OOPSLA'96 Extending Smalltalk Workshop

Subsystems

– Proposal–

October 7, 1996

Allen Wirfs-Brock

ParcPlace-Digitalk[1]

One of the most attractive features of Smalltalk is the ease with which an application can be constructed from reusable class libraries. One of the problems that can occur when using independently developed class libraries is that the developers of the libraries may have inadvertently chosen the same name for different classes. Some modern Smalltalk development tools can detected such conflicts as errors but they do not provide any help to the programmer that actually needs to use both of the identically named classes. Such help is provided through the use of *subsystems.*

Subsystem Concepts

A **Subsystem** is a separate, global name space[2] within a Smalltalk program (or image). Every program consists of one or more subsystems. Each subsystem defines an independent set of global names (classes, global variables, and variable pools). Within a subsystem, a global name may not be multiply defined. For example, a class named "Foo" cannot be defined twice within a single subsystem nor could "Bar" be defined as both a global variable and as a class within a subsystem. Because the global name spaces of separate subsystem are independent, a single name may have different definitions in different

---

[1] Author's current affiliation: Instantiations Inc, Allen_Wirfs-Brock@Instantiations.com

[2] Smalltalk programmers have traditionally thought about its single "global name space" in terms of the dictionary named Smalltalk that was used to implement it. In this document name spaces are generally discussed at a more abstract level that avoids implementation details. An implementation of this proposal might implement subsystem name spaces by providing a separate dictionary similar to the Smalltalk dictionary for each name space.

subsystems. For example, the name "Foo" could be defined as a class with two different definitions in two different subsystems. "Bar" could be defined as a global variable in one subsystem, as a class in a second subsystem, and as a variable pool in a third subsystem. Essentially, each subsystem has its own independent set of class definitions. Several subsystems can contain class definitions that have the same name but different structures and behaviors.

A subsystem encapsulates definitions of Smalltalk program elements (classes, methods, variables, initializers, etc). Program element definitions might be further managed by grouping them within modules (such as Team/V packages and clusters or Envy applications) however the module structure has no impact upon the name space semantics of a subsystem. Alternative code management systems could be used to manage or enter the definitions within a subsystem or a subsystem could be treated as an unstructured set of program element definitions.

The contents of a subsystem's name space includes all of the global names of program elements defined in the subsystem. Smalltalk code in program element definitions within a subsystem may directly reference, by name, any global name defined in the same subsystem. Names defined in other subsystems normally are invisible and hence cannot be directly accessed. Even though a class may not be directly accessible from outside its defining subsystem, it is still possible to send messages to instances of the class from within different subsystems.

Sharing Between Subsystems

Occasionally it is useful to have global definitions that are shared between several subsystems. For example, we normally would not want each subsystem to have its own local definition of the class named "Object". All subclasses in all subsystem should normally inherit from a common class "Object". This can be accomplished through the use of *imported names*.

An **imported name** is a global name that is referenced within a subsystem but which is not defined by that subsystem. The program element (a class, global variable, or variable pool) that is bound to the imported name must be defined by another subsystem within the same program.

A program element defined within a subsystem is available for binding to imported names within other subsystems only if the global name of the program element is an **exported name**. All global names are exported names unless the definition of the global name explicitly designates the name as **private**[3]. Note

[3] Exported was chosen as the default for consistency with the traditional "public" nature of Smalltalk.

that because imported names are global names within a subsystem, an imported name will also be an exported name unless the imported name is explicitly designated as private[4].

A subsystem is not fully defined until specific program elements are bound to its imported names. These bindings are specified by a *program specification*.

A **program specification** is an ordered list that identifies all of the subsystems that make up a program. The program specification defines for each subsystem the bindings for its imported names. There are two mechanisms that may be used to specify the bindings: *explicit binding* and *implicit binding*. The two mechanisms may be used in combination with explicit binding taking precedence over implicit binding.

Explicit binding associates an imported name with a specific exported name defined in another, explicitly identified subsystem[5]. The exported name need not be the same as the imported name.

Implicit binding uses a list of subsystems[6] that are to be used to provide the bindings for any unbound imported names. For each unbound imported name, the subsystems on the binding list are examined, in order, until an exported name that is identical to the unbound imported name is encountered. The binding of the imported name is set to the same program element definition as the matching exported name.

Consider the following example. Assume that a program is composed of three subsystems, described as follows[7]:

Subsystem Standard The “base image”. Defines and exports, among other things, Object, Array, and File.

Subsystem NewFiles

Defines and exports a class named File. Imports Object.

Subsystem Application Imports Object, File and Array.

---

[4] This means that a subsystem can be used as a renaming mechanism.

[5] This must be a “backward” reference within the program specification.

[6] Called the “binding list”. It also may only contain backward references.

[7] This syntax is for explanatory purposes only.

If the program wants the subsystem named "Application" to use the standard definition of Object and the "new" definition of File its program specification could be:

```
Subsystem Standard

Subsystem NewFiles uses: (Standard)

Subsystem Application uses: (NewFiles, Standard)
```

The first line simply specifies that "Standard" is part of this program. Because it has no imports its specification does not include any binding information. The second line specifies that "NewFiles" is part of this program and that any imported names should be implicitly bound using the definitions in "Standard". Finally, the third line includes the subsystem "Application" and specifies that any imported names should be resolved by first looking in "NewFiles" and if a matching name is not found then looking in "Standard". So, "Application" will import File and Object[8] from "NewFiles" and Array from "Standard".

If "NewFiles" defined a class named BetterFile instead of a class named File, explicit binding could be used to allow "Application" to import BetterFile as File:

```
Subsystem Standard

Subsystem NewFiles uses: (Standard)

Subsystem Application
    set: File from: NewFile as: BetterFile
    uses: (Standard)
```

Imported Name Definitions

Within a subsystem, imported names are treated like other language elements and specified using a definition. An imported name definition specifies an identifier and a *usage* of the identifier. A usage defines the intended use of the name within the subsystem and limits the type of program element that may be bound to the imported name. The valid usages are: *constant*, *variable*, *class*, and *pool*.

[8] This might be called transitive binding. Object is imported from "NewFiles" where it is defined as an imported name. The actual class definition is provided by "Standard", then imported into "NewFiles" and finally imported into "Application".

A **constant usage** is designated for imported names that are used to access objects. Such imported names may appear in any expression context except that they may not be used as the target of an assignment statement.

A **variable usage** is designated for imported names that are used as targets of assignment statements. Such imported names may also appear in all other expression contexts.

A **class usage** is designated for imported names that are used as the superclass in a class definition or as the extended class in a loose method definition[9]. Class usage implies constant usage.

An imported name with class usage may also be referred to as an *imported class*. If a method definition[10] in a subsystem needs to refer to an instance variable (including class instance variables), class variable, or pool variable of an imported class then the imported name definition must include the name of the variable as part of the class usage specification. Such variables are called *imported instance variables* or *imported pool variables*[11].

A **pool usage** is designated for imported names that are used as a pool reference in a class definition or as the extended pool in a loose pool variable definition. Pool usage also implies constant usage.[12]

Tool Assistance

The Smalltalk programming environment can automate much of the process of defining and binding imported names.

Any name that is referenced but not defined by a subsystem could be automatically defined (assuming user concurrence) as an imported name and the usage could be inferred from the referencing context. Similarly, within a method definition for an imported class or a subclass of an imported class, imported instance variable or imported pool variables may be presented as an alternative

---

[9] This means that a subsystem may add new methods to a class defined in another subsystem.

[10] Including method definitions in subclasses of the imported class.

[11] Variable names are included within a class usage specification to enable the compilation of methods in the absence of a binding for the imported class. The distinction between imported instance variables and imported pool variables is a reflection of currently implementation technology and probably should be eliminated.

[12] This assumes that some protocol is actually defined for pools. The X3J20 draft does not currently define any such protocol; however, for backwards compatibility we probably still need to recognize pools as objects.

for resolving references to undeclared variables.

An implementation may require that an actual binding for an imported class exist at development time in order to compile methods that reference imported instance or pool variables.

When inspecting objects, the names of classes may need to be displayed with a subsystem name qualification in order to disambiguate them.

Subsystems might be presented to the user in a manner similar to Smalltalk-80 projects (change “desktop” when switching between subsystems). More likely, browsers would be modified such that an individual browser is opened on a particular subsystem and only presents the classses, globals, and imported names that are defined by the subsystem.

Message Selector Conflicts

To this point we have only addressed the issue of independent name spaces for the global identifiers used within Smalltalk programs. Name conflicts can also occur in the domain of message selectors.

The most common such conflict occurs when two different logical subsystems both attempt to independently extend an imported class with a new method and coincidentally choose the same message selector name. Under Team/V this would be detected as a method definition conflict and the two subsystems could not be used together. In other development environments one of the subsystems would end up using the wrong method with probable disastrous results.

What is a Message Selector?

In classic Smalltalk-80 implementations message selectors are strings and the selection of a method is accomplished by doing a string comparison[13] between the selector used in a message and the selector of each method supported by the receiving class. The various forms of message selector conflicts arise because of the universe of strings is a flat, unpartitioned space. Any two strings containing the same character sequence will compare equal no matter when or where the string was created.

One solution to this problem is to modify the definition and implementation of Smalltalk such that it treats the *value* of a message selector as a distinct concept from the identifier that is used to refer to the message selector. This is analogous

[13] Implementations actually use instances of class Symbol and symbol comparison for message selectors but in this context symbols are just an optimized representation for strings.

to the manner in which variables are treated in most programming languages. The name of a variable is separate and distinct from the actual storage cell that is accessed using the variable name. Scoping mechanisms allow a variable name to refer to different storage cells in different contexts and allow a single cell to be referenced using different names in different contexts (consider for example, reference parameters in some programming languages). A similar treatment of message selectors can be accomplished by separately defining the concepts of message *selector values* and symbolic message *selector names*

A message **selector value** is the key that is used to actually identify methods. A message send uses a message selector value to select a method whose selector value is the same as that specified in the message. The actual representation of message selector values is a detail of the implementation and is transparent to the Smalltalk programmer.

A message **selector name** is the actual symbolic name that is used in a Smalltalk program to refer to a message selector value. A message selector name is what is actually written by a Smalltalk programmer when coding a message send expression, defining a literal selector (e.g. #foo), or in coding the message pattern in a method definition. Reference to a message selector name in a program implies the use of whichever message selector value is bound to the symbolic selector name at the point of reference.

Selector Name Scopes

Once selector names have been made separate and distinct from selector values the bindings of selector names to selector values can be organized into selector name scopes (name spaces) in much the same way as the bindings for variable names to variable cells are organized into variable scopes. Different parts of a Smalltalk program can then have access to different selector scopes. Selector scopes can also be organized into hierarchies that permit redefinition and shadowing just like variable scopes. When the Smalltalk compiler processes a message expression it looks up the message selector name in the current selector name scope (and if necessary any enclosing scopes) and uses the selector value that is found.

Because selector names are now managed in the same manner as variable names the same mechanisms that are used to control visibility and access of variable names in subsystems and classes can be applied to controlling the visibility of selectors.

The **universal selector scope** is a selector name scope that encompasses all other selector name scope. By default, all message selectors are defined in the universal selector scope; however, subsystems and classes may explicitly define selectors in a local (private) selector scope that shadows the universal scope.

Because of the default definition of selectors in the universal scope everything "works normally" in the absence of the use of explicit selector scopes. In this situation all selectors are defined in a single, flat namespace and behave exactly like traditional Smalltalk-80.

Subsystem Selector Scopes

Each subsystem may have its own selector scope that shadows the universal selector scope. A selector is created in a subsystem scope by declaring it in a new type of definition, a *selector definition*. If a selector is defined in a subsystem selector scope then any reference to the selector name from code defined within the subsystem will use a selector value that is known only to that subsystem. Any method defined using that selector name within the subsystem may only be invoked from within the subsystem.

The following example illustrates a possible use of subsystem private selector spaces. The syntax is hypothetical for purpose of the example. Message selector names defined in the universal selector space are shown in a normal font while selectors defined private to a subsystem appear in a bold font with a subscript that identifies their defining subsystem:

```
1       Subsystem A
2               private selector privateNew_A.
3               Object subclass: MyClass
4                       class method
5                               new
6                                       ^self privateNew_A
7                       class method
8                               privateNew_A
9                                       <primitive: ...> "create an instance"
10              ... := MyClass privateNew_A
11
12      Subsystem B
13              Import MyClass  "from subsystem A"
14              ...
15              ... := MyClass new .   "creates an instance of MyClass"
16              ... := MyClass privateNew.  "message not understood"
```

Line 2 defines #privateNew as selector that is private to subsystem A. Any use of #privateNew within subsystem A will use the selector value that is private to the subsystem, all other selectors in subsystem A use selector binding in the universal selector space. For example line 5 specifies #new as the selector for a method definition. Because this selector is not explicitly defined as being private to the subsystem it will be defined in the universal selector scope (assuming it has not already been defined there). This is in contrast to the specification of

#privateNew in line 8 as the selector for a method definition. In this case, because #privateNew has been declared as private to Subsystem A, the private selector value will be used as the key to the method. Messages sent to MyClass using the selector value for #privateNew from the universal scope would not find the method defined in lines 8-9. However any message send, such as those in lines 6 and 10, within the scope of the subsystem A's definition of #privateNew will invoke the method defined in lines 8-9.

Subsystem B does not define any selectors that are local to the subsystem thus all selectors within the subsystem will be bound using the universal selector space. When the expression in line 15 is evaluated the universal value of selector #new will be used to perform a message lookup in MyClass. The message lookup will find the method defined in lines 5-6 because that method is defined using the universal value of selector #new. Execution will proceed with the execution of line 6 and a message send using subsystem A's private selector value for #privateNew. The message lookup will find the method defined in lines 8-9 because that method is also defined using subsystem A's private value for #privateNew. Note that subsystem A's private selector value is used even though the send of #new originated from subsystem B. Binding of selector names to selector values is based upon static scoping at compile time, not upon dynamic execution time scoping.

The effect of the expression of line 16 is quite different. It begins by using the universal value of selector #privateNew to perform a message lookup on MyClass. The method definition in lines 8-9 will not be found because it was defined using Subsystem A's private value for #privateNew. Assuming no superclasses of MyClass define a method using the universal value of #privateNew a message not understood exception will result.