SW 2 - An Object-based Programming Environment

Mark R. Laff
Brent Hailpern

IBM Thomas J. Watson Research Center
Yorktown Heights, New York 10598

Abstract

Programming systems traditionally deal with only a few different types of data objects. Operating-system command languages, for example, are concerned with files and programs. Typical programming languages deal with computer-related objects such as integers, strings, arrays, and records. This is in sharp contrast to the variety of real-world objects that people reason about. Smallworld is a programming environment in which the real world is represented by *objects* that have *properties*. A property represents a fact about the corresponding real-world entity. Thus Smallworld actions (programs), which operate on objects specified in this simple but general way, are "smart": they consider all of the relevant facts concerning (that is, all of the properties of) the objects they manipulate.

Smallworld was strongly influenced by the design of Smalltalk, especially in the organization of objects into classes and superclasses. The two languages differ (1) in their treatment of the difference between classes and objects that are not classes and (2) in their definition of methods that act on classes. Smallworld minimizes the differences between classes and non-class objects, resulting in a simpler and more consistent system. Where Smalltalk is a programming language using a pure object-oriented paradigm and dependent on a powerful graphical interface, Smallworld is a shell language that runs on conventional terminals and allows multiple program paradigms where appropriate.

1.0 Introduction

Smallworld is a programming environment where the information to be stored and manipulated is structured in terms of objects. It is not a programming language, but rather a system where application programs and other uses of a computer can be developed. The basic entity of Smallworld is the *object*. Each "thing" that Smallworld manipulates is an object: programs, documents, memos, the operating system, phone books, and so on. Objects have *properties*. For example, a program is stored in one or more files; it is compiled with a particular compiler and a set of compiler options; its listing is printed on a specific printer. Therefore, a program object could have properties such as a list of file names, a compiler, a list of compiler options, and a printer name. Similarly, a person in a phone book could have a name, an address, a phone number, a user id, and a birthday. Smallworld allows the user to remember all of the facts about an object in one place, instead of writing them down on scraps of paper.

Smallworld is more than just a database; it provides actions for manipulating objects. It is easier to implement sophisticated actions in Smallworld than in a typical computer environment, because all of the facts concerning the objects to be manipulated are stored with those objects. Consider, for example, the problem of compiling a program. A COMPILE action (when dealing with a program) could inspect the SOURCE-FILE and COMPILER-NAME to determine which file to compile, which compiler to invoke.

Smallworld actions have different semantics when operating on different objects. A PRINT action applied to a program might print the program listing, while PRINT applied to a document might call a document formatter to do the printing; the definition of the PRINT action that applies to program objects is separate from the definition of PRINT that applies to document objects. Each such definition is called a *method*. Thus, to define PRINT for some new class of objects, the programmer writes a new method to print objects of the new class. There are, of course, techniques for allowing different classes to share methods. This is in contrast to the classical approach of writing a large print program, which must know how to print objects of every type, using every kind of printer; modifying such programs to accept new types of objects is difficult or impossible.

Smallworld is intended to make the user/operating-system interface more friendly. It is currently implemented under VM/SP running CMS/SP, where it acts both as a program development tool and as an operating system shell.

The user need embrace the object-oriented structure only as far as is convenient. For example, data associated with an object can be stored as a property of that object or as a file in the underlying file system whose name is a property of the object. Similarly, actions can be defined directly as methods in Smallworld or as programs invoked by methods. In addition, methods can be programmed either in the object-oriented paradigm or in a conventional block-structured programming paradigm.

2.0 Constructs

2.1 Classes

Each Smallworld object belongs to some *class*, and each Smallworld class is an object. The class of an object is determined by its CLASS property. Because classes are themselves objects, the set of all Smallworld objects can be thought of as a tree; the edges being the CLASS relation, and the root being a distinguished object called CLASS. (CLASS is also a class.)

Suppose, for example, PRINT, SORT, and DECIDE are Pascal programs, belonging to the class MAIN-PROGRAM. The CLASS property of the object PRINT has the value "MAIN-PROGRAM", as do the CLASS properties of the objects SORT and DECIDE. The CLASS property of the object MAIN-PROGRAM has the value "CLASS", as does CLASS property of the object CLASS. A diagram of this class structure is shown in Figure 1.

Classes, being objects, can have properties other than the CLASS property. For example, they can have properties that serve as default values for objects in that class, such as the default printer for a kind of letter or memo. Similarly, they can have methods defining the actions that apply to all members of that class.

2.2 Methods

A method, which implements some action, applies either to one object or to a class of objects. A method that applies to a single object is called *simple*. A method that applies to a class of objects is called *inherited*; objects that do not have a method for a particular action inherit the method from their class. We assume the existence of a shell (or exec) language that can be used to specify methods; our CMS implementation uses REXX [2, 9, 10]. REXX is a shell language for CMS that has a PL/I-like syntax augmented with string processing and file manipulation commands.

A Smallworld method is executed in an environment with three string variables initialized: *verb*, *subject*, and *parameters*. The verb is the name of the action being requested. The subject is the name of the object being acted upon. The parameters represent the arguments being passed to the action.

The object database is implemented by GVX [13]. GVX is a simple object and property-list database, written in C, designed to provide static data structures to REXX. The method environment contains eight basic functions used to manipulate the object database. [1]

- CREATE(object-name,class-name) - creates a new member of the specified class with the given name.
- DESTROY(object-name) - removes the object from the database
- EXISTS(object-name) - returns *true* if the object exists and *false* otherwise.
- GETPROP(object-name, property-name) - returns the value of a property of an object.
- SETPROP(object-name, property-name, value) - sets the value of the property.
- ALLPROPS(object-name) - returns a list of names of all properties of an object.
- ALLOBJS(class-name) - returns a list of names of all objects belonging to a given class.
- EVAL(expression) - evaluates a Smallworld expression, described below, and returns the value.

A simple method is represented as a property of the object that it acts on. If object A has a method implementing action C (on object A), then A has a property called C.METHOD. The contents of the C.METHOD property is the text that defines C. The text of C.METHOD is interpreted as a REXX program, allowing manipulation of the database and use of local variables.

An inherited method is represented as a property of the class of objects that it acts on. If class B has a method implementing action C (for the members of class B) then B has a property called INHERITED:C.METHOD. The contents of the INHERITED:C.METHOD property is the text of the definition of C (also a REXX program).

Most methods are inherited, that is, they act on a entire class of objects. Examples include printing a memo and compiling a program: PRINT acts on objects of class MEMO and COMPILE acts on objects of class MAIN-PROGRAM. For those cases where one object must be treated specially, a simple method allows an object to "know" how to perform a particular action on itself. See Figure 2 for an illustration of inheritance of the PRINT method.

2.3 The Universe

There is a distinguished object in Smallworld called *UNIVERSE*. UNIVERSE is a class. One of the functions of UNIVERSE is to organize a special set of methods and objects. These universal methods serve as default definitions for actions; that is, if some object does not have the method for an action and the class of the object does not have the method for the action, then the UNIVERSE is consulted for a method. This is a special instance of Smallworld's class-superclass inheritance structure, which is described in more detail in Section 4.

The objects that belong to UNIVERSE represent concepts that do not fit the normal object/class structure of Smallworld. For example, actions that manipulate properties of objects, instead of the objects themselves, are specified by methods of the PROPERTY object, which belongs to the UNIVERSE class.

[1] More database functions are described later.

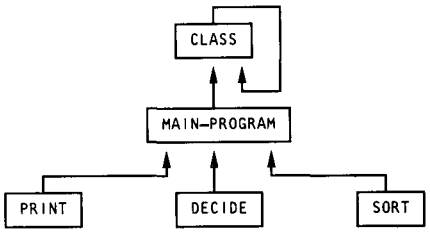

Figure 1. The class structure of MAIN-PROGRAM

2.4 Expressions

The user types expressions to the shell (in some ways similar to the conventional LISP read-eval-print user interface). Smallworld expressions begin with the name of an action, followed by the name of a subject, optionally followed by other parameters. An expression always evaluates to a result (a character string). The shell eval loop prints the result on the terminal. Expressions have the form

```
action-name subject-name parameters
```

and are evaluated according the following rules: [2]

1. Does the subject-name represent an object? If so, continue with step 2. If not, report that the subject does not exist.
2. Request the subject to perform the action on itself. That is, use the action-name.METHOD property of the subject, if it exists. If it can, then we are done. If not, continue with step 3.
3. Request the class of the subject to perform the action on the subject. That is, use the INHERITED:action-name.METHOD property of the subject's class, if it exists. If it can, then we are done. If not, continue with step 4.
4. Request UNIVERSE to perform the action on the subject. That is, use the INHERITED:action-name.METHOD property of UNIVERSE. If it can, then we are done. If not, report that the action is not defined for the subject.

2.5 More About Properties

In the last section, we saw how the method defining an action could rest with the subject, could be inherited from the class of the subject, or could be inherited from UNIVERSE. In the same way, the value of a property can be inherited. This allows default values to be defined for a class of objects, such as a default printer or compiler.

The GETPROP(S,P) function first checks object S for property P. If not found, GETPROP checks the class of S for INHERITED:P. If still not found, it checks UNIVERSE for INHERITED:P. Hence, inheritance permits overriding of default values.

3.0 The Base Environment

Smallworld includes several pre-defined classes, objects, and methods. These objects form a minimum starter set for using the system. The user will add classes, objects, and methods to this set to customize his environment. In addition, packages of objects are provided to perform advanced functions, such as manipulating the CMS environment.

These are the objects in the Smallworld base environment: CLASS (class CLASS), UNIVERSE (class CLASS), PROPERTY (class UNIVERSE), INHERITED-PROPERTY (class UNIVERSE), METHOD (class UNIVERSE), and INHERITED-METHOD (class UNIVERSE).

The inherited methods for the class UNIVERSE are as follows:

- CLASS? object-name - returns the name of the class to which the object belongs.
- FORGET object-name - deletes an object from the database.
- SHOW object-name - lists all of the property-value pairs for an object.
- COPY old-object new-object - creates a new object with the same properties as the old object.
- TO-FILE object-name - similar to show, but lists the property-value pairs in a file.

The inherited methods for the class CLASS are as follows:

- NEW class-name object-name - create a new object of a class.
- MEMBERS? class-name - list the names of all objects in a class.
- FORGET class-name - similar to the method of UNIVERSE, but checks that the class contains no objects before it is deleted.

The PROPERTY object represents all properties of all objects, so that methods can be defined that manipulate individual properties. Some of the methods defined for the PROPERTY object are as follows:

- SET PROPERTY property-name object-name value - create a property belonging to the indicated object with the specified value.
- SHOW PROPERTY property-name object-name - show the contents of the indicated property.
- FORGET PROPERTY property-name object-name - delete a property belonging to the indicated object.
- COPY PROPERTY old-property old-object new-property new-object - creates a new property, belonging to the indi-

[2] There are more rules than this. They are described later.

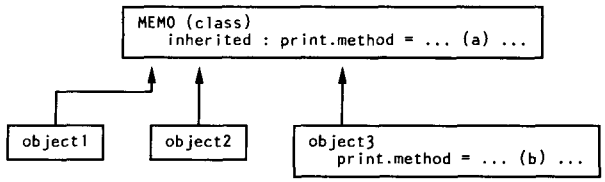

Figure 2. Inheritance of a PRINT method: The links are class links. The commands PRINT OBJECT1 and PRINT OBJECT2 use definition (a). The command PRINT OBJECT3 uses definition (b).

cated "new object". The contents of the old property are copied into the new property.

- GET PROPERTY property-name object-name - returns the contents of the indicated property. This would be used inside a method to get the value of a property, using the EVAL function.

The INHERITED-PROPERTY object represents all inherited properties of classes. The methods listed below automatically insert the "INHERITED:" prefix for the property names. It is perfectly valid to use the PROPERTY object and insert the prefix yourself. The methods defined for INHERITED-PROPERTY are as follows:

- SET INHERITED-PROPERTY property-name object-name value - create an inherited property belonging to the indicated object with the specified value.
- SHOW INHERITED-PROPERTY property-name object-name - show the contents of the indicated inherited-property.
- FORGET PROPERTY property-name object-name - delete an inherited-property belonging to the indicated object.

Methods defined on the METHOD object represent actions on methods. They automatically add the ".METHOD" suffix to the method name. When displayed, the contents of the methods are broken into lines so that the text looks like a program instead of a long string. The methods defined for METHOD are as follows:

- SET METHOD method-name object-name value - create a method belonging to the indicated object with the specified value.
- SHOW METHOD method-name object-name - show the contents of the indicated method.
- FORGET METHOD method-name object-name - delete a method belonging to the indicated object.
- COPY METHOD old-method old-object new-method new-object - creates a new method, belonging to the indicated "new object". The contents of the old method are copied into the new method.

Similarly there are methods for the object INHERITED-METHOD that manipulate inherited methods.

- SET INHERITED-METHOD method-name object-name value - create an inherited-method belonging to the indicated object with the specified value.
- SHOW INHERITED-METHOD method-name object-name - show the contents of the indicated inherited-method.
- FORGET INHERITED-METHOD method-name object-name - delete an inherited-method belonging to the indicated object.

4.0 Advanced Features

The features presented so far provide the minimal support necessary to customize the user's programming environment. A large environment, however, would become unwieldy with only these functions because (1) the syntax would become tiresome for frequently executed actions, (2) the names of objects and methods are long enough to be annoying to type, (3) it would be difficult to share methods between classes, (4) there would be no way to organize classes into classes of classes, and (5) it would be difficult to make packages of objects available to users, because there would only be one database.

The rest of this section describes the Smallworld features that solve these problems.

4.1 Hierarchical Organization

As described, so far, information in Smallworld is organized into objects. The objects are collected into classes. The evaluation of an action on an object searches for a method first in the object, then in the class of the object, and finally in UNIVERSE. When it comes to action evaluation, we can consider the structure of Smallworld to be a tree, with UNIVERSE at the root and objects as leaves, as shown in Figure 3.

Smallworld also allows the user to organize classes into *superclasses*. Superclasses serve to share inherited actions between similar classes and to help keep track of a potentially large number of classes that a user could define. Superclasses are classes; that is, they have CLASS property "CLASS". If classes W, X, and Y belong to superclass Z, then each of the component classes (called *subclasses*) have a SUPERCLASS property with value "Z". Z need have no special property for it to be a superclass (other than class CLASS).

Consider, three classes of objects: technical papers, internal memos, and external letters. In our CMS system, objects of all three types would be processed by the same text-processor -- Script/VS. Memos and letters have blank forms (skeletons) that can be filled in, but general papers can take any form. Hence, we will organize three classes: SCRIPT-PAPER, LETTER, and MEMO. The LETTER and MEMO classes are subclasses of SCRIPT-PAPER; they need their own NEW methods to customize their skeletons. All three classes of objects can be printed on the same kind of printers, and can share a PRINTER property. The SCRIPT-PAPER class is, in turn, a subclass of DOCU-

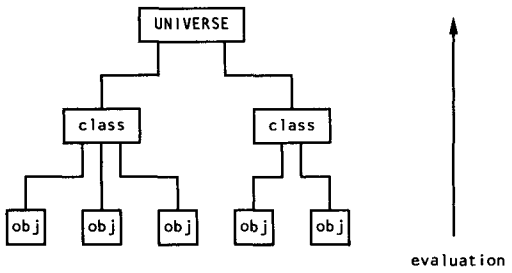

Figure 3. A Simple Evaluation Hierarchy

MENT, which includes other types of text processors, such as YFL and Tex.

To complete the superclass picture, every class that does not have a SUPERCLASS property considers UNIVERSE to be its superclass. Hence, evaluation of an action starts at the subject, proceeds to the class of the subject, then to the superclass of the class of the subject, then to the superclass of the superclass, and so on until UNIVERSE is reached. Searching for properties proceeds in the same way. Smallworld contains actions "SUPERCLASS?" and "SUBCLASSES?" that correspond to the "CLASS?" and "MEMBERS?" discussed before.

An example of the evaluation tree for non-class objects is shown in Figure 4. Notice that the superclasses in the hierarchy are the same as the simple classes we discussed before: they all have class CLASS and can have members and subclasses. They are superclasses only because some other class considers them to be a superclass.

4.2 Actions on Classes

Actions can be applied to classes, because classes are also objects. The same evaluation algorithm applies to classes and to non-class objects. First check the object (itself a class) for a method, then check the class of the object (it must be CLASS) for a method, then check the superclass chain of the class of the object (the superclass of CLASS is UNIVERSE) for a method, until UNIVERSE is reached (the first time). The hierarchy for evaluating actions on classes is shown in Figure 5.

Consider the action NEW. It is defined as an inherited method of CLASS and is used to create a new instance of the subject class. If PROGRAM were a class, then

```
NEW PROGRAM FAST-SORT
```

would create a new object representing the FAST-SORT program. Here PROGRAM is the subject of the action, and the method is inherited from the class of PROGRAM, which is CLASS. Having created a new program, it would be up to the user to add the appropriate properties to the new object, such as compiler, options, and so on. Because programs always have a certain set of properties, we could define a new method for NEW that would prompt for the values of these properties when the new program object was created. This would be a method of PROGRAM; it is not inherited, because it applies to PROGRAM itself, not the members of the PROGRAM class.

4.3 Escape to Universal Methods

Consider the SHOW action as defined for the PROPERTY object. The command

```
show property compiler quicksort
```

shows the contents of the COMPILER property of the QUICKSORT object. How would one, then, show the contents of the PROPERTY object? Using the normal (universal) definition of SHOW, the command

```
show quicksort
```

displays all properties of the QUICKSORT object, but the command

```
show property
```

is preempted by the PROPERTY object; it would result in an error, because no arguments to the action were given.

To solve this problem, Smallworld provides an escape to universal actions. If the name of the action begins with an exclamation mark (!), then the search for the method is short-circuited: only UNIVERSE is checked. Therefore, to show the contents of the PROPERTY object, use

```
!show property
```

4.4 Synonyms for Nouns and Verbs

Some of the names for objects and actions used in Smallworld are long and difficult to type: INHERITED-PROPERTY, MEMBERS?, and so on. These long names were included, because they were less prone to misinterpretation than were short terms and because Smallworld provides a synonym feature.

Smallworld contains an object SYNONYM, belonging to UNIVERSE. The properties of SYNONYM represent abbreviations and substitutions for objects and actions. Synonyms need not result in a one-word to one-word translation; they can be one to many, just by having the SYNONYM property contain more than one word. The expression evaluation algorithm tries to expand the first word of the expression as a verb synonym; the resulting (possibly new) first word is the verb. It then tries to expand the (possibly new) second word as a noun synonym; the (possibly doubly new) second word is the subject. The rest of the string becomes the parameters.

Some suggested synonyms are "p" for PROPERTY, "m" for METHOD, "i-p" for INHERITED-PROPERTY, and "i-m" for INHERITED-METHOD.

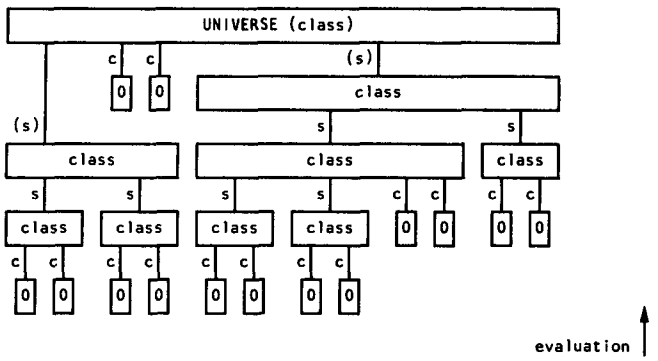

Figure 4. An advanced evaluation tree: Objects are labeled with "O". Links labelled "c" represent the members of a class. Links labelled "s" represent the classes of a superclass. Links labelled "(s)" represent subclasses that belong to UNIVERSE.

The PROPERTY object can contain a property EXPAND-NOUNS. If the value of that property is "YES", then the GET method of PROPERTY will do noun synonym expansion, before executing the GETPROP database function. As a result, "eval(get property x y)" is preferred to "getprop(y,x)" in the definition of a method.

4.5 Expression Evaluation

We now present the full expression evaluation algorithm including inheritance, synonyms, and escapes to universal methods. Call the expression being evaluated EXP. Let WORD1 and WORD2 be functions that extract the first word and second word of a string.

1. Is the first character of WORD1(EXP) an exclamation point? If so, remember that this evaluation requires the universal escape and remove the exclamation point from the beginning of EXP.
2. Is WORD1(EXP) a verb synonym? That is, does the WORD1(EXP).VERB property of the SYNONYM object exist? If so, substitute the value of the synonym for WORD1(EXP).
3. Is the first character of WORD1(EXP) an exclamation point? If so, remember that this evaluation requires the universal escape and remove the exclamation point from the beginning of EXP.
4. Is WORD2(EXP) a noun synonym? That is, does the WORD2(EXP).NOUN property of the SYNONYM object exist? If so, substitute the value of the synonym for WORD2(EXP).
5. Does the WORD2(EXP) represent an object? If so, continue with the next step using WORD1(EXP) as the action-name and WORD2(EXP) as the subject-name. If not, report that the subject does not exist.
6. Does this evaluation require a universal escape? If not, continue with the next step. If so, request UNIVERSE to perform the action on the subject. That is, use the INHERITED:action-name.METHOD property of UNIVERSE. If UNIVERSE is unable to do so, then report that the action is not defined for the subject.
7. Request the subject to perform the action on itself. Use the action-name.METHOD property of the subject, if it exists. If it can, then we are done. If not, continue with the next step.
8. Request the class of the subject to perform the action on the subject. That is, use the INHERITED:action-name.METHOD property of the subject's class, if it exists. If it can, then we are done. If not, continue with the next step.
9. Request the superclass of the class to perform the action on the subject. That is, use the INHERITED:action-name.METHOD property of the superclass. If it can, then we are done. If not and this superclass is not UNIVERSE, then repeat this step on the superclass of this superclass. If not and this superclass is UNIVERSE, then report that the action is not defined for the subject.

4.6 Libraries

Smallworld allows the user to organize the physical storage of objects into separate databases called *libraries*. The user can then specify the order in which the libraries are searched, when an object or property is being retrieved. Libraries also permit the distribution of packages of functions that provide extensions to the base environment. Libraries can be read-only or modifiable, allowing data in a library to be protected or not. Libraries also permit the distribution of packages of functions, such as a CMS package, providing extensions to the base environment.

In CMS the file NORMAL GVXLIBS A contains the default search order for the libraries. A minimal starter set would be

```
USER SW2CMS SW2BASE
```

The SW2BASE library contains the basic objects and methods described earlier in this document and the advanced objects and methods described in this section. The SW2CMS library contains the objects and methods to manipulate the CMS environment; they are described in the appendix. The USER library contains all objects defined by the user. Note that the user library need not be called "USER", any name not used by an existing package would suffice.

Libraries are implemented by a single object, LIBRARY, which belongs to the UNIVERSE class. The LIBRARY object and actions to manipulate it are described later in this section.

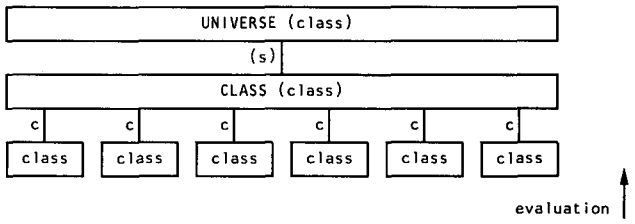

Figure 5. The evaluation tree for classes: Links labelled "c" represent the members of a class. Links labelled "s" represent the classes of a superclass. Links labelled "(s)" represent subclasses that belong to UNIVERSE.

The object-database functions can also manipulate the library structure. The syntax of the functions presented earlier allows for optional library parameters. In addition, two more functions determine in which library a property or method resides.

- EXISTS(object-name, library-name) - returns 1 (true) if the object exists and 0 (false) otherwise. If the optional library-name is specified, then the search for the object will be restricted to the indicated library. If the library-name is omitted, then all libraries will be searched using the current search order.
- GETPROP(object-name, property-name, library-name) - returns the value of a property of an object. The library-name is optional; if it is included, then only that library will be searched for the property. If the library-name is omitted, then all libraries will be searched using the current search order.
- SETPROP(object-name, property-name, value, library-name) - sets the value of the property and returns the previous value as a result. If the optional library-name is specified, then the property will be written in the designated (writeable) library. If no library-name is given then the property will be written in the first writeable library in the current search order.
- ALLPROPS(object-name, library-name) - returns a list of names of all properties of an object. If the optional library-name is included then only those properties residing in that library are returned.
- CREATE(object-name, class-name, library-name) - create a new member of the indicated class with the given name. If the library is specified, then the object will be created in that library.
- DESTROY(object-name, library-name) - remove the object from the database. If the library name is specified, the object (more specifically the CLASS property of the object) will be removed from the library.
- ALLOBJS(class-name, library-name) - returns a list of names of all objects belonging to a given class. If the class-name is null ("") then a list of names of all objects in the database is returned. If the optional library-name is included then only those objects whose CLASS property resides in the indicated library will be returned.
- EVAL(expression) - evaluates a Smallworld expression and returns the value. EVAL has no library parameter.
- PINPOINT(object-name, property-name) - returns a string of the form

```
object-name property-name library-name
```

  The result shows how a GETPROP(object-name, property-name) would be resolved. That is, given that inheritance of properties can occur and that properties can reside in any library in the search order, where will the property finally be found.

- RESOLVE(action-name, subject-name) - returns a string of the form

```
object property library subject
        subject-library
```

  The result shows how EVAL(expression) would be resolved. The resulting string means that the method defining the action belongs to object "object" and is stored as property "property" (such as, action.METHOD) in the indicated library. Furthermore, the subject of the action is the object "subject" and (its class property) is stored in library "subject-library".

Note that the library search order is completely independent of the notions of inheritance. The inheritance algorithm is the same if there is three or thirty libraries in the search list; whenever any property-name is searched for, the libraries are searched (in the search order). The inheritance algorithm can then be thought of as a sequence of property searches.

4.7 The Advanced Environment

The advanced environment contains all of the objects and methods of the base environment. Both the base and advanced environments are contained in the SW2BASE library. In addition, the UNIVERSE class contains the objects SYNONYM and LIBRARY. The class CLASS has two additional inherited methods:

- SUPERCLASS? class-name - returns the name of the superclass of the indicated class.
- SUBCLASSES? class-name - returns the list of all classes that have the indicated class as a superclass.

The SYNONYM object has three methods:

- SHOW SYNONYM word - prints out any noun or verb synonyms of the word.
- NOUN SYNONYM word definition - makes the indicated word into a noun synonym with the specified definition.
- VERB SYNONYM word definition - makes the indicated word into a verb synonym with the specified definition.

The LIBRARY object has six methods:

- ADD LIBRARY lib1 lib2 ... - add the indicated libraries to the beginning of the search order. Only new libraries will be added; existing libraries will not be moved.
- DELETE LIBRARY lib1 lib2 ... - remove the indicated libraries from the search order.
- FORGET LIBRARY library-name object-name - remove all properties of the indicated object from the specified library.
- READONLY LIBRARY lib1 lib2 ... - make the indicated libraries readonly.
- READWRITE LIBRARY lib1 lib2 ... - make the indicated libraries writable. Note that the libraries must reside on writable minidisks (in CMS).
- SEARCH LIBRARY lib1 lib2 ... - change the search order to be lib1 lib2 .... The command SEARCH LIBRARY with no arguments returns the current search order.

5.0 Comparison to Related Work

The programming language Simula is the historical foundation of object-oriented systems [5]. A Simula class is a collection of data objects and procedures that manipulate that data. The data survives between successive calls to the class procedures. In effect, the class provides static storage independent of the main program.

The best known object-oriented language is Smalltalk [6, 7, 8, 11, 15]. See also Rentsch's survey [14]. It is obvious that we were strongly influenced by Smalltalk and have used its terminology when reasonable; though our definitions differ from the original Smalltalk meanings.

Smalltalk consists of objects and classes. The class of an object defines an object much in the way that a Pascal type defines a variable of that type. The class specifies what variables (properties) its objects (instances) can have and defines the methods that its objects can use. Classes can be refinements of superclasses, from which they inherit methods. Smalltalk objects interact only by sending messages to each other. Messages start with the name of the object that is to process the message and follow that with the request. The object attempts to match the message to one of the predefined patterns of its class, superclass, and so on. If the pattern matches, then the corresponding method is executed. Since classes are objects too, they cannot manipulate themselves. They must be manipulated by methods defined in the class's class, called a *metaclass* For example, a metaclass can provide a "new" function that creates a new instance of the class in a special way. Smalltalk classes can override methods defined in superclasses, but they cannot override the values of variables (though variables can be shared by all the objects of a class).

Smallworld differs from Smalltalk in the philosophy of what an object is. In Smalltalk, everything is a refinement of something else. The highest class, "Object" is the eventual superclass of every object. Classes and their subclasses are refinements of "Object". Eventually, non-class objects are instances of a class, which defines all aspects of the object's structure. In some sense the Smalltalk structure is homogenous, because everything is a refinement or instance of something else.

In contrast, Smallworld is heterogeneous: *each object is an independent unit*. Each object is a collection of properties and can define its own methods for implementing actions. The number or type of properties are in no way specified or restricted by the class of the object. Objects are grouped into classes for organization and sharing of properties and methods (methods being just a special kind of property). The objects of a class need not have the same or even similar structures. Inheritance follows similar rules in both systems, but note that Smalltalk must define a special kind of "metaclass" to describe methods for classes, whereas Smallworld treats classes the same as ordinary objects: they can define their own special methods on themselves, or inherit methods from their class.

Another major difference between Smalltalk and Smallworld is their view of the "purity" of the object-oriented paradigm. Smalltalk views every operation as an object or method manipulation, including the steps within the definition of a method. This consistency continues down to the basic level of arithmetic, where 1+2 is a message "+2" sent to the object "1". Efficiency is gained by transparently short-circuiting the object-oriented paradigm at the arithmetic level.

Smallworld, on the other hand, allows both REXX and object-oriented operations within method definitions. In the operating system shell environment, we found that usually it was easier to define methods as conventional programs (with variables, loops, if-statements, and so on) than with the object-oriented paradigm. In contrast, at the user interface, the object-oriented paradigm was simpler. Of course, no special implementations are necessary for Smallworld to optimize arithmetic within methods: all such operations are REXX primitives.

A final difference is the reliance on sophisticated display technology. Smalltalk depends on bit-mapped displays and pointing devices (such as mice) [8, 11]. In contrast, Smallworld runs on conventional display "line-input" terminals. In not relying on high-technology displays, we have shown that the object-oriented paradigm and graphic window systems are not mutually dependent. The object-oriented paradigm is an independent concept and is useful in conventional environments.

Borning and Ingalls [1] have added a system for declaring and inferring types in Smalltalk. They wanted to gain the benefits of compile-time checking and documentation that accrue from type declarations, while still retaining Smalltalk's flexibility. A *type* specifies the messages that an object of that can understand. A message declaration gives the name of the message, the types of its arguments, and the type of the result. Types come in hierarchies: a subtype inherits the message declarations of its supertype (and can add more).

> When the user compiles a method, the system checks at compile time that the messages that might be sent during execution of the method body would be understood by their receivers, that the message arguments will be of the correct type, and that an object of the correct type will be returned.

The type system is not intended to change the Smalltalk inheritance scheme, but rather to reduce errors from "illegal" messages and to eliminate the cost of run-time checking.

The *traits model of subclassing* [3, 4] permits multiple inheritance paths. The the designers of the Xerox Star 8010 workstation began with conventional pure-tree class hierarchies, but found that code became contorted to fit within the tree. They extended the notion of class hierarchy to include directed acyclic

graphs. In their new system, an object is constructed from more primitive abstraction, called traits. A given trait may be defined in terms of several more primitive traits.

> Traits correspond to what Smalltalk calls "abstract classes" (subclasses which generate no instances). The difference is one of emphasis. The term "subclass" seems to emphasize instantiation, whereas the term "trait" seems to emphasize properties that an object may possess [4, p.523].

The multiple-component subclassing increases code sharing by allowing operations to be implemented only once and by allowing the definition of operations to be separate from the problem statement rather than bent to fit artificial design constraints.

The key notion of Sandewall's Carousel system [16] is the use of tree orthogonal abstractions hierarchies: interaction contexts, interactive operations (commands), and data types. The orthogonal hierarchies permit multi-dimensional inheritance. For each interaction between the user and the environment there is a mapping from context (mail program, text editor, and so on) and operation (add, delete, new, and so on) to actual programs. Hence a hierarchy of contexts can be used to say that the mail editor is an instance of the text editor, and can inherit all properties and operations of the text editor. Context independent operations, such as list or tree traversal, are organized in the command hierarchy. They have specialized *leaf procedures* that perform the actual operations on the data (displaying, printing, and so on). These leaf operations have their own mapping from operation and data-type to actual methods, using a hierarchy of data types.

6.0 Conclusions

We have developed Smallworld to be a practical shell language for customizing the VM/CMS user interface. It has been in use (in one form or another) since 1981 [12]. It is not intended to be a programming language, but rather a system for organizing files, programs, and system commands in one manageable whole. The object-oriented paradigm need be used only when the user wishes and when convenient or appropriate; Smallworld can rely on compiled programs, CMS files, or functions of the underlying interpreter.

We have shown (1) the usefulness of multi-paradigm languages and (2) the utility of the object-oriented paradigm in conventional environments. Our future work will be (1) to expand the existing set of Smallworld libraries, (2) to further simplify the class/object dichotomy (if possible), and (3) to investigate more general forms of inheritance and multi-paradigm systems.

7.0 References

1] Alan H. Borning and Daniel H. H. Ingalls. A type declaration and inference system for Smalltalk. *Proceedings of the Ninth Annual ACM Symposium on Principles of Programming Languages* (Albuquerque), pages 133-142. January 1982.

2] M. F. Cowlishaw. The design of the REXX language. *IBM Systems Journal* **23**(4): 326-335, 1984.

3] Gael Curry, Larry Baer, Daniel Lipkie, and Bruce Lee. Traits: An approach to multiple-inheritance subclassing. *Proceedings of the SIGOA Conference on Office Information Systems* (Philadelphia), pages 1-9. June 1982.

4] Gael A. Curry and Robert M. Ayers. Experience with Traits in the Xerox Star Workstation. *IEEE Transactions on Software Engineering* **SE-10** (5): 519-527, September 1984.

5] O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming. A.P.I.C. Studies in Data Processing*, volume 8, pages 175-220. Academic Press, 1972.

6] Adele Goldberg. Introducing the Smalltalk-80 system. *Byte* **6**(8): 14-26, August 1981. This issue of *Byte* is devoted entirely to Smalltalk.

7] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

8] Adele Goldberg. *Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley, 1984.

9] International Business Machines Corporation. *VM/SP System Product Interpreter User's Guide, SC24-5238.* IBM Corporation, 1983.

10] International Business Machines Corporation. *VM/SP Product Interpreter Reference, SC24-5239.* IBM Corporation, 1983.

11] Glenn Krasner. *Smalltalk-80: Bits of History, Words of Advice.* Addison-Wesley, 1983.

12] Mark R. Laff. Smallworld - An Object-Based Programming System. IBM Research Report RC 9022, Yorktown Heights, September 1981.

13] Mark R. Laff. GVX: Global Variables with REXX. In preparation, 1983.

14] Tim Rentsch. Object oriented programming. *SIGPLAN Notices* **17**(9): 51-57, September 1982.

15] David Robson. Object-oriented software systems. *Byte* **6**(8): 74-86, August 1981.

16] Erik Sandewall. Unified dialogue management in the carousel system. *Proceedings of the Ninth Annual ACM Symposium on Principles of Programming Languages* (Albuquerque), pages 146-156. January 1982.

A.0 The SW2CMS Library

The SW2CMS library adds editing capabilities to the property and method objects in the SW2BASE library. Also included is the framework for manipulating and printing CMS files and mail.

- EDIT methods for PROPERTY, METHOD, INHERITED-PROPERTY and INHERITED-METHOD that use the XEDIT editor to view and change the contents of a property. For example,

  ```
  EDIT PROPERTY ID GALLEY-PRINTER
  ```

- A SUMMARIZE inherited method for UNIVERSE that allows a full screen description of an object including properties, methods, subclasses and superclasses. The library associated with each property is also displayed. For example,

  ```
  SUMMARIZE PRINTER
  ```

  would display all information about the PRINTER object.

- CMS - a class.
  - ADDRESS-UPPER CMS command - (method) convert the command to upper case and pass it on to the CMS operating system. Some CMS commands expect that their arguments are all upper case; mixed case can cause a great deal of confusion. Suggested verb synonym: CMS -> ADDRESS-UPPER CMS.
  - ADDRESS CMS command - (method) send the command to CMS; convert only the first word of the command to upper case.

- CP - a member of UNIVERSE.
  - ADDRESS CP command - (method) convert the command to upper case and pass it on to the CP operating system in such a way that the result can be captured by the CMS scrolling environment. Suggested verb synonym: CP -> ADDRESS CP.

- INBOX - a member of CMS.
  - EDIT INBOX - (method) look at your mail with the RDRLIST program.
  - SHOW INBOX - (method) print a list of the mail in your inbox.
  - COPY INBOX - (method) collect all of your mail on other systems and put it in the inbox on the machine you are connected to.

- CONSOLE - a member of CMS.
  - EDIT CONSOLE - (method) look at the transcript of your current session.
  - SHOW CONSOLE - (method) show the status of the console mechanism.
  - COPY CONSOLE - (method) if the system has crashed your last transcript may be in your "virtual printer". Move any such transcripts to your inbox.

- PRINTER - a subclass of CMS.
  - SHOW printer-name - (inherited method) query the status of the indicated printer. Uses auxiliary method GET-PRINTER-ID.

- A set of generic names for printers (members of PRINTER). In increasing quality: LISTING-PRINTER (b-6), DRAFT-PRINTER (modcopy), GALLEY-PRINTER (iowproof), FINAL-PRINTER (aps5). We assume that the user will customize the ID properties of these printers to his or her favorites.

- A special printer PRT (member of PRINTER) that acts as an escape for printers not already defined in the SW2 environment. PRT returns its parameters as the system name of the printer. Hence, to query the status of the IOWCOPY printer, the user could say

  ```
  SHOW PRT IOWCOPY
  ```

- CMS-FILE - a class. File names are stored in CMS-FILE objects using the FN, FT, and FM properties. An auxiliary method GET-FILE-NAME is used to get the file name of an object.

- FILE - a member of CMS-FILE. A special purpose file object (similar to PRT as a special purpose printer object). FILE has its own GET-FILE-NAME method that returns the parameters as the CMS file name.

- PINPOINT methods for PROPERTY and METHOD that expose the underlying "pinpoint" database function.

- An improved SHOW method for PROPERTY that formats its output into a user friendly form. This causes the SHOW methods for INHERITED-PROPERTY, METHOD, and INHERITED-METHOD to become more user friendly as well.

- A TO-FILE inherited method for UNIVERSE that puts all properties of the argument object in a file.

B.0 The PERSON Library

This library provides the structure to maintain a database of people and companies, including electronic mail over VNET and CSNET. Future releases should be able to totally replace the CMS functions of NAMES and NAMEFIND.

- PERSON - a class.
  - NEW PERSON person - (method) create a new person with name, address, phone, and so on. Uses am auxiliary method: CUSTOMIZE.
  - SHOW person - (inherited method) prints out details nicely formatted.
  - CALL person - (inherited method) prints out only the phone number information associated with the person.

- IBM-PERSON - a subclass of PERSON. Suggested noun synonym: IBMER.
  - NEW IBM-PERSON person - (method) creates a new IBM-PERSON and prompts for data on that person. Uses auxiliary method: CUSTOMIZE.
  - MSG person message - (inherited method) send a quick message to the user that will appear on the screen if the user is logged on. Uses FMSG exec, but is similar to TELL command in CMS.
  - MAIL person - (inherited method) send electronic mail to the indicated person. Uses auxiliary method: GET-VNET-ADDRESS.
  - CALL person - (inherited method) check the online telephone directory for the indicated person's phone number, unless a PHONE property exists for the person. If the NAME property of the person differs from the name as specified in the phone book then the PHONE-NAME property can be used to override NAME in the search.
  - WHEREIS person - (inherited method) is the indicated person logged on a local system, if so where?

- NODE - (inherited property) default node id for electronic mail.
- PHONE-ADDRESS - (inherited property) default online phone book for CALL.

- IBM - a member of IBM-PERSON. A special object that allows an escape for person not already in the database. It has its own MSG, CALL, and MAIL functions that use the command line parameters for names or user ids.

```
CALL IBM Hailpern
MAIL IBM mrl at yktvmx
```

  The first example invokes the PHONE exec on "Hailpern"; a good verb synonym for PHONE would be CALL IBM. The second example invokes the NOTE exec with argument "mrl at yktvmx".

- SERVICES - a member of IBM-PERSON. Another special object that provides access to the online "yellow pages" -- the services directory.

```
CALL SERVICES expense
```

  The example would list all entries in the services directory, including the phone number of the Expense Account Department.

- COMPANY - a subclass of PERSON. This class is meant to keep data about airlines, post offices, and so on. It has its own NEW method to prompt for its special properties.

- CSNET-PERSON - a subclass of PERSON. Similar to IBM-PERSON, but for users of the CSnet and ARPAnet. Inherited methods include MAIL and CUSTOMIZE. The CSNET object, a member of CSNET-PERSON provides the same escape for CSNET-PERSON that IBM provides for IBM-PERSON.

- CSNET-NODE - a class. Provides for synonyms for CSnet node names. This is particularly useful for UUCP nodes that have part of their address as a prefix to the user id and the CSnet gateway as a postfix. The properties PREFIX-ID, PREFIX-AT, POSTFIX-ID, and POSTFIX-AT of members of the class allow for easy description of complicated addresses. Default values for PREFIX-AT (!) and POSTFIX-AT (@) are included as inherited properties of the class.

C.0 The DOCUMENT Library

The document library is used to manipulate files that require text processing or printing. The CMS-FILE class from the SW2CMS library becomes a subclass of DOCUMENT, so that printing a CMS-FILE makes sense.

- DOCUMENT - a class. Members of this class either have or inherit properties for file name (FN), file type (FT), file mode (FM), PRINT-COMMAND (for example, PRNT, SCRIPT and YFL), PRINT-OPTIONS, and PRINTER. The NEW method will also impose a skeleton (file name in the TEMPLATE property) on new documents.
  - PRINT - (inherited method) print the document out, using the specified text processor (PRINT-COMMAND) and on the indicated printer (PRINTER). Note that the printer must be a member of class PRINTER or an escape using PRT.
  - EDIT - (inherited method) edit the document using XEDIT.

- SCRIPT-PAPER - a subclass of DOCUMENT. Default file type is SCRIPT. Default PRINT-COMMAND is SCRIPTVS.

- YFL-PAPER - a subclass of DOCUMENT. Default file type is SCRIPT. Default PRINT-COMMAND is YFL (new version on TRACKER 193).

- CMS-FILE - a subclass of DOCUMENT. Default print command is PRNT.

- LETTER - a subclass of SCRIPT-PAPER, with its own template.