A Declarative Model for Defining Smalltalk Programs

Allen Wirfs-Brock, Juanita Ewing, Harold Williams, Brian Wilkerson

Allen Wirfs-Brock
ParcPlace-Digitalk Inc.
7585 SW Mohawk Street
Tualatin, Oregon 97063
Phone: 503-961-0800 x235
Fax: 503-691-2742
Email: allen@parcplace.com

Abstract

Most programming languages have used declarative descriptions for describing programs. Smalltalk has traditionally used an imperative description. In this paper we describe Smalltalk's usage of the imperative model, and identify program maintenance and delivery issues that arise from it. We then present a declarative alternative for describing Smalltalk programs and show how the use of such a model addresses maintenance, delivery, and portability problems encountered by Smalltalk programmers. This model of Smalltalk programs has been used in the implementation of a commercial Smalltalk development environment and is an integral part of the Smalltalk standard which is currently under development by the X3J20 committee.

1. Introduction

Smalltalk programming has traditionally been performed through the imperative execution of reflective Smalltalk expressions within a development environment. A Smalltalk "program" is essentially an accumulation of side-effects upon the development environment. There is no "objective" description of what constitutes such a program. A record of the sequence of expressions may be maintained but their actual effect is highly dependent upon the initial state of all referenced entities within the development environment. The problem is compounded because of the polymorphic nature of Smalltalk. The actual operations performed by a Smalltalk expression are only determinable at the actual time of execution when polymorphic operations are resolved in the context of actual objects.

This style of program definition is a source of significant program maintenance and delivery problems for Smalltalk programmers. Once a program is created, it exists as an extension of the development environment. The most basic problem is the inability to actually identify which program elements are part of an application and which are part of the development environment. This makes it difficult to extract an application from its development environment for delivery, archival, transport, or even collaborative development purposes. Even if a complete record is kept of the expressions used to create the program, initial state dependencies may result in the inability to recreate the program within a different version of a development environment.

Most other programming languages describe programs in a declarative manner. A declarative program description is a sequence of declarations which define the program elements in an objective manner, independent of a development environment context. Declarative descriptions of programs permit them to be easily transported between programmers and language implementations.

We have successfully adapted Smalltalk to use a declarative model of program definition. The declarative version of Smalltalk preserves the essential characteristics of Smalltalk as used by application programmers, but eliminates the principal source of Smalltalk's maintenance, delivery, and portability deficiencies. This declarative model has been used as the basis for a commercial Smalltalk development environment [Digitalk93, Parcplace95] and it has been adopted by the X3J20 committee for use in the Smalltalk standard [X3J20-96] that is currently under development.

This paper examines the problems that result from Smalltalk’s traditional usage of an imperative model of program definition, and presents our alternative declarative model that addresses these problems. We begin in Section 2 by explaining the differences between declarative and imperative modeling techniques. Section 3 examines details of how a program is imperatively defined in traditional Smalltalk development environments. We then link some well known problems in the areas of program maintenance and application delivery to Smalltalk’s use of the imperative definition style. Section 4 shows that a declarative definition style can be applied to Smalltalk without changing the fundamental nature of Smalltalk programs. We present both an execution model and an abstract syntax that supports a declarative model of specifying Smalltalk programs. Section 5 presents the advantages of the declarative model and how it specifically addresses the problems that were identified in Section 3. The use of reflection in the imperative definition of Smalltalk programs has been of significant utility in the implementation of Smalltalk development environments. In Section 6 we show that the use of a declarative model does not preclude the use of reflection and actually offers significant advantages to the implementors and users of a programming environment. Experience with the use of the declarative model in commercial Smalltalk implementations and the prospects for its use in an ANSI Smalltalk standard is described in Section 7. Related work is described in Section 8. In Section 9 we draw some conclusions.

2. Imperative and Declarative Models

An imperative model is a description of an entity that consists of a set of commands (operations, functions, "imperatives", etc.) that, when executed in sequence, will reproduce the entity. Lisp and Smalltalk have traditionally used an imperative model for describing programs. A program is described by a set of commands that will, when executed, reconstruct the program. In traditional Smalltalk implementations the imperative commands that create a program are executed in the same environment as the program that is being created. The commands are expressed in terms of reflective operations upon the data structures that implement the executable program.

A declarative model is a description of an entity that consists of a set of existential statements that enumerate the distinguishing characteristics of the item. FORTRAN, ALGOL, C, COBOL and most other programming languages use a declarative model for describing programs[1]. A program is a set of *declarations* that define the procedures, functions, variables, types, and other elements that make up the program. The declarations for program elements express the characteristics of the elements in terms of abstractions that are independent of the algorithms and data structures used to implement the language. It is possible to fully understand the meaning of such a program by reading it. The reader does not have to know anything about how the compiler works, the runtime representation of variables, or even what computer will execute the program.

A declarative model describes an entity in terms of "what it is". An imperative model would describe the same entity in terms of "how to build it". For example, in the domain of geometry an imperative description of a geometric element might be: "Place a pen at the origin of the coordinate system; move the pen 5 units to the right; move the pen 5 units straight up; move the pen 5 units left; move the pen 5 units straight down." The declarative description of the same figure might be: "A square with sides 5 units in length with its lower left corner at the origin."

This example illustrates many of the advantages of a declarative model over an imperative model. With the declarative description the type of geometric figure is explicitly stated, while with the imperative description the side-effects of executing the commands must be examined in order to recognize the geometric figure. The imperative description unnecessarily constrains the implementation. There are many different ways to create the square as described in the declarative model, the choice is up to the implementation. There is only one way to draw the item from the imperative model. The declarative description is also shorter and easier to modify.

3. Imperative Definition of Smalltalk Programs

Smalltalk program construction has traditionally been performed in the context of a "virtual image" [Goldberg93]. A virtual image consists of a set of objects. These objects include not only those that define a class library that is intended to be used and extended by application programs, but also objects that implement the interactive Smalltalk programming environment itself. In such an environment, a Smalltalk application program

[1] This should not be confused with the concept of "declarative programming languages" such as Prolog where the desired result of a computation is defined in a declarative manner. In this paper we are talking about the declarative specification of imperative programs.

is constructed by directly or indirectly executing imperative Smalltalk expressions that extend and modify the objects within the virtual image to include the classes and variables necessary to implement the functionality of the program. Smalltalk does not include the concept of a program as a distinct entity. In practice, a Smalltalk program is the state of a virtual image when work is declared completed.

The image contains the objects that are the implementations of classes, global variables and pools, but not the imperative expressions that created them. Therefore, to transfer a program to another virtual image, it is necessary to synthesize and externalize expressions that will recreate the program elements. However, the types of some program elements may not be readily discernible by examining their implementation artifacts. For example, in some implementations it is not possible to distinguish a pool from a global variable whose current value is a dictionary with strings for keys. More generally, it is not possible to synthesize the original initialization expressions for global variables. It is only possible to produce expressions that reproduce their current values.

Lack of a program definition in traditional Smalltalk environments leads to an undue reliance on the virtual image. Images can become obsolete or broken. Because the program is encoded in the image, the program is in danger of becoming inaccessible if the image becomes outmoded or corrupt.

Smalltalk’s imperative program construction model also requires that the same virtual image be used both for program creation and program delivery. This makes it very difficult to support situations where the development must be performed in a computing environment that is different from the target execution environment.

The imperative expressions that extend the virtual image into an application program are either entered interactively using programming environment tools or read from an external file. There are two primary tools used for interactively creating Smalltalk programs: a browser and a workspace. A browser is used for defining classes and methods. A workspace is used for composing expressions that define other language elements such as global variables and pools. Workspaces are also used to execute expressions that initialize or test all or part of the program.

Using either a browser or a workspace, a class is created with the message #subclass:instanceVarableNames:classVariableNames:poolDictionaries:. This message, or a variation of it, is sent to a pre-existing class object, the intended superclass. The side-effect of the evaluation of such a message is to create a new global variable (named by the argument of the “subclass:” keyword) whose value is initialized to a new class object which is itself an instance of a newly created metaclass object. It is sometimes difficult to deal with the natural inconsistencies that may arise during the creation and editing of a program. Because a class cannot be defined unless its superclass exists, hierarchical forward references may not be supported, even though other types of forward references may be tolerated by the development environment. It is also difficult to exchange the name of a class with the name of its superclass.

Additional state for the class object may be defined with a message to the metaclass object. The message #instanceVariableNames: is used to define class instance variables. This message can only be sent after the successful creation of the class.

As an example, these expressions create a class named “UIPalette”:

```
ApplicationModel subclass: #UIPalette
 instanceVariableNames: 'activeSpecs toolName '
 classVariableNames: 'ActiveSpecsList CurrentMode PaletteOffsets '
 poolDictionaries: ''!
UIPalette class
 instanceVariableNames: 'selectIcon stickyIcon '!
```

The “!” is used in Smalltalk source files to separate expressions that must be independently compiled and executed. Note that the first expression must be evaluated before the second expression can be successfully compiled. Otherwise, the global name “UIPalette” would not be defined and the compilation would fail. This illustrates a basic problem with the traditional techniques used to externalize Smalltalk imperative program descriptions. The imperative statements are Smalltalk message expressions directed to objects that are referenced via global variables. Because of the polymorphism that is implicit in a message send, the effect of an expression can only be known when it is actually evaluated. Thus the validity of an imperative program description is dependent upon the environment within it is executed.

After a class has been created, methods may be defined for the class or metaclass through the use of a sequence of messages. Each method requires the evaluation of several messages to compile the source code of the method into a CompiledMethod object and the use of the message #compiledMethodAt:put: to actually install it into the class or metaclass.

Because of the complexity and verbose nature of these expressions a browser or file-in reader [Krasner83] is typically used to automate this process. The browser automatically invokes the method creation messages without the programmer having to explicitly type them.

The expressions to define global variables and pools have a similar form. Both involve messages to “Smalltalk”, a global variable containing a dictionary that implements the global name space. Global variables and pools are represented as elements of this dictionary and dictionary message protocol is used to define new global variables and pools. A global variable is defined using a message such as:

```
Smalltalk at: #GlobalVariableName put: nil
```

A pool is itself implemented as a Dictionary, so a pool is created using a sequence of expressions such as:

```
| p |
p := Dictionary new.
p at: ‘Red’ put: Color red.
p at: ‘Blue’ put: Color blue.
p at: ‘Green’ put: Color green.
Smalltalk at: #ColorConstants put: p
```

This sequence creates a pool containing three pool variables. Note that there is nothing that explicitly identifies the entity that is being created as a variable pool. The programmer must know that the language implementation uses instances of the class named Dictionary with strings as keys to represent variable pools, and that the implementation is constrained to use such dictionaries to represent variable pools. The above code would not be portable to an implementation that implements pools using identity dictionaries or some other class or that uses symbols as keys. This is an existing source of portability problems between various Smalltalk implementations.

The implementation of the global name space and pools as dictionaries was a design decision make by the original Smalltalk implementors. There is nothing inherent in the Smalltalk language that requires such an implementation and other designs with various advantages are easy to imagine. However, the imperative definition style precludes the designers of new Smalltalk implementation from considering such alternatives if compatability is a goal

Browsers typically do not include provisions for defining global variables, pools, or pool variables. Instead the appropriate expressions to create them are usually entered using a workspace. Workspace expressions are also used to initialize classes and to set the initial values of global and pool variables. These expressions are often discarded and not captured as a permanent part of the program. The manual and unreliable nature of the initialization of Smalltalk programs leads to a number of program errors. Especially prevalent after reconstruction of a program in a new image are errors where program elements have an initial value of nil instead of some other value as originally intended by the programmer. It is also common for a program execution to start with variables initially set to values that were unintentionally retained from a previous invocation.

4. Declarative Smalltalk Programs

Because of the issues identified above we have chosen to use a declarative model to define Smalltalk programs. This requires the introduction of additional declarative abstractions to the language for program elements that previously had only been defined in terms of implementation artifacts. All elements of a Smalltalk program are described existentially at a level of abstraction that does not overly constrain implementations of the language. The meaning of such a Smalltalk program should be understandable solely from the definition of the program without actually executing a program construction processor or making use of a pre-initialized execution environment.

Unlike our earlier efforts [Wirfs-Brock88], our goal is not to define a significantly different dialect of the Smalltalk language that addresses a broad range of issues. Instead, we reinterpret the process of defining Smalltalk programs from the perspective of a declarative model while making only minimal modifications to the existing core of the language and without obsoleting existing implementations.

The use of a declarative specification model has little direct impact upon Smalltalk programmers. Even though Smalltalk has traditionally been implemented using an imperative program description model, the perception of most Smalltalk programmers is of a declarative model. This is because Smalltalk programmers typically create and edit programs using a browser that presents the classes that make up the program in a declarative style.

The definition of the declarative model of Smalltalk programs consist of two parts. The first part defines the computational model for Smalltalk programs. The second part defines the declarative structure used to specify such computations. Taken together the two parts are intended to define the semantics of a Smalltalk program, but avoid requiring any specific implementation techniques. In addition, the use of reflection is not required in order to define a Smalltalk program. The following are a few traditional assumptions made about the traditional execution environment for Smalltalk programs that are eliminated using this declarative model:

- A system dictionary exists.
- All classes, globals, and pools are in this system dictionary.
- Pools are realized using dictionaries.
- Global and pool variables are represented as associations.
- Each class has an associated metaclass.
- Each class is an object.
- Methods are objects.
- Reflection is permitted and the definition of a program may dynamically change while it executes.

Although our current implementation contradicts none of these assumptions, the goal of the declarative model is to convert Smalltalk programmers to thinking about a program as a specification of a computation and to lay the groundwork for language standardization and further implementation evolution.

4.1 An Abstract Model of Smalltalk Execution

A Smalltalk program is a means for describing a dynamic computational process. This section defines the computational environment for Smalltalk programs.

An *object* is a computational entity that is capable of responding to some set of messages and that encapsulates some (possibly mutable) state. The set of messages is called the object's *behavior*.

An object's behavior is a set of associations between *message selectors* and *methods*, executable functions that are normally written in the Smalltalk programming language. A *message* consists of a message selector and a set of arguments. Each argument is a reference to an object. Program execution proceeds by *sending* messages to objects. An object to which a message is directed is the *receiver* of the message. When an object receives a message, the message selector of the message is used to select the corresponding method from the object's behavior. The method is subsequently evaluated. It is an error if the object's behavior does not include a corresponding selector.

A *variable* is a computational entity that stores a reference (the *value* of the variable) to a single object. The encapsulated state of an object consists of a set of variables. Such variables are called *instance variables*. Normally each variable is bound to an associated instance variable name[2]. The visible extent within a program of such a binding is called the *scope* of the variable.

Variables are not objects. Messages are not sent to variables. The only operations a program may perform upon a variable is to access its current value or to assign it a new value. Other than instance variables, all variables are discrete execution environment entities. These are known as *discrete variables.* Discrete variables are not objects. A discrete variable whose scope is the entire program is a *global variable*. Other types of discrete variables will be defined in subsequent sections.

The objects that exist during program execution consist of both *statically created objects* and *dynamically created objects*. A statically created object is an individual object that is explicitly defined by the Smalltalk program. Typically these are either literals or class objects. Some statically created objects are bound to an object name within some scope. Such objects are called *named objects*. The most commonly occurring named objects are *class objects*.

Dynamically created objects are not individually defined by the program, instead they are dynamically created as a side effect of method execution during the course of program execution. Dynamically created objects do not have names. They are typically referenced as the value of a variable.

During program execution each object must continue to exist, preserving its state, for as long as it is possible to execute any statement that may reference a variable having that object as its value.

Immediately prior to the execution of a Smalltalk program all statically created objects are in their initial state as specified by the Smalltalk program definition and the value of all discrete variables is undefined. Execution proceeds by sequentially executing each

[2]An object may also encapsulate anonymous variables called *indexable instance variables*.

initializer in the order specified by the program definition. The final initializer should initiate the primary computation of the program.

Rationale

The vast majority of Smalltalk application programs do not utilize the reflective capabilities available in traditional Smalltalk implementations. For this reason, we view such reflective capabilities as artifacts primarily used in the implementation of incremental program facilities and do not mandate their presence in all Smalltalk implementations. Given this view, the standard execution model only needs to define the entities that are part of a programmer's view of a running Smalltalk program. These are variables and objects with state and behavior. Implementation artifacts such as compiled methods, method dictionaries, or associations representing variables are excluded.

Class objects have no special significance other than having names and having behaviors and state distinct from that of their associated instance objects. Unlike classic Smalltalk definitions [Goldberg83], they are not defined as being the containers or implementors of their instances' behavior. The techniques used to implement the behavior of objects is left to the implementor. Finally, because classes are not specified as the implementors of behavior, metaclasses are not needed to provide the behavior of class objects.

4.2 An Abstract Syntax for Smalltalk Programs

A Smalltalk program is a means for describing a dynamic computational process. The previous section defined the computational elements of such a process. This section defines the means for describing such a process. It defines the abstract, macro level syntax and static semantics of Smalltalk programs. The macro syntax is used to specify the definitions of globally named entities that make up Smalltalk programs. These entities are classes, global variables, and pools. This grammar is an adjunct to the traditional method syntax which specifies the concrete syntax used for individual method definitions. The specification of this syntax is abstract in that it defines the logical structure of program elements but does not define a concrete syntax for representing this structure. There may be various concrete syntaxes used for storage or interchange of programs.

For the sake of brevity, the static semantic definitions are excluded from this section. The full specification may be found in the appendix to this paper.

Program Definition

The definition of a Smalltalk program consists of a sequence of program element definitions. The program element definitions define all discrete variables, statically created objects and the behaviors of all objects that will take part in the computation. In addition, the program definition specifies the order of dynamic initialization for all program elements.

```
<Smalltalk program> ::= <program element>+
<program element> ::=          <class definition> |
                               <global definition> |
                               <pool definition>
```

Rationale

Traditional Smalltalk systems consider a "program" to be the state of a virtual image when programming is completed. Program execution is the activation of any suspended computational processes within such an image. Under the declarative model a program is a set of definitions (declarations) of language elements along with a specification of the order of initialization. This allows us to precisely describe and repeatably produce an executable program whose computation will proceed in a well defined manner.

Traditionally, Smalltalk messages are used to reflectively create program elements. We have replaced these messages with an abstract syntax that declaratively specifies the existential attributes of each program element. Although a concrete syntax that appears similar to the traditional messages could be used, its interpretation would be declarative not imperative.

Throughout this description of the declarative model of Smalltalk programs there are no references to any of the traditional Smalltalk program implementation artifacts. Pools and the superclass are referenced by name rather than by the objects (Dictionaries and Classes) that might implement them. Other components of a class, such as the methods, are also specified such that references to implementation artifacts, such as CompiledMethods or method dictionaries, are avoided.

Class Definition

A *class definition* defines the instance variable structure (the encapsulated state) and behavior of objects. In addition, a class definition introduces a named object binding with global scope. This name is called a *class name* and the associated object is a *class object*.

A class definition specifies the behavior and instance variable structure for both the statically created class object and dynamically created instances of the class.

A class definition specifies two behaviors, the *instance behavior* and the *class behavior*. The instance behavior is the behavior of any dynamically created instances of the class. The class behavior is the behavior for class object. Through the use of inheritance, the instance variable structure and behavior for both the class object and instance objects may be specified as a refinement of that specified by another class definition known as its *superclass*. Conversely, a class definition that inherits such structure or behavior is known as a *subclass* of its superclass.

A class definition may also define discrete variables called *class variables* whose scope is all methods (either class or instance methods) defined as part of the class definition or as part of the class definitions of any subclasses of the class definition. In addition a class definition may specify the importation of pools. Any pool variables defined in such pools are included in the scope of all methods defined as part of the class definition.

```
<class definition> ::=
        <class name> [<superclass name>]
                [<instance variables>]
                [<class instance variables>]
                [<class variables>]
                [<imported pools>]
                [<instance methods>]
                [<class methods>]
                [<class initializer>]
<class name> ::= <global identifier>
<superclass name> ::= <global identifier>
<instance variables> ::= <local identifier>*
<class instance variables> ::= <local identifier>*
<class variables> ::= <global identifier>*
<imported pools> ::= <global identifier>*
<instance methods> ::= <method definition>*
<class methods> ::= <method definition>*
<class initializer> ::= <expression sequence>
```

Rationale

Traditional Smalltalk systems consider a “class” to be an object that defines and implements the behavior of other objects [Goldberg83]. Because every object must be an instance of some class, metaclasses are required to provide the class of the class object. Under the declarative model, a “class definition” is a declarative element that is used to specify the structure and behavior of objects. The declarative model provides for a specification of object behavior without requiring any particular style of implementation. In particular, it does not require that behavior be implemented in terms of runtime class objects. This eliminates the requirement for metaclasses and all other reflective entities from the abstract model of Smalltalk programs.

At runtime, classes are simply objects that respond to the class message protocol. It is an implementation decision whether class objects are also used to implement the behavior of instance objects. Similarly, it is an implementation decision whether metaclasses are used in the implementation of the behavior of class objects.

Global Definition

A global variable definition is used to specify a discrete variable whose name scope includes the entire program. The definition includes the Smalltalk code providing the initial value of the variable.

```
<global definition> ::= <global variable name> [<variable initializer>]
<global name> ::= <global identifier>
<variable initializer> ::= <expression sequence>
```

Rationale

Traditional Smalltalk systems define a “global variable” as an association within a dictionary which itself is a global variable named Smalltalk. The value instance variable of such associations is used as the storage cell for these variables. Dictionary messages are used to reflectively add such associations and thereby create new global variables. Classes and pools are also typically accessed via associations in this dictionary.

The declarative model recognizes that a global variable is simply a discrete storage cell that exists during program execution. The model is neutral concerning which of the many possible implementation strategies for discrete variable cells might be used by an implementation. There is no implicit or explicit requirement that the name of the variable must be present during program execution nor that a variable is represented by an object. It is also not required that a runtime mechanism exist for using a string to access a variable or for enumerating over all global variables.

Pool Definition

A pool definition introduces a global name binding for a variable pool and defines the names of the discrete variables within the pool.

```
<pool definition> ::= <pool name> <pool variable definition>*
<pool variable definition> ::= <pool variable name> [<variable initializer>]
<pool name> ::= <global identifier>
<pool variable name> ::= <global identifier>
```

Rationale

Traditional Smalltalk systems define a "pool dictionary" as a dictionary, each of whose associations is a used to represent a variable. Dictionary messages are used to reflectively add such associations and thereby create new pool variables.

In the declarative model, pools are simply a scoping mechanism for discrete variables. As with global variables, there is no implication or requirement that actual dictionaries or associations be used to implement these variables. Because of the elimination of the need to use reflective dictionary operations to create pool variables, the only context where a reference to a pool name is defined is an <imported pool> production of a class definition. Because the declarative model does not specify that a pool name is bound to an object, the meaning of a reference to a pool name in a method or initializer is also unspecified. However, the fact the pool names are including in the global name space permits implementations to continue to use traditional implementation techniques and define that references bind to an underlying dictionary object.

5. Advantages of the Declarative Model

5.1 Precise Program Definition

Declarative specification simplifies the identification of programs. Though a virtual image is no longer a required feature, Smalltalk implementors may still chose to support one. With definitions available for all program elements, no analysis of an image is required to externalize the definitions. This has the following advantages:

- The program is clearly separated from the image. It is easy to externalize programs and load them into a new image either to recover from a corrupt image or to move to a new version of the base virtual image.

- It is easy to recreate a program because all the information about the program is recorded as definitions. Programmers need not record and manage workspaces used to create and initialize program elements.

- Program refinement and evolution is simplified because definitions can exist and be manipulated even if they cannot be compiled given the current state of the program. A class could be developed, for example, in the absence of definitions for its superclass or collaborating classes. Porting an application using traditional Smalltalk tools implies "filing in" code generated from another system. Illegal references to classes or globals generally aborts the process. The programmer must create stub classes or edit the file-in file to eliminate conflicts and iterate until the file is successfully read before its contents can be browsed. Definitions, however, can be browsed even if they can't be compiled.

5.2 Known starting point for program

The declarative model specifies a well known starting point for a program execution. This is accomplished by specifying the order of the initialization definitions within a program such that they run in a deterministic order. This determines an initial state that allows Smalltalk programs to behave consistently each time they are executed.

5.3 Program Definition Separated from Implementation Artifacts

The declarative model decouples program specification from language implementation decisions. Once programs are created using definitions, Smalltalk implementors are freed from various constraints and assumptions about how language elements are to be translated into an executable Smalltalk program. Some areas of opportunity include:

- Implementations can evolve without impacting existing programs. Because a program is defined without reference to implementation artifacts, an implementor has the freedom to make changes to the implementation without the fear that the changes will invalidate existing programs.

- A program can be portable between implementations even if they use very different implementation strategies and artifacts. For example, a Smalltalk implementation designed to support deployment of embedded applications might use significantly different implementation strategies than an implementation that was designed to support rapid prototyping.

5.4 Pool Distinguishable from Global Variables

The declarative model allow pools to be distinguished from global variables, even if the underlying implementation is identical. By defining the semantics of pools such that they are only referenced at compilation time, this opens the door for static analysis of programs and various optimizations.

Pools can be pruned to their required size and need not be represented as dictionaries. Large general-purpose pools are sometimes created during development without knowing which pool variables will ultimately be required by the program. A pool containing operating system constants, for example, must be very large to be complete but only a small fraction of the entries may be used by any particular program. When programs are defined declaratively, two optimizations can be performed. First, the variables that are actually referenced can be identified so that unused pool variables can be omitted from the executable program. Second, the pool need not be realized as a dictionary nor pool variables as associations. If a pool variable is determined to be read-only (no assignments other then its initializer), the value can be in-lined and the association that traditionally implements the variable can be eliminated.

6. Reflection and Interactive Development Environments

Smalltalk development environments are known for their high degree of interactivity and functionality. They support fine grained incremental creation of Smalltalk programs with immediate feedback and executability. They also support interactive programming aids such as immediate cross-referencing and non-linear access to program elements using browsers. It was the reflective implementation of the imperative definition model that originally enabled Smalltalk environments to provide this level of functionality.

Incremental program creation and immediate execution is possible because the individual program elements are added reflectively to an already executing program. Each new class or method immediately becomes part of the currently executing development environment program and hence can be immediately executed. Similarly, any modification of an existing program element also has immediate effect. Interactive debugging is also implemented through reflection upon the objects that implement the program.

Higher level services such as browsing and cross-referencing are also implemented using reflection upon the objects that implement the program. Browsers locate classes by navigating the data structures that implement inheritance (subclass references within class objects) or by directly accessing the "symbol table" (the system dictionary) that stores global names. Cross-referencing is accomplished by scanning method dictionaries to determine which classes implement particular methods or by scanning compiled methods to see if a particular method is used to send a message.

What traditional Smalltalk development environments are actually doing is using reflective operations upon the implementation data structures as an "object model" of the program. The existence of such a directly manipulatable object model means that the programming environments can directly operate upon the logical elements of the program and avoid time consuming parsing and editing of a textual representation of the program in an external file. Arguably, it was the existence of a directly manipulatable program model that enabled implementors of Smalltalk and Lisp to create the first highly interactive development environments.

Usage of a declarative model of Smalltalk program definition in no way precludes use of such a reflective program model. A programming environment may translate the declarative specification of the program into implementation objects that it then directly manipulates. Similarly, an implementation may provide runtime access to implementation objects to enable reflection from within application programs.

Adoption of the declarative model provides an opportunity for development environments to use a new form of program object model, one that models the declarative specification of the program rather than its implementation artifacts. Such an object model has objects that directly correspond to the elements of a declarative Smalltalk program definition (classes, methods, pools, variables, etc.) rather than the implementation artifacts (CompiledMethods, MethodDictionaries, Associations, Symbols, etc.) Typical operations upon the model include declaring a class or variable, removing a method definition, or querying to find all definitions that reference some other definition. These operations upon the object model are also expressed in terms of manipulations of the declarative program specification abstractions rather than the direct manipulation of the runtime implementation of the program elements. However, an operation may, as a side-effect, translate changes to the abstract program model into changes in the executable form of the program.

Such a declarative object model has a number of advantages. Its enables tool builders and end-users to write code to programatically manipulate Smalltalk program definitions without having to understand all the details of the underlying implementation. More importantly, it decouples the model of the program used by the development tools from the actual executable implementation of the program elements. This permits the technology for executing Smalltalk code to evolve without requiring changes to the implementation of the programming environment tool set. It also permits a tool set to support alternative execution environments or technologies. Finally, it permits the execution environment for a Smalltalk program to be completely separate and distinct from its development environment. Such a separation has the potential to enable or greatly simplify the implementation of capabilities such as cross development or remote debugging and to significantly increase the robustness and reliability of the Smalltalk development environment.

7. Experience and Application of the Declarative Model

A declarative model of Smalltalk program specification was first used in our design of Modular Smalltalk [Wirfs-Brock88]. Subsequently, this approach was explored and refined for more conventional Smalltalk dialects in the context of developing advanced development environments for Smalltalk-80 [Goldberg84], Smalltalk/V, Visual Smalltalk, and VisualWorks dialects of Smalltalk. Team/V [Digitalk93, Parcplace95] is a comprehensive commercial Smalltalk development environment that is based upon the declarative model.

Team/V uses a fully declarative model of Smalltalk programs. Externally, program definitions are stored in a textual declarative format. This format can be translated into a dynamic object model of the declarative program definition and the object model can be externalize in the textual form. The object model serves as the "API" for both the development tools and user "scripts" that programatically manipulate the program definitions. Changes to the declarative program model have the side-effect of reflectively modifying the executable form of the program.

Team/V fully exploits the declarative model to allow Smalltalk program elements to be reliably externalized and transported between different versions of the development environment. Most significantly, Team/V is able to directly create deliverable versions of the application from the declarative description. Other Smalltalk development environments require a “stripping” process that attempts to heuristically extract a deliverable version of an application from the development environment.

The advantages of the declarative model for specifying Smalltalk programs have also been recognized by the X3J20 committee which is chartered with developing an ANSI standard [X3J20-96] for the Smalltalk programming language. The use of the declarative model is expected to result in a language standard that precisely specifies the meaning of conforming programs while allowing implementors wide latitude in implementing conforming implementations.

8. Related Work

Although they were not expressed in these terms, issues arising from the imperative definition of Smalltalk programs motivated the development of Modular Smalltalk [Wirfs-Brock88] and led to its use of a declarative style of program specification. Much of the work described in this paper can be viewed as the application of these results to mainstream Smalltalk dialects.

The designers of modern object oriented languages including Eiffel [Meyer92] and more recently Java [Gosling95] have used a declarative program specification model. The designers of Dylan [Apple95] explicitly recognized problems that imperative program definition created for application program delivery and chose to use the declarative model for a dynamic object-oriented language.

Other researchers and language designers have chosen to emphasize and expand the imperative reflective nature of languages similar to Smalltalk. Self [Ungar87] took as one of its starting points the capability to directly manipulate objects that is available in reflective Smalltalk systems and broadly generalized it such that programming becomes a process of directly instantiating concrete objects with unique behaviors. In Self, a program is generally considered to be a collection of dynamic objects, and program interchange or transportability is treated as transporting such objects between execution environments [Ungar95].

9. Conclusions

In this paper we have contrasted imperative and declarative techniques for describing programs and have argued that Smalltalk’s use of an imperative program definition model is a factor in many program maintenance and delivery issues encountered by Smalltalk programmers. We assert that a declarative program definition model that addresses these issues can be retrofitted to the Smalltalk language with minimal impact upon existing Smalltalk implementations and programmers.

In support of this assertion we have presented a specification for Smalltalk program definition and Smalltalk program execution that is based upon a declarative model. Application of this specification has enabled us to create Smalltalk development environments that significantly improve the process of creating, maintaining, and delivering Smalltalk applications. Use of the declarative model also provides Smalltalk implementors the opportunity to be innovatative in their implementation technology while increasing the portability of Smalltalk programs among various implementations of the language.

We believe that the adoption of a declarative model of Smalltalk program definition is an important step in the maturation of the Smalltalk programming language. It is a key component of the emerging definition of a Smalltalk standard and will further solidify Smalltalk's emerging status as a "mainstream" application development language.

Acknowledgments

Many of our colleagues at Tektronix, Instantiations, Digitalk, and ParcPlace-Digitalk have participated in the development of the declarative model of Smalltalk and programming environments based upon it. John Wiegand, Dale Henrichs, and Carl McConnell have all been instrumental in the creation of an object model for manipulating declarative Smalltalk programs. Steve Messick, Tim O'Connor, and Pat Caudill have contributed to the development of development tools that exploit this object model.

We would also like to acknowledge the organizational and individual participants in the X3J20 Smalltalk standardization process. They have been exceptionally receptive and supportive of the adoption of this declarative model. Some of the material in this paper has appeared in significantly different forms in various working papers of the X3J20 committee.

References

[Apple95] Apple Computer, *Dylan Reference Manual*, 1995,
http://www.cambridge.apple.com/dylan/drm/drm-1.html

[Digitalk93] Digitalk Inc., *Team/V Programmers Reference Manual*, 1993

[Goldberg83] Adele Goldberg and David Robson, *Smalltalk-80 The language and its Implementation*, Addison Wesley, 1983.

[Goldberg84] Adele Goldberg, *Smalltalk-80: The Interactive Programming Environment*, Addison Wesley, 1984.

[Gosling95] James Gosling and Henry McGilton, "The Java Language Environment: A White Paper", 1995, http://java.sun.com/whitePaper/java-whitepaper-1.html

[Krasner83] Glenn Krasner, "The Smalltalk-80 Code File Format" in *Smalltalk-80 Bits of History, Words of Advice*, Glenn Krasner editor, Addison Wesley, 1983.

[Meyer92] Bertrand Meyer, *Eiffel: The Language*. Prentice-Hall, New York, 1992.

[ParcPlace95] ParcPlace-Digitalk Inc., *Visual Smalltalk Enterprise Tool Reference Manual*, 1995

[Ungar87] David Ungar and Randall B. Smith, "Self: The Power of Simplicity", in *Proceedings of OOPSLA '87*, Orlando, Florida, October 1987, pp. 227-241.

[Ungar95] David Ungar, " Annotating Objects for Transfer to Other Worlds", in *Proceedings of OOPSLA '95*, Austin, TX, October 1995, pp. 73-87.

[Wirfs-Brock88] Allen Wirfs-Brock and Brian Wilkerson, " A Overview of Modular Smalltalk", in *Proceedings of OOPSLA '88,* San Diego, CA, September 1988, pp. 123-134.

[X3J20-96] X3J20 Committee, *Working Draft Smalltalk Standard*, March 1996.

Appendix — Abstract Syntax and Static Semantics of Smalltalk Programs

This appendix is a more complete specification of the abstract syntax and semantics for Smalltalk programs that is summarized in section 3. This specification is a somewhat simplified version of what has been proposed for use in the Smalltalk standard that is currently under development by X3J20.

Program Definition

The definition of a Smalltalk program consists of a sequence of program element definitions. The program element definitions define all discrete variables, statically created objects and the behaviors of all objects that will take part in the computation. In addition, the program definition specifies the order of dynamic initialization for all program elements.

```
<Smalltalk program> ::= <program element>+
<program element> ::=          <class definition> |
                               <global definition> |
                               <pool definition>
```

The order of the <program elements> determines the initialization order of the program elements.

Class Definition

A *class definition* defines the instance variable structure (the encapsulated state) and behavior of objects. In addition, a class definition introduces a named object binding with global scope. This name is called a *class name* and the associated object is a *class object*. A class definition specifies the behavior and instance variable structure for both the statically created class object and dynamically created instances of the class.

A class definition specifies two behaviors, the *instance behavior* and the *class behavior*. The instance behavior is the behavior of any dynamically created instances of the class. The class behavior is the behavior for class object. Through the use of inheritance, the instance variable structure and behavior for both the class object and instance objects may be specified as a refinement of that specified by another class definition known as its *superclass*. Conversely, a class definition that inherits such structure or behavior is known as a *subclass* of its superclass.

A class definition may also define discrete variables called *class variables* whose scope is all methods (either class or instance methods) defined as part of the class definition or as part of the class definitions of any subclasses of the class definition. In addition a class definition may specify the importation of pools. Any pool variable defined in such pools is included in the scope of all methods defined as part of the class definition.

```
<class definition> ::=
        <class name> [<superclass name>]
                [<instance variables>]
                [<class instance variables>]
                [<class variables>]
                [<imported pools>]
                [<instance methods>]
                [<class methods>]
                [<class initializer>]
<class name> ::= <global identifier>
<superclass name> ::= <global identifier>
<instance variables> ::= <local identifier>*
<class instance variables> ::= <local identifier>*
<class variables> ::= <global identifier>*
<imported pools> ::= <global identifier>*
<instance methods> ::= <method definition>*
<class methods> ::= <method definition>*
<class initializer> ::= <expression sequence>
```

The <class name> is the global name of the class object. There may be no other global definitions of this name within the program. The binding of the <class name> to the class object is a constant binding. It may not be the target of an assignment statement.

The <superclass name> identifies the class definition from which this definition inherits. It must be the <class name> of another <class definition> within this program. The instance behavior defined by the class definition consists of the instance behavior of the superclass augmented by the <instance methods> of the class definition. An <instance method> whose selector is the same as the selector of a method in the inherited behavior replaces the inherited method in the behavior. Similarly, the class behavior defined by the class definition consists of the class behavior of the superclass augmented by the <class methods> of the class definition.

If the <superclass name> is absent then this class has no inherited instance behavior and the instance behavior consists solely of the <instance methods> that are part of the class definition. The class behavior of such a class is defined to inherit from the instance behavior of the <class definition> whose <class name> is the identifier ‘Object”.

It is an error if the <superclass name> is the same as the <class name> or if <superclass name> is the name of a class that directly or indirectly (via inheritance) specifies <class name> as a superclass.

The encapsulated state of an object consists of a fixed set of variables capable of referencing any object and an optional variable sized, anonymous sequence of such variables. The variable sized sequence of anonymous instance is known the *indexable-part*. The size of an object's indexable-part is fixed when the object is instantiated. Class objects do not have an indexable part.

The <instance variables> production defines the names of instance variables for instances of the class. If present, it must include a set of identifiers. These identifier are called *instance variable names*. It is erroneous for the same identifier to occur more than once in the set of instance variable names. The meaning of the class definition is undefined if any of the instance variable names is the same identifier as an instance variable or class variable defined by any superclass. It is an error for an instance variable name to also appear in the set of class variable names.

The fixed set of instance variables encapsulated by an instance object consists of one variable corresponding to each instance variable name specified in the class definition and transitively in any superclass definitions. The instance variable names are included in the scope of any <instance methods> defined by the class definition or its subclasses. The binding of such an instance variable name is the corresponding instance variable of the object that is the receiver of the message that invoked the method.

The <class instance variables> production defines the names of instance variables of the class object. If present, it must include a set of identifiers. These identifiers are called *class instance variable names*. It is erroneous for the same identifier to occur more than once in the set of class instance variable names. The meaning of the class definition is undefined if any of the class instance variable names is the same identifier as a class instance variable or class variable defined by any superclass. It is an error for a class instance variable name to also appear in the set of class variable names.

The fixed set of class instance variables encapsulated by a class object consists of one variable corresponding to each class instance variable name specified in the class definition and transitively in any superclass definitions. The class instance variable names are included in the scope of any <class methods> defined by the class definition or its subclasses. The binding of such a class instance variable name is the corresponding class instance variable of the object that is the receiver of the message that invoked the method.

The <class variables> production defines the names of discrete variables which are accessible by both class and instance methods of the class and its subclasses. If present, it must include a set of identifiers. These identifiers are called *class variable names*. It is erroneous for the same identifier to occur more than once in the set of class variable names. The meaning of the class definition is undefined if any of the class variable names is the same identifier as an instance variable, class instance variable or class variable defined by any superclass. It is an error for an class variable name to also appear in the set of instance variable names or class instance variable names.

The fixed set of class variables encapsulated by a class object consists of one variable corresponding to each class variable name specified in the class definition and transitively in any superclass definitions. The class variable names are included in the scope of any <instance methods> or <class methods> defined by the class definition or its subclasses. The binding of such a class variable name is the corresponding class variable of the object that is the receiver of the message that invoked the method.

The <imported pools> production, if present, must include a set of identifiers. These identifiers are *pool names*. It is erroneous for the same identifier to occur more than once in the set of pool names. It is an error if a pool name is not the <pool name> of a <pool definition> within the program. All pool variable names contained within the pools associated with the names in the set of pool names are included in the name scope of the class and instance methods of this class.

The behaviors of the instances of the class and the class object are defined, respectively, by the <instance methods> and the <class methods>.

The <class initializer> production consists of a zero argument block body that defines the code that is used to initialize the class variables and class object defined by the class definition. The name scope of the class initializer is the same as that of a class method for the class. A class initializer is not inherited by subclasses. The value returned by a class initializer is discarded.

Global Definition

A global variable definition is used to specify a discrete variable whose name scope includes the entire program. The definition includes the Smalltalk code that provides the initial value of the variable.

```
<global definition> ::= <global variable name> [<variable initializer>]
<global variable name> ::= <global identifier>
<variable initializer> ::= <expression sequence>
```

The <global variable name> is the global name of a discrete variable. There may be no other definition of this name as a global, class or pool within the program. The <variable initializer>, when evaluated, provides the initial value of the global. If no <variable initializer> is specified the initial value is nil. The value of the global, before evaluation of the <variable initializer> is undefined.

Pool Definition

A pool definition introduces a global name binding for a variable pool and defines the names of the discrete variables within the pool.

```
<pool definition> ::= <pool name> <pool variable definition>*
<pool variable definition> ::= <pool variable name> [<variable initializer>]
<pool name> ::= <global identifier>
<pool variable name> ::= <global identifier>
```

The <pool name> is the global name of the variable pool. There may be no other definition of this name as a global, class or pool within the program. An identifier that is bound to a variable pool with a <pool definition> is called a *pool name*. Pool names normally are listed in the <imported pools> production of a <class definition>. The meaning of using a pool name in any other context is unspecified.

A <pool variable definition> introduces a name binding within a pool for a discrete variable. The <pool variable name> is the name of the discrete variable. It is an error to have more than one definition of a name within the same pool. The <variable initializer>, when evaluated, provides the initial value of the pool variable. If no <variable initializer> is specified the value is nil. The value of the pool variable, before evaluation of the <variable initializer>, is undefined.