. In many ways it represents a radical departure from the BASIC-dominated style of computing presently in use on these machines. The major goals of the system are to support a lively interactive style of working and to provide an open-ended medium in which personalized tools may easily be constructed.

Rosetta Smalltalk is based on the very successful work of the Learning Research Group at Xerox's Palo Alto Research Center [4]. The original Smalltalk was created by Alan Kay and the LRG to serve as the communication medium for the Dynabook, a personal computer the size of a notebook "with the power to handle virtually all of its owner's information-related needs". Kay's Dynabook would be able to store thousands of pages of text, display images with resolution surpassing newsprint, and perform real-time audio synthesis with similarly high fidelity. Although the envisioned Dynabook does not yet exist, Smalltalk has been running on minicomputers ("interim Dynabooks") at PARC since 1972. Hundreds of visitors at PARC have learned Smalltalk; programmers and nonprogrammers alike have found the language both friendly and powerful. Children, animators, musicians, secretaries, and administrators have used Smalltalk to build their own personalized tools with relative ease.

Rosetta Smalltalk is an attempt to provide this kind of computing on today's low-cost microcomputers. Our prototype implementation currently runs on two different Z-80 based personal computers. Although less ambitious than the Dynabook, it demonstrates that such a system is viable on these machines. While we acknowledge a tremendous debt to the LRG, our language is not an implementation of either Smalltalk-72 or Smalltalk-76 as developed at Xerox PARC. Rosetta Smalltalk differs significantly from published descriptions of Xerox's languages [3,5], but since we have no detailed knowledge of those languages we do not discuss the differences here. In the remainder of the paper, "Smalltalk" should be understood to mean "Rosetta Smalltalk", though many of our statements may apply to Xerox's languages as well.

This paper is organized as follows. In Section 1 we give an overview of the Rosetta Smalltalk system. Section 2 describes the use of multiple CRT windows for interactive computing. Section 3 presents a more detailed picture of the Rosetta Smalltalk language, and Section 4 contains a brief description of the basic building blocks provided by the system. We conclude by discussing the performance and limitations of our prototype.

1. SYSTEM OVERVIEW

Imagine a personal computer that could store, retrieve, and edit almost any information that its owner was interested in: notes, drafts of papers, phone lists, structured files of data, pictures, simulations, music, and so on. Suppose it could be used as an interactive desk calculator for any problem domain, permitting its owner to deal directly with the items he is interested in such as notes, timbres, and pitches in music, equations and substitutions in mathematics, or paints, cels, and frames in animation. How could the user, not a computer specialist, cope with the immense variety of data formats, language rules, and processing conventions found among so many diverse application packages? How could he sufficiently understand the intricacies of all of this pre-written software to be able to customize it to his own needs, or to combine separate facilities to accomplish a particular task? And how could he add new facilities of his own, without a large programming effort?

Our answer to these questions, inspired by the work of the LRG, is to provide an open-ended conversational medium in which many special-purpose tools can be embedded, then accessed via a single uniform notation. The goal of Rosetta Smalltalk is to be a personalized information portfolio and application-specific desk calculator. The system provides the user with an APL-like workspace which contains all his data and programs and which is retained from session to session. Program text is a kind of data and can be manipulated as naturally as numbers and strings. Such an environment has been called *residential* [9] since everything of interest to the user resides within the system itself rather than being stored and edited by some external facility. The user is given highly interactive access to his workspace contents, with rich visual feedback. The underlying language of the system treats all entities in the same way, and is easily customized with new syntax and new kinds of objects for a particular area of interest. The system encourages a tool-building paradigm of problem solving. Tools are built by extending the base language rather than by writing monolithic programs, and are easily combined within the existing language framework.

In the following paragraphs we discuss in more detail the features of Rosetta Smalltalk that make it an open-ended problem solving medium. These features are categorized as follows: a rich interactive style, a fully conversational language, a single uniform notation for all operations, syntactic and semantic extensibility, and the modularity to permit building general tools.

Interactive. One of the major goals of Rosetta Smalltalk is to provide a lively medium for spontaneous problem solving. While the injunction "Think first, program later" is good advice when engineering a software product, it amounts to a strait jacket for the conversational user. Working in an interactive environment is more like sculpting in clay than building from blueprints. It is not uncommon to make false starts, frequently interrupting one activity in order to do something else first. In Rosetta Smalltalk multiple independent CRT windows [10] make these shifts of attention easy.

In many ways windows behave like pieces of paper on a desk. We can move them around on the screen; if two windows overlap, one will be partially hidden behind the other, but its contents are unaffected. When we move the window in front, the window behind it is instantly redisplayed. Using windows we can keep more information visible at once than if the whole screen were dedicated to imitating a single hard-copy device. This visual richness augments our short-term memory, helping us do several things at once without losing our place. Figures 1 to 3 show an admittedly contrived example in which we type a command in one window and view its output, a histogram, in another. Our next command causes a third window to display an error message, and we use a fourth to edit the code in error. If the screen is too small to display all of this at once, we can rearrange the windows as necessary, moving an interesting window where it can be seen or pushing others out of the way.

Conversational. The standard way of communicating with items in the workspace is by typing in commands for immediate execution. Rosetta Smalltalk makes no distinction between program text and commands from the keyboard, and vice versa. Input may be as simple as 2+2 or may contain multiple statements including control structures, so there is no need to write a program to try something out. Most work is done incrementally by typing in commands and immediately observing their effect, rather than by first writing and then running a long program. Solving problems in this way is faster and more natural than in the conventional "edit-run-debug" programming cycle, because each command we type performs some part of our actual task and gives us immediate feedback. The fully conversational aspect of Smalltalk simplfies testing new code and allows Smalltalk to be used as its own debugging language.

One language level. The Smalltalk language is based on the single notion of *objects* communicating by sending and responding to *messages*. Every entity in Smalltalk is represented as an object, from numbers and strings to control structures and arbitrary facilities defined by the user. The only operation in Smalltalk is to send an object a message requesting it to perform some action and possibly send back a reply. Objects are grouped into *classes* which describe their representation, the messages they can receive, and the methods they use to respond.

These ideas are detailed in Section 3; it suffices for now to note that the same notation for sending messages to objects can add two numbers, turn on a peripheral, rename a file, or invoke an

application package. Rosetta Smalltalk thus serves as a command language, a programming language, a debugging language, and host to any number of special purpose applications. This uniformity greatly simplifies the user's view of the system, and reduces the system's size by eliminating the need for several distinct language processors. The user also has an unusual amount of power available at the "command" and "application control" levels, since Smalltalk variables and control structures may be used there as well as in programs.

Extensible. The Rosetta Smalltalk user can easily customize his system by defining new classes of objects or adding new messages to existing classes. The necessary programming is done conversationally, testing each definition as it is entered. The user then solves his problems at the keyboard by directly executing Smalltalk commands. His extensions customize the language so that variables can hold things like musical scores, payroll records, or circuit diagrams, and his commands do real work like playing music, calculating payroll deductions, or simulating waveforms. In this way Smalltalk becomes a high-level programmable desk calculator tailored to his particular application.

Extensibility permits new facilities to be used as if they were built in. The user interacts with new kinds of objects through the same notation for sending messages with which he is already familiar. When the user adds an extension to his workspace, such as a set of objects for composing and playing music, he gets more than he would from a monolithic program performing the same functions. He has not just a music program but a music language. As an extension of Smalltalk, this language contains powerful features for programming as well as for performing music.

Modularity and building general tools. Rosetta Smalltalk encourages the construction of open-ended tools rather than fixed solutions to a problem. The notion of objects sending messages provides a uniform way of accessing extensions to the system, and the class mechanism permits extensions to be self-contained and thus suitable for loading into any workspace. As a result separately written tools may be combined with relative ease. For example, the music extension mentioned above could be combined with an extension for statistical analysis to permit the interactive search for patterns in a set of scores. A third extension for drawing histograms could be added so that such patterns could be viewed graphically. There is no need to venture outside of Smalltalk into the realm of operating systems to make the connection between applications. Smalltalk thus supports the same "software tools" approach to software development practiced on the UNIX† system [6], but without the distinction between command language and programming language. We see this toolkit approach to problem solving as programming in a very high level language specialized for the task at hand [7].

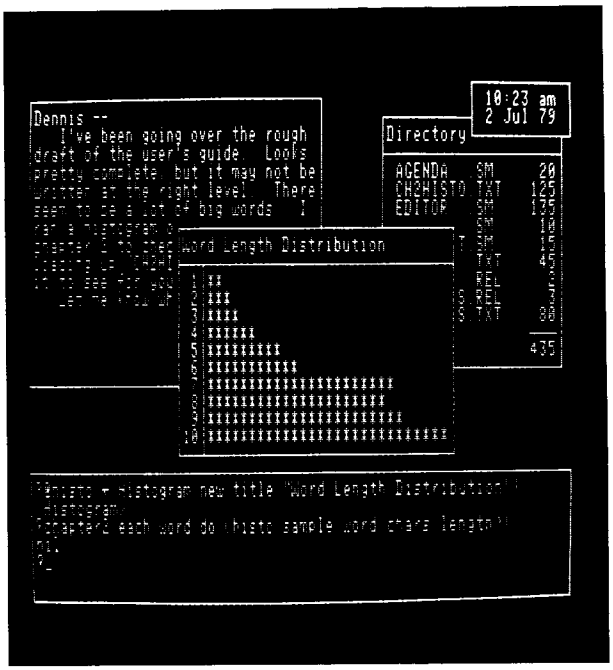

Figure 1. Multiple independent CRT windows.

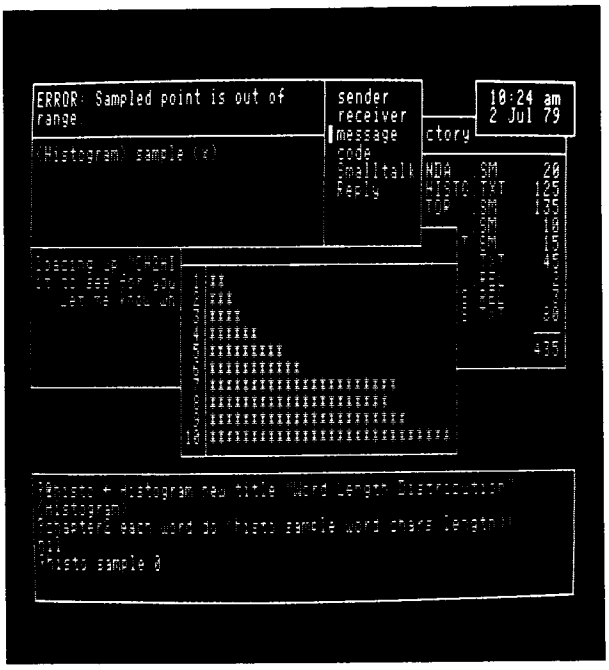

Figure 2. A diagnostic window.

† UNIX is a trademark of Bell Laboratories.

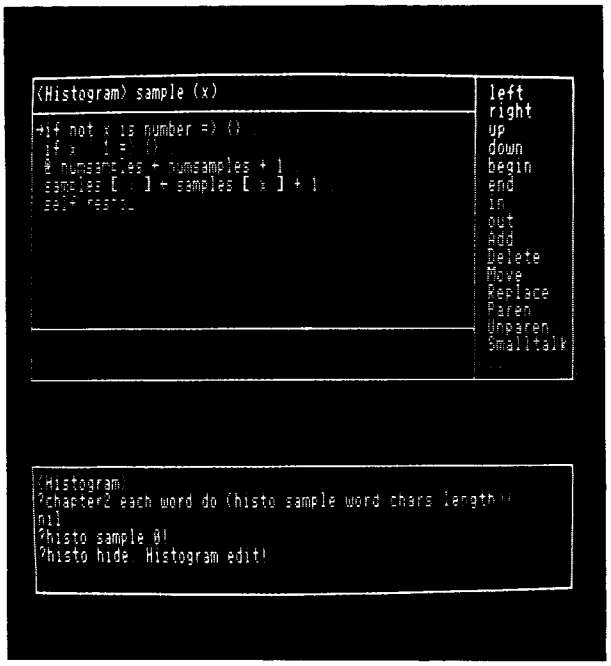

Figure 3. An editing window.

2. INTERACTING THROUGH WINDOWS

We have already seen that many windows may be displayed at once, much like sheets of paper on a desk. Each window may represent a different activity or view into the workspace. We turn our attention from one activity to another by pointing at the window of interest with the cursor; the window we select can then receive our keyboard input. For example, while editing a document in one window we might use a second window to consult a reference. Different styles of interaction may also be associated with different windows. One window might be used for dialog with Smalltalk, another may represent a menu of key-selectable commands, and another may simply display some text. Three styles of window interaction are part of the basic Rosetta Smalltalk system: dialog windows, diagnostic windows, and editing windows. Other kinds of windows may be defined using the basic *Window* class discussed in Section 4.

Dialog windows. A dialog window is used for typing Smalltalk code for immediate evaluation. The user's keystrokes and the result of each evaluation are printed in the window. A dialog window is thus the standard way of interacting with objects in the workspace, providing the "desk calculator" mode of operation common to most conversational systems. The bottom window in Figure 1 is a dialog window.

Each time a dialog window is ready for input it invokes the system scanner named *read*. *read* prompts with a "?" followed by a cursor. Input is gathered up until a special key called DOIT is pressed; this key is chosen as convenient for each keyboard, but always echoes as "!". Before he presses DOIT the user can edit his input by pressing keys to delete the previous character, the entire current line, or all of the input typed so far. When DOIT is finally pressed the input is evaluated and the result printed. By convention all objects print themselves in the window named *disp*. Each dialog window evaluates its inputs in a context in which *disp* may be temporarily re-bound to another window to send output elsewhere.

It is often convenient to use more than one dialog window. A running program, for example, may prompt the user for input. He can then open a new dialog window, perform some calculations, and finally reply to the waiting program with his calculated value.

Diagnostic windows. When an error is detected during Smalltalk evaluation, a diagnostic window will appear with a brief statement of the complaint. In this window the user can examine the context of the error. The message receiver, the message it received, and the Smalltalk code it was running can be displayed upon request. The user can engage in Smalltalk dialog in the context of the error, for instance to print or modify variables in the local name scope. The full power of Smalltalk is available as a debugging tool, making a special debugging language unnecessary. It is also possible to move up and down in the chain of contexts that led to the error, inspecting each one in turn. After examining and perhaps modifying the context of the error, the user can either terminate the suspended execution or resume it in a context prior to the one in which the error occurred. In the latter case he may supply a value to be used as the result of the suspended context. When he is done the diagnostic window disappears; any windows it obstructed become visible again. A diagnostic window appears in the top left corner of Figure 2.

Like most other Smalltalk system facilities, the error machinery is easily accessible to the user. By evaluating something like

error "This is my complaint"

any Smalltalk program can open up a diagnostic window.

Editing windows. Smalltalk programs are typically edited with a hypertext [2] editor which uses windows for displaying, entering, and pointing to program text. Figure 3 shows the screen layout after this editor has been invoked. The window at the top describes what is being edited. The largest rectangle is the text window, in which the current portion of program text is shown. The window at the right is a menu of available editing commands. Selections from this menu are made by pressing single keys corresponding to the first character of a command name. Selecting the "..." command brings a new menu of additional commands into the window. The bottom window is used to enter lengthy pieces of text such as insertions. The implementation of the editor is simplfied by the fact that each of these windows may be scrolled, cleared, and so on, independently of the rest.

The editor knows about the structure of Smalltalk programs and uses this knowledge to format the displayed code attractively and to allow easy selection of substructures for examination. The Smalltalk code is always shown neatly indented, with each statement starting on a new line. Whenever the text is altered it is immediately reformatted. Only

the top level of the code is displayed in the text window; parenthesized subexpressions are simply shown as "{}". The *in* command descends into one of these subexpressions to see its top level. The code in the window may be altered by selecting other commands from the menu, with the current version of the text always visible. The *out* command returns to the surrounding level.

To illustrate the use of the editor, suppose we want to insert some new text into the current program level. First we position the editor's cursor to where we want the text to go by using some combination of the *left*, *right*, *up*, *down*, *begin*, and *end* commands. Next we press *a* for *add*. Immediately the menu goes blank to indicate that no selection from it can be made until our insertion is complete. Simultaneously the prompt "*add?*" appears in the bottom window, followed by a typing cursor. We are now talking to the same *read* object used by dialog windows. After typing in the new text, we press DOIT and the text window immediately shows the result. At the same time, the bottom window clears and the menu reappears.

3. THE ROSETTA SMALLTALK LANGUAGE

The Smalltalk language is based on a metaphor of intelligent objects which communicate by sending and responding to messages. An object cannot be operated on directly, but can only be sent requests to perform actions and return replies. Every object is a member of some class which describes its representation, the messages it can receive, and the methods it uses to answer them. Smalltalk is easily extended with new classes of objects and new syntax for messages.

Message sending. A message is sent by writing the message receiver followed by the message itself. For example, to move the window *disp* to a different place on the screen we can say

disp move to 10 2

In this expression *disp* is the message receiver and "*move to 10 2*" is the message being sent. We represent the syntax of this message by the *message pattern*

... move to (newl) (newc)

The "..." indicates the message receiver, and the parenthesized variables indicate evaluated parameter slots. An object may also receive message parameters unevaluated. For instance, the control structure object *do* responds to a message of the form

... (n) (@code)

The symbol "@" indicates that *code* is received by *do* unevaluated. When we send

*do 3*4 (disp hide show)*

n is *12* and *code* is the list *(disp hide show)*. *do* answers this message by evaluating *code* 12 times, causing *disp* to repeatedly disappear and reappear.

The object *x* is returned as the reply to a message by evaluating

reply x

When a message is sent to an object just to achieve an effect and not to compute a result, *reply* may be omitted. In this case the message receiver itself is replied by default. This reply permits several messages to the same object to be cascaded together, as *...hide* and *...show* were in the example above. Rosetta Smalltalk uses periods to separate message sendings when it is not intended for the reply of one message to become the receiver of the next. Thus the expression

*do 3*4 (disp hide. disp show)*

has the same effect as the example above.

Smalltalk evaluates an expression by first obtaining the message receiver, then matching message patterns against the following tokens. Only those patterns belonging to the receiver's class are eligible to be matched. Matching proceeds from left to right, interleaved with evaluation of subexpressions corresponding to parameter slots. Smalltalk matches a specific token in preference to a parameter slot, and always takes the longest possible match. The empty message will be matched if the receiver can answer it and no longer pattern is found. Once a unique pattern is matched Smalltalk sends the message, setting up a new context for the object to respond in.

Context of a message sending. Every Smalltalk object owns some private data that can be directly accessed only by itself. These *instance variables* are property names common to all instances of a class, for which each instance has particular values. For example, a window object's size is described by two variables: *h*, its height in lines, and *w*, its width in columns. Each window has its own values for these variables and refers to them whenever it is asked to show on the screen. We cannot change these values directly, but a window will do so if asked:

disp grow to 10 30

Sending this message has the visible effect of setting *disp*'s size to 10 lines of 30 columns each. To accomplish this, *disp* has to hide itself, adjust its text buffer to 300 characters, update its *h* and *w* values, and show itself again. Because unauthorized access to instance variables is prohibited, the window is able to ensure that its buffer size and visible appearance remain consistent with its height and width.

Objects answer their messages by running pieces of Smalltalk code called *methods*. A method refers directly to the object's private data by mentioning its instance variable names. The method can also mention the special name *self* to refer to the object receiving the message. Objects often send themselves messages this way. For instance, the method by which windows respond to the "*grow to*" message could be

```
... grow to (newh) (neww)  =>
    ( self hide.
      @text ← String new newh*neww.
      @h ← newh.  @w ← neww.
      self show )
```

An object may reveal as much or as little of its representation as it desires by the messages it chooses to answer. It can grant full access to its representation by answering

... 's (@code) => (reply code eval)

When this message is sent, *code* is an unevaluated piece of Smalltalk code, and the object replies with the result of evaluating that code in its private context. If this message is defined for windows, sending

disp's h

will reply with the height of *disp*. This kind of message is helpful when debugging, but must be used with care since the object's assumptions about its own data can be disrupted. For instance,

disp's (@h ← h+2)

increases *disp*'s height without making a corresponding adjustment in its text buffer, and will cause an error the next time *disp* is asked to show.

There are actually three sets of variables in the local context of a message sending: temporary variables, instance variables, and class variables. All three kinds may be accessed directly by a method. Temporary variables are created when a message is sent and disappear as soon as a reply is made. These variables may be used as scratchpad storage while the method is running. Certain temporaries are initialized with values from the message and thus serve as formal parameters; the variables *newh* and *neww* in the "*grow to*" message are examples of this. Instance variables are names for the data private to each instance of a class, as discussed above. Their values persist between message sendings as long as the object exists. Class variables are accessible to all instances of a class. They usually hold data for communication between instances or for class-wide bookkeeping. Their values are stored within the class itself and persist as long as the class exists. Class variables play the role often filled by global variables in other languages, but in a more secure and modular way. The shared information held in class variables is accessible only to members of the class and not to the world at large.

When a name is mentioned that is not one of the three kinds of locals, Smalltalk looks for it in the dynamically enclosing context -- that is, the one from which the current message was sent. The search ends in the user's workspace. A common problem with dynamic name scoping is the accidental hiding of global variables when code is run inside a context that happens to use those names for another purpose. Rosetta Smalltalk does not suffer from this problem because all class-related data is accessible from the innermost context, including the class variables that would have been global in some other languages.

Classes. We group objects into classes so they can share the same representation, message patterns, and methods. The Smalltalk class mechanism is modelled after that of Simula 67 [1], but Smalltalk is unique in representing every facility as an instance of some class. A new class is a description of a kind of object, or data type, of which there may be many instances. A new class is thus a semantic extension to the Smalltalk world. Furthermore, the message patterns of a new class form a direct extension to the language syntax. By creating new classes the Smalltalk user creates objects modelling his own abstract ideas, and invents his own notation for using them as well.

Classes are a tool for extending a language in a modular way. The representation of an object is ordinarily concealed from outside the object, providing information hiding in the sense of Parnas [8]. The only operation that can be performed on an object is to send it a message requesting some action; how that action is carried out is of no concern to the sender and may be changed without affecting existing code. Moreover, this object-oriented style of programming collects related code into a central place, the class definition. For instance, details of how a class of objects should be printed are grouped with other details about the class rather than in some all-purpose print routine. This makes it easier to find all affected code when a change is made.

A new object is created by sending a *...new* message to the desired class. The object should respond to a message beginning with the special token *isnew* by initializing its instance variables appropriately. One cannot forget to initialize an object because the *isnew* token is automatically supplied by the system. Apart from this bit of synchronization, *isnew* messages are no different from other messages. For example, a new window must be told its initial size and location. To create a new window and name it *mywindow*, we say

@mywindow ← Window new 5 30 2 2

This creates a new window which immediately receives the message "*isnew 5 30 2 2*". The new window initializes its height and width to 5 lines of 30 columns and its screen location to line 2, column 2. Other instance variables are computed from the given information. For instance, *mywindow*'s text buffer is allocated to hold 150 characters.

Every object in Rosetta Smalltalk belongs to a class, and classes are no exception. Every class is an instance of the class named *Class*; this class has the unique property of being an instance of itself. To create a new class we send the *...new* message to *Class*:

@Stack ← Class new

Of course, the new class must be given variable dictionaries, message patterns, and methods for it to be useful. This could be done by sending appropriate messages to *Stack*, though we would ordinarily invoke the built in hypertext editor.

One can also extend or modify the definitions of existing classes. This includes predefined classes of the Rosetta Smalltalk system as well as those created by the user. As a simple example, suppose we want windows to be able to flash themselves in order to attract our attention. We must define two things: the syntax of the message and the method used to answer it. Our new message syntax will be

... flash (n) times

The method for flashing will be to erase and redraw the window's frame the requested number of times. We can add this capability to class *Window* by evaluating

Window answer @(flash (n) times)
by @(do n (self unframe frame))

This is just a message to *Window*. The "@" tokens

indicate that the following parenthesized lists should be taken literally rather than evaluated. After adding the above message to class *Window* we can say

mywindow flash 20 times

and our window will blink its frame off and on 20 times. Note that when a new message is added to a class, all existing instances can immediately respond.

4. THE PREDEFINED OBJECTS

The message sending discipline of the previous section is only a language framework. This skeleton must be augmented with enough predefined objects to enable the construction of extensions. The basic Rosetta Smalltalk system provides fundamental programming language elements and some high-level building blocks to support interactive computing. This set of basic objects includes the primitive classes *Atom*, *Number*, *String*, and *List*; the objects *yes* and *no*; the control structure objects *if*, *do*, *for*, *repeat*, and *done*; the objects *Window*, *read*, *kb*, *File*, and *lp* for input and output; the workspace management objects *vars* and *erase*; and of course the class *Class*. In addition there is a hypertext editor for creating and modifying classes and other program text. Our summary of the basic system is rather informal; many of the predefined messages answered by these objects are omitted.

There are several messages which every object should be able to answer. Rosetta Smalltalk supplies default methods for answering these messages to every new class. These are:

... print	=>	print the title of the object's class in brackets
... is ?	=>	reply the object's class
... is (c)	=>	reply *yes* if the object is an instance of class *c*; otherwise reply *no*

Usually the default print method gets replaced by something more useful.

The Primitive Classes.

Under this heading we include the classes *Atom*, *Number*, *String*, and *List*. We use these objects for variables, arithmetic, and data storage. These classes use familiar notation for concepts found in other languages, such as arithmetic and assignment.

Atom. Atoms are LISP-like symbols used as variable names and syntactic tokens in messages. When an atom receives a message of the form *... ← (ob)* it will bind itself to the object *ob* in the current context. Binding occurs under the rules of dynamic name scoping discussed earlier. The message *...eval* sent to an atom replies with the object to which it is bound.

The atom spelled "@" is always bound to an object called *quote*, which receives a single unevaluated parameter and replies with that parameter. An object may thus be referred to literally in a message by preceeding it with *@*. For example, the result of evaluating *@x* is just the atom *x*. An "assignment statement" in Smalltalk hence looks like *@x ← 3*. After this assignment the atom *x* is bound to the number *3*.

Atoms also answer messages to print themselves, obtain their print names, and inquire whether one atom is the same as another.

Number. Numbers in our prototype implementation are provided only in the form of small integers. These numbers respond to the usual complement of arithmetic and relational messages. Examples of other messages are

97 chars	replies the string *"97"*
97 ascii	replies the string *"a"*
97 print	performs *disp ← 97 chars*

String. Strings are sequences of characters that respond to a rich set of string manipulation messages. For instance, if *s1* and *s2* are strings, then

s1 length	replies the number of characters in *s1*;
s1 + s2	replies the concatenation of *s1* and *s2*;
s1[k]	replies the *k*-th character;
s1[j to k]	replies the substring of *s1* from positions *j* to *k*;
s1 find first s2	replies the position of the leftmost occurrence of *s2* in *s1*

and so on. Strings may also be used as byte arrays which may be selectively updated. For example

s1[k] ← "a"	replaces the *k*-th character;
s1[j to k] ← s2	replaces a substring of *s1*.

List. Lists are arrays much like strings except that each position of a list can contain any object. Most of the string messages are also answered by lists; one may concatenate two lists, pick out an element or subsequence of elements, or replace elements of a list. Lists also have a method for iterating over their elements. The expression

l each x do (x print)

will print each element of the list *l*. Smalltalk also uses lists to represent programs. A list will respond to the message *...eval* by running itself as Smalltalk code. The Smalltalk interpreter is thus just another method of the class *List*.

Control Structure Objects.

Control structures in Smalltalk are implemented by objects which answer messages containing unevaluated code as parameters. Users can easily define new control structures of their own. We have already seen an example of how *do* works. Other control structure objects are briefly discussed below.

if. The object *if* implements the McCarthy conditional, as found in LISP. The syntax of *if*'s message is

... (expr) => (@yespart)

The expression *expr* should evaluate to either *yes* or *no*. If *expr* is *yes*, the *yespart* code is evaluated and the entire list in which the *if* occurs is exited. For example,

```
(if i < j => (i print). j print)
```

will print the smaller of i and j. If the value of *expr* is *no*, *if* does nothing and replies immediately with itself. This permits a series of else-if tests to be cascaded together, as in

```
if x < node val => (left)
   x > node val => (right)
   x = node val => (found)
```

for. The object *for* implements a for-loop control structure. It answers a message of the form

```
... (@var) ← (lb) to (ub) do (@code)
```

For example, the expression

```
for k ← 1 to n do (k print. cr)
```

will print the first n integers on separate lines.

repeat. The object *repeat* implements an infinite-loop control structure. Control leaves the loop when either the user interrupts, or the object *done* is invoked. For instance,

```
repeat (read eval print. cr)
```

is a typical Smalltalk dialog loop.

done. The object *done* performs single level exits from *for*, *do*, and *repeat* loops, and from the list iteration method for the "...*each*" message. A use of *done* may optionally exit a loop with a reply, which becomes the reply of the loop itself. Thus the loop

```
repeat (if i < j => (@i ← i + 1).
        done with "ok")
```

will reply with the string *"ok"* when i becomes greater than or equal to j.

Input and Output Objects.

Smalltalk input is done primarily with the objects *kb* and *read*. *kb* will wait for a single keystroke and reply with the ASCII code of the key depressed. The object *read* is used to input Smalltalk tokens or untokenized lines of characters. A token is any instance of one of the classes *Atom*, *Number*, *String*, or *List*. *read* can thus read anything from a single number to an entire Smalltalk program. By default, *read* reads from the keyboard and echoes in the window named *disp*; its reply is a list of the tokens read. The following messages to *read* are also defined:

read in w	echoes input in window w;
read of ob	*ob* can be a string or any object that replies to the message ...*next* with a character; result is as if the characters were typed at the keyboard but no echoing occurs;
read line	replies a string of the characters typed, which are echoed in *disp*;
read line in w	like *read line* but echoes in window w.

Window. Windows display themselves as rectangular areas on the screen, optionally bordered by a frame. Each window has its own size, screen location, text buffer, cursor, and status bits. Each window may be written into, scrolled, cleared, moved, changed in size, and so on independently of the rest of the screen. Examples of some messages to windows include:

w ← "some text"	writes the text in w at the current position of its write cursor;
w clear	fills the text buffer with blanks;
w unframe	erases w's frame;
w at 2 1	sets w's cursor to its line 2 and column 1;
w hide	erases w from the screen; previously obstructed parts of other windows are brought into view;
w show	displays w on the screen;
w grow to 10 30	gives w 10 lines of 30 characters each;
w move to 15 1	moves w to line 15, column 1 of the screen;
w scroll	scrolls the text in w up by one line.

File. The class *File* provides sequential and random access to secondary storage. The contents of a file are a sequence of bytes. Examples of file messages include:

f open "x" output	opens f for output with filename *"x"*;
f ← "some text"	writes the text to f;
f seek n	sets f's position to its n-th byte;
f next	reads the next byte from a file open for input;
f end	replies *yes* if at end of file;
f close	closes the file.

lp. The object *lp* is used to write to a hardcopy device. *lp* answers some of the same messages as do windows; in particular, *lp ← "print it"* prints the text *"print it"* on the line printer.

Workspace management.

As mentioned earlier, all atom bindings not local to a particular message sending context are made in the user's workspace. A list containing all the atoms so bound can be obtained from the object named *vars*. The object *erase* will remove variables from the workspace, e.g. *erase (x y z)*.

The Class *Class*.

New classes are created by sending the message *...new* to the class named *Class*. The messages and methods of a class may be changed by using the messages

... *answer (message) by (method)*

... *forget (message)*

The method for a particular message may be obtained by sending

... *method for (message)*

The *...messages* message to a class will reply with a list of all its message patterns. Messages for changing the other parts of a class also exist. A class can be edited using the hypertext editor by sending it the message *...edit*.

CONCLUSION

A prototype implementation of Rosetta Smalltalk currently runs on two different Z80 based personal computers. Our prototype was intended as a feasibility demonstration and design tool rather than a finished product; its implementation was deliberately kept simple even where it was clear that special-case optimizations would be needed for adequate performance. Despite this the system's performance is encouraging. The basic system described in this paper occupies about 16K bytes, consisting of 12K of machine code and 4K of predefined objects. This does not include the workspace. No thorough benchmarking has been done, but an in-memory bubble sort runs seven times slower than the same algorithm in interpreted BASIC on the same machine. We believe we have learned enough from our prototype to design a new version of the system with performance comparable to BASIC.

Our experience in using Rosetta Smalltalk, though limited, has also been encouraging. We have created a number of toy extensions ranging from menu-driven drawing tools to a simple discrete-event simulation system. In addition, we built several versions of the hypertext editor in Smalltalk before manually translating it into assembly language. Only a few persons besides ourselves have used the system, but they have reacted enthusiastically. We are aware of a number of limitations in our present system; some are the result of deliberate design tradeoffs, some are due to the simple implementation of the prototype, and some are imposed by our target machines. These limitations include:

No declarations: this is a simplification for the novice and is traditional in highly interactive languages, but the drawbacks are lost security and the necessity for interpretation; we plan to add an incremental declaration facility in a later version.

Dynamic parsing: in the absence of declarations, message pattern recognition must be interleaved with evaluation; message patterns allow a friendly, readable syntax and easy syntactic extensibility, but can be confusing if deeply nested or if one is not familiar with the classes of intended message receivers; also, code may be parsed in an unexpected way if the class of a message receiver is not what was expected.

No subclass capability: the use of subclasses makes the effort involved in defining a new class less, but it is difficult to provide in our prototype's implementation of parsing.

No coroutines: as with subclasses, this desirable feature was sacrificed in favor of simplicity in the prototype.

Low bandwidth: of course our target machines do not have the high-resolution graphics or the computational resources of PARC's interim Dynabooks; still, our system is qualitatively similar to PARC's.

No applications software: the most serious limitation of our present system is the lack of the application extensions that would make Rosetta Smalltalk a full-fledged personal information handling system.

We have described Rosetta Smalltalk as a system offering a rich interactive style, a fully conversational language, a single uniform notation for all operations, syntactic and semantic extensibility, and the modularity to permit building general tools. But perhaps its most important characteristic for personal computing is its friendliness. The notion of communicating with intelligent objects has an anthropomorphic flavor which puts abstract data types in a lively, concrete setting. The idea of classes is based on the familiar idea of grouping together objects which share common properties. The Rosetta Smalltalk syntax has a pleasant, readable appearance because syntactic extensibility allows a suggestive notation to be chosen for every operation. Our experience and that of Xerox's Learning Research Group show that programmers and nonprogrammers alike readily accept the metaphor of active objects communicating by sending messages, and can effectively use the powerful tools for abstraction and extensibility that Smalltalk provides.

REFERENCES

1. Birtwistle, G., Dahl, O.-J., Myhrhaug, B., Nygaard, K. *Simula Begin*. Auerbach, Philadelphia, Pa., 1973.

2. Carmody, S., Gross, W., Nelson, T.H., Rice, D., Van Dam, A. A hypertext editing system for the S/360. Faiman, M. and J. Nievergelt (eds.), *Pertinent Concepts in Computer Graphics*. University of Illinois, 1969, 291-330.

3. Goldberg, A. and Kay, A. (eds.). *Smalltalk-72 Instruction Manual*. SSL76-6, Xerox PARC, Palo Alto, Ca., 1976.

4. Goldberg, A. and Kay, A. Personal dynamic media. *IEEE Computer* 10, 3 (March 1977), 31-41.

5. Ingalls, D.H.H. The Smalltalk-76 programming system design and implementation. *Proc. Fifth Annual Symp. on Principles of Programming Languages* (ACM) (Jan. 1977), 9-15.

6. Kernighan, B.W. and Mashey, J.R. The UNIX programming environment. *Software - Practice and Experience* 9, 1 (Jan. 1979), 1-15.

7. Liskov, B. and Zilles, S. Programming with abstract data types. *Proc. Symp. on Very High Level Languages*, SIGPLAN Notices (ACM) 9, 4 (April 1974), 50-59.

8. Parnas, D. On the criteria to be used in decomposing systems into modules. *Comm. ACM* 15, 12 (Dec. 1972), 1053-1058.

9. Sandewall, E. Programming in an interactive environment: the "LISP" experience. *Computing Surveys* 10, 1 (March 1978), 35-71.

10. Teitelman, W. A display oriented programmer's assistant. *5th International Joint Conference on Artificial Intelligence* (1977), 905-915.