An Overview of Modular Smalltalk

Allen Wirfs-Brock
(503) 242-0725

Instantiations, Inc.
1020 SW Taylor St., Suite 200
Portland, OR 97205

Brian Wilkerson
brianw%spt.tek.com@relay.cs.net
(503) 627-3294

P.O. Box 500, Mail Sta. 50-470
Tektronix, Inc.
Beaverton, OR 97077

Abstract

This paper introduces the programming language Modular Smalltalk, a descendant of the Smalltalk-80 programming language. Modular Smalltalk was designed to support teams of software engineers developing production application programs that can run independently of the environment in which they are developed. We first discuss our motivation for designing Modular Smalltalk. Specifically, we examine the properties of Smalltalk-80 that make it inappropriate for our purposes. We then present an overview of the design of Modular Smalltalk, with an emphasis on how it overcomes these weaknesses.

Introduction

Modular Smalltalk is an evolution of the Smalltalk programming language and system designed to support teams of software engineers developing production application programs that can operate under the control of standard operating systems and display environments.

The Smalltalk programming language and system was originally intended to be the software component of the Dynabook, a portable personal information management tool [Kay77a, Kay77b]. As described by one of its developers, its purpose was "to support children of all ages in the world of information" [Inga78]. Smalltalk is a uniformly object-oriented system which integrates a programming language and its implementation, development tools for the language, a window-oriented user interface manager, and other system software services. The development of Smalltalk was an evolutionary process which took place over an extended period [Inga83]. Its developers typically built a version of the system, experimented with it, and finally used what they learned to build the next version upon the base of the current version. The final result of this process was the Smalltalk-80™ system [Gold83, Gold84].

As Smalltalk became available to a broader group of users, it first found acceptance as a rapid prototyping system. The fact that Smalltalk proved to be an excellent prototyping tool should not be surprising, as Smalltalk's developers had themselves used the system in this manner. However, outside of research laboratories, prototypes are not viewed as ends unto themselves but rather steps on the path towards the development of a final product or solution. The success of prototype applications developed using Smalltalk has led many Smalltalk programmers to look for ways to develop and deliver the final production versions of applications using Smalltalk. These attempts have so far had only limited success.

While we have have found the Smalltalk language a very effective tool for building complex systems, we believe it is currently unrealistic to expect that an excellent rapid prototyping system can also be an excellent application delivery system. If Smalltalk is to be widely used to develop production applications, then the language, its development environment, and its implementations need to be re-engineered for that purpose. Modular Smalltalk is an offshoot of the Smalltalk language which is specifically designed to support the engineering of production application programs. Modular Smalltalk will ultimately consist of a formal language specification, an essential module and class specification, an incremental development environment, and a production-quality compiler. This paper addresses the architectural features of the Modular Smalltalk language.

In the following section we will discuss specific difficulties of conventional Smalltalk implementations for producing production applications. We then present the design goals of Modular Smalltalk. Finally we present a general overview of the Modular Smalltalk language.

Smalltalk-80 as a Production Tool

As a tool for software development, Smalltalk-80 has some serious drawbacks. These include

- the image paradigm,
- a confusion between the language and its programming environment,
- a confusion between the language definition and its implementation,
- the ability to learn the system, and
- its performance.

The Image paradigm

Conventional Smalltalk systems are built around a virtual image [Gold83]. A *virtual image* is the dynamic data structure representing all code and data for the running Smalltalk system. This includes not only the application data, but the application programs themselves. It also includes the tools (compiler, editors, debuggers) for building programs, and for basic system support facilities such as window and file management. All code and data are represented as objects described by classes which are themselves objects. Both application and system objects share common class definitions. Applications are built by incrementally adding or modifying class definitions within a running image. A Smalltalk application "program" is in effect a set of edits to some baseline virtual image.

However, applications are not best defined as a set of incremental changes to some base environment. Other programmers may have also made a set of changes to their environments. In practice, it proves impossible to predict when two applications, two sets of independently made changes, will conflict. Conflicts can also occur during the engineering of a single application. Engineers working together can never be sure when the changes made by one will conflict with the changes made by another. The ability to refine existing programs is similarly hampered.

Furthermore, because any application can access and modify any class or object in the entire system, all dependencies between an application and other features of the system are implicit. It becomes difficult to prove that any given feature of the system is not used. It therefore becomes very difficult to extract the application from the surrounding system.

Also, every aspect of the system is open to inspection and modification. Changes made by an application for its own purposes can effect the operation of the system. For example, an application can modify a class in such a manner that it unintentionally introduces errors into the compiler or the debugger.

This can also cause problems for the distribution and continued support of applications. If users may change anything about their own systems, it becomes impossible to support an application that depends upon certain features of that system remaining stable. Similarly, user applications are vulnerable to the effects of system revisions by the system developer.

Another result of the image paradigm is that applications may rely upon a part of the state of the image for which there is no code to reproduce the state. For example, a programmer might execute code in a workspace to initialize the state of an object without including that code in the application. If the application was loaded into another image, there would be no mechanical means to perform the appropriate initializations

Confusion between language and programming environment

With conventional Smalltalk implementations, no clear distinction exists between the language and the environment in which programs are written. No distinction is made between the language and the standard set of abstractions (classes or components). No distinction is made between the standard set of abstractions and the implementation of the environment.

Because of these confusions, it becomes effectively impossible to build a new implementation optimized for a different purpose, for example the delivery of stand-alone applications instead of a system for rapid prototyping. Features implemented for the sake of the programming environment are used by programs, compounding the confusion so that neither implementor nor programmer can say which features of the environment are essential to the language, and which are just quirks of the implementation of the environment. If implementors need to reimplement the system anyway, they are forced to guess. For example, is the ability to dynamically add a method to a class a convenience of a programming environment designed for rapid prototyping, or is it an essential feature of the language definition?

Confusion between language definition and implementation

One of the main principles of object-oriented programming has been said to be separating the "what" from the "how" [Robs81]. But the Smalltalk-80 language fails to make this distinction at the grossest level. If the "what" is taken to be the language specification, and the "how" is taken to be the language implementation, Smalltalk-80 fails to separate these. For example, one would like a clear distinction between what it means to send a message, and a particular implementation of message-sending. No such distinction has ever existed for Smalltalk-80. This confusion even extends to the terminology used by Smalltalk programmers to describe the behavior of programs. Smalltalk programmers may be frequently heard speaking of "message lookup" but seldom, if ever, use phrases such as "message binding" or "message resolution" to describe the activity of sending a message to an object. Because of this confusion between language definition and implementation, a programmer can write an application that depends upon representational or algorithmic details of the canonical implementation of the virtual machine. All implementations of Smalltalk-80 must mimic these details, regardless of the effect upon the efficiency of the implementation. At best an implementation may "cheat but not get caught" [Deut86].

The Smalltalk-80 system was originally designed as a self-hosted, incremental online program development environment. As such, not only was it required to support dynamic modification of a running program but the image implementation model required such modification be accomplished using reflective operations upon the running system. A program may make use of these same operations to dynamically modify itself. While a handful of experimental applications have made effective use of the reflective characteristics of Smalltalk-80 [Born81], the vast majority of applications do not. Should reflectiveness be considered an essential characteristic of Smalltalk, or an implementation detail? Must all implementations, including those targeted for ROM-based embedded systems, support reflective operations?

Learnability of the system

The Smalltalk-80 system is notably difficult to learn [Born87]. The size and complexity of the system (hundreds of classes and thousands of methods) alone serves as a formidable impediment. In addition, various conceptual difficulties can hamper the novice, such as metaclasses, or the distinction between methods and blocks. Other conceptual difficulties spring from the confusions discussed above. If Smalltalk is to become a widely used system, these problems need to be addressed.

Performance

Existing Smalltalk-80 systems are slower and less efficient than more conventional languages, even other dynamic languages such as LISP. Present systems are optimized for incremental rapid prototyping systems, but they are not optimized for execution speed.

While the poor performance of Smalltalk implementations is frequently attributed to the dynamic binding of procedure names to procedure implementations (message send overhead), commonly known techniques allow dynamic message binding to be only slightly more expensive than a standard procedure call. A much more severe performance problem comes from the inability of a Smalltalk compiler to perform *any* significant local or global optimization. This is a direct consequence of the

reflective nature of the language. Given the possibility of reflective operations, it becomes impossible for a compiler to reason definitively about a program and hence perform any optimizations.

For example, it might be reasonably expected that the binding of message-sends to self could be statically resolved since the class of self can be determined at compilation time*. Given such a static binding, it should be possible to inline expand the target method and then apply standard local optimization techniques. Unfortunately, such optimizations prove to be impossible since the program may arbitrarily modify any of its methods. At best, an implementation could attempt to maintain both optimized and unoptimized representations of each method and switch representations if a reflexive operation is performed.

As another example, consider that it is impossible analytically to remove classes or methods from an image. Even if there are no lexically apparent references to a class or a message selector, a program may dynamically construct a new method which references them.

Modular Smalltalk Design Goals

To correct these deficiencies, we are designing a new generation of Smalltalk that we call Modular Smalltalk. Modular Smalltalk departs from Smalltalk-80 with the addition of a module facility and the elimination of all reflexive operations. In addition, we have sought to clarify and modify the definition of Smalltalk-80 to make it more semantically consistent. The module facility supports the encapsulation and hiding of classes. Modular Smalltalk is designed to support the development of separately deliverable applications — an implicit goal of most software engineering efforts.

Modular Smalltalk is an object-oriented programming language. Using Modular Smalltalk, programs can be developed within an interactive development environment similar to that of Smalltalk-80. Programs developed within the Modular Smalltalk environment, however, can then be delivered as stand-alone applications.

Modular Smalltalk differs from other proposals to modify or extend Smalltalk-80 such as Deltatalk [Born87] in that it specifically addresses the problems of building stand-alone production programs instead of enhancing Smalltalk's utility as an exploratory programming system. A principal goal of Modular Smalltalk is to maintain a clear distinction between the language specification, its implementations and its development environment. In addition Modular Smalltalk seeks to:

- support the development of application programs that execute independently of their development environment,
- allow for team engineering efforts,
- provide consistent and explicit semantics, independent of any implementation,
- allow for the possibility of efficient implementation,
- be a recognizable descendant of Smalltalk-80, thereby allowing existing Smalltalk programmers to master it quickly, and
- be simple enough for new Smalltalk programmers to learn easily.

The Modular Smalltalk Language

Modular Smalltalk defines a language with a specification that is independent of its implementation. The semantics of Modular Smalltalk allow for many varying efficient implementations. Modular Smalltalk follows the commonly understood syntax and semantics of Smalltalk-80, but differs in the following major respects:

- The language is oriented towards the construction of programs, which are stand-alone entities.
- Programs consist of modules. Modules provide the units to divide the functional and organizational responsibility within a program.
- Modules encapsulate class definitions and other constants.
- Class definitions are static, declarative syntactic structures.
- Modular Smalltalk is not reflexive.

* This requires the additional optimization that all inheritance issues be resolved at compilation time, and this optimization is itself greatly complicated by Smalltalk's reflectiveness.

In addition, Modular Smalltalk augments Smalltalk-80 in several ways. It provides explicit syntactic support for practices that have heretofore been merely commonly used programming conventions. It also addresses several commonly recognized deficiencies, especially the absence of multiple inheritance.

Programs

A program in Modular Smalltalk is the unit that defines an independent application. A program defines the classes of all objects used within an application. It also defines the sequence of actions performed with instances of those classes when the program is executed.

A program is a collection of modules. One module is the main module of a program. For example, a program to play the game of blackjack might consist of modules implementing the blackjack game itself, playing cards, user interface components, graphics packages, data structures, random number generators, and the kernel classes required by all Modular Smalltalk programs.

Modules can depend upon definitions from other modules, but no module can depend upon the main module. The dependencies of modules within a program form a directed, acyclic graph with a single root node (the main module). In the blackjack example above, the blackjack module would be the main module and would depend on definitions from the playing card module, among others.

When a program is executed, modules are initialized in the order given by a depth-first traversal of the dependency graph. No module is ever initialized more than once.

The objects upon which a program operates exist only within the context of the executing program. That is, independent executions of the program would operate upon distinct sets of objects. The classes that a program defines or uses and the objects that a program manipulates are separate and distinct from the classes and objects used to construct the development environment for Modular Smalltalk.

Modules

Modules are program units that manage the visibility and accessibility of names. A module defines a set of constant bindings between names and objects. Modules are not objects and have no existence (representation) during the execution of a program. A module typically groups a set of class definitions and objects to implement some service or abstraction. A module will frequently be the unit of division of responsibility within a programming team.

A module provides an independent naming environment that is separate from other modules within the program. A module consists of a sequence of *named object* definitions. A named object definition introduces a static binding between an identifier (a name) and an object. Because the binding is static, the named object may be used as a constant value within expressions. Named objects may not be the target of an assignment statement. Modular Smalltalk has no global variables. All mutable global state is encapsulated within objects. Where Smalltalk-80 makes extensive use of global variables to name the classes and utility objects that make up a program, Modular Smalltalk uses named objects.

In the blackjack example, the playing cards module might define the classes Card and CardDeck, as well as the constant collections of Ranks and Suits. An example implementation is given in the appendix.

Names must be uniquely defined within a module: no name clashes are allowed. Because a naming conflict is one form of conflict possible when teams of engineers work together on a program, modules support team engineering by providing rigorously isolated name spaces.

A module controls the visibility of named objects. Principles for the management of names follow the commonly accepted principles for the management of separate name spaces, as exemplified by languages such as Modula-2 [Wirt84] or Ada [Booc83].

Definitions

There are two ways to introduce a named object binding into a module.

- The binding may be defined locally by the module, or
- it may be imported from another module.

A local definition consists of either:

- an expression whose value is the object to which the name is bound, or
- a class definition.

Imports

Modules can import other modules. Imported modules introduce additional bindings. Imported bindings are specified by naming the module in which the binding is available, and the desired named object.

Modules must specify explicitly which other modules, and the bindings within them, they wish to import. When a module imports another module, it implicitly limits which object names are imported. Unless a module specifically requests an object named Belshazzar, for example, it will not import that object. A module, therefore, consists of:

- a set of bindings between names and objects, and
- a declaration of other bindings to import.

As stated earlier, all names, whether local or imported, must be unique within a module. Nevertheless, sometimes a module may require the import of an object with a duplicate name from another module. A renaming mechanism allows for the resolution of such name clashes. When a module imports another module, it may rename any object it imports. It may specify, for example, that it import an object named Joe as Joseph.

The management of libraries of modules is considered to be outside of the scope of the Modular Smalltalk language definition. Modules will be the most common unit of code reuse by Modular Smalltalk programmers. A typical implementation would include a module library manager that would manage a repository of a wide selection of modules and versions of modules. In order not to place undue constraints on the design of such library managers, Modular Smalltalk module names are literal strings whose interpretation is implementation defined.

Visibility

A module also incorporates a mechanism to make named objects selectively available to users of the module. Each named object, in addition to binding an object with a name, includes a visibility attribute allowing the programmer to specify whether the binding is to be public or private.

Names defined in a module are visible throughout that module. Names defined in a module are not visible outside that module unless the programmer explicitly specifies otherwise. If a named object is specified to be public, it is exported and may be made visible to another module. Even so, the named object is not visible to another module unless the other module specifically imports the module containing it, and the named object itself.

The ability to specify that a given name is private to a module provides explicit syntactic language support for information-hiding. This enhances the maintainability of an application by making explicit all dependencies.

Because all dependencies of a program must be explicit to support its delivery as a separate application, Modular Smalltalk has no global object naming space. No objects are implicitly available to all modules. Objects are available to a module by name from two sources only:

- the named objects defined within the module itself, and
- named objects declared public within imported modules, and explicitly requested by the module.

Modules control the accessibility of the *names* of objects, not the objects themselves. Objects may be freely passed as values between methods defined in different modules regardless of whether the object's name or its class name is visible in either module. While modules restrict the visibility of class names they do not restrict the use of message selector names. A program includes a single program-wide name space for message selector. A mechanism other than modules, discussed in the section entitled **Methods**, is provided to restrict selector usage.

Classes

In many ways, classes in Modular Smalltalk are quite similar to their Smalltalk-80 counterparts. There are certain crucial differences, however.

- Class definitions are static.
- Multiple inheritance is supported.
- Modular Smalltalk has no metaclasses.
- Encapsulated state is uniformly accessed using message selectors instead of variable names.

Class Definitions

A class definition is not an object and has no existence during the execution of a program. A class definition is also not an expression. New classes cannot be created using message-sends.

Instead, a class definition is a static description of the behavior* of a group of objects. Naturally, class definitions can be manipulated within a development environment. However, because Modular Smalltalk is not a reflexive language, a running program may not modify its class definitions.

A class definition defines two sets of behavior:

- the behavior of instance objects, and
- the behavior of a unique class object.

A *class object* is a named object introduced by a class definition. It is the object bound to the name associated with the class definition. Class objects in Modular Smalltalk provide a place to define behavior such as instance creation methods.

A class definition is considered to define the behavior of the associated objects completely. There are two ways a class definition may specify its object's behaviors:

- the behavior may be inherited, or
- the behavior may be locally defined as part of the class definition.

Behavior may be inherited from zero or more other class definitions (the class' superclasses). Even though behavior is inherited, it is still considered to be part of the inheriting class' definition. Modular Smalltalk does not imply or require any sort of dynamic superclass message lookup algorithm. Dynamic lookup remains a valid implementation technique, one that is especially useful in incremental development environments. Modular Smalltalk specifies the effect of sending a message, not the mechanism for sending a message.

Local behavior definitions consist of a set of mappings between message selectors and their implementations. A local definition may mask or override an inherited definition. Each selector has a visibility attribute; it may be declared public or private to its class.

Encapsulated state is declared and inherited as a special case of method definition.

Variables

Instance variable names do not exist within Modular Smalltalk. Instead, instance variables are referred to using accessing or modifying messages. For each instance variable defined by a class, two accessing methods are defined: one to set the value of the variable, and one to retrieve to value of the variable. Variable state can only be accessed or modified using message sends that invoke the accessing methods.

An accessing method only stores or retrieves the value of its associated variable. It performs no other computations. For example, one could use the unary selector `getY` to access an object's instance variable. One could also use the keyword selector `setY:` to modify the instance variable. The accessing and modifying protocols need not be lexically similar.

Referring to variables entirely through the use of accessing protocol makes it easier to reuse existing code by subclassing [Wirf88]. It also simplifies the semantics of multiple inheritance, as the semantics of variable inheritance is exactly the semantics of method inheritance.

Unlike Smalltalk-80, which defines six or more different types of variables that may appear on the left hand side of an assignment operator, Modular Smalltalk defines exactly one type, block temporaries. All other variable state is modified using message syntax.

Multiple Inheritance

Modular Smalltalk supports multiple inheritance. Multiple inheritance provides the ability to break free of a rigid, hierarchical view of the world. It allows programmers to specify that instances of a given class behave a great deal like instances of another class, but also share aspects of their behavior with a third, unrelated class. This mechanism is sufficiently compelling that Smalltalk-80 has tried to incorporate it [Born82, Wilk86]. Problems arise with multiple inheritance, however, when a class tries to inherit from two or more superclasses that contain conflicting method or variable definitions.

* By *behavior* we mean the complete set of message selectors recognized by an object and their associated method definitions.

Modular Smalltalk addresses the problem of conflicting methods in the following manner. A class may have any number of superclasses, including none. A class inherits behavior equally from all of its immediate superclasses (some of which may itself be inherited behavior). There is no order dependency among superclasses.

It is an error for a class to inherit two different definitions for the same message selector. Such an error can be avoided by explicitly redefining the conflicting selector in the class itself. Notice, however, that inheriting the same method definition (one defined in a single lexical location) from multiple superclasses is not an error. Because instance variables are specified in terms of message selector definitions, the rules for variable inheritance and conflict resolution are exactly those that apply to any other methods.

Metaclasses

In Smalltalk-80 all objects must be an instance of some class. The class defines the behavior of its instances in terms of the message lookup algorithm. Since classes are themselves objects, they too must be instances of a classes. Metaclasses are the classes of which each class is an instance.

In Modular Smalltalk, the behavior of every object is specified by a class definition. This includes both instance and class objects. Class definitions are not objects. Therefore the behavior of an object is not dependent upon any other object and hence an object does not need to be an instance of a class. Specifically, class objects do not need a class. Therefore, Modular Smalltalk needs no metaclasses.

Class objects have the ability to instantiate the instance objects described by their class definition. Class objects are instantiated as part of the module initialization process. The standard definition of the class message in Modular Smalltalk is to return the class object defined by the class definition that describes the behavior of the receiver. Thus the message class sent to an instance object will return its associated class object and the class message sent to a class object will return that same class object.

Metaclasses are one of the features of Smalltalk-80 that make it difficult to teach and understand [Born87]. The Modular Smalltalk model of class and instance objects is a direct reflection of the class behavior/instance behavior model presented by the standard Smalltalk-80 browser. Metaclasses are one implementation of this model, one that is difficult to understand. Because the semantics of classes and instances are defined independently of any implementation, Modular Smalltalk is easier to learn and use.

This model of a syntactic class definition which defines the behavior of both class and instance objects is essentially the same as used by the Objective-C language [Cox86]. Objective-C uses the term *factory object* for class objects.

Class Extensions

Class extensions provide the ability for a module to add protocol to existing classes defined outside the module. This mechanism is another case where the

Modular Smalltalk programming language provides explicit syntactic support for a common object-oriented programming convention, that of specifying a default behavior for all objects.

Extensions provide the ability to encapsulate behavior supporting a function that may be common to many classes, spread across several modules. For example, an application might require that objects of a wide variety of classes be able to store themselves in a file. Just such a system has been implemented for Smalltalk-80 [Vegd86]. It is reasonable to assume that most of these classes are defined in modules other than the module of the application requiring this ability. Let us further assume that none of these classes have the desired ability.

Because this ability is required for a wide variety of classes, very different methods must be used to store the different kinds of instances. Using the mechanism of class extensions, it is possible to define a module to provide the needed storage functionality. This module could define extensions to all the classes that require the functionality. Each class extension would consist of the small number of methods required to implement the functionality. In this way, the storage module becomes a component that can be included in any application requiring this functionality.

Class extensions can only add behavior. They cannot modify or remove behavior.

Methods

Method definitions associate a message selector with an implementation.

Unlike Smalltalk-80, Modular Smalltalk message selectors are not instances of class Symbol (for example, they are not synonymous with their textual

representation). Instead they are instances of class MessageSelector. Message selectors may not be dynamically constructed and specifically, strings may not be dynamically converted into message selectors. Message selectors still have a literal representation, so that one can, for example, use:

```
anArray perform: (#(#first #second #third) at: n)
```

Disallowing the dynamic construction of message selectors allows an implementation to compute for a program the set of defined but unused selectors. Code need not be generated for such selectors.

Implementations

The implementation of a method can be either

- a literal block,
- the keyword primitive,
- the keyword abstract, or
- the keyword undefined.

The ordinary implementation of a method in Modular Smalltalk is a block which is evaluated when a message is sent. Modular Smalltalk thus simplifies Smalltalk-80, unifying the semantics of block and method evaluation.

Because methods are blocks, the default value returned from a method is the value of the last expression in the block. The default value returned from a method is not self, as it was in Smalltalk-80.

Blocks in Modular Smalltalk can declare temporary variables. Blocks can be lexically nested with properly nested variable scope, and blocks are re-entrant. This means that separate invocations of the same block do not share the same state for block arguments and temporaries, thus allowing two or more executions to overlap in time.

The implementation primitive means that the associated method definition is fully specified by either the language definition or the implementation. The method is not specified by Modular Smalltalk code. Unlike Smalltalk-80, a primitive number is not associated with a primitive specification. The class name along with the message selector is sufficient to uniquely identify the primitive. A special primitive failure mechanism and associated Smalltalk code is not used. Instead, each primitive's specification fully defines its behavior, including error conditions. This behavior may include sending other messages. For example, an integer division primitive specification might specify that if the divisor is zero, the message zeroDivide would be sent to the receiver of the divide message.

Whenever possible, the syntax of Modular Smalltalk seeks to support what have up to now been only programming conventions. The method implementations abstract and undefined are an example of this support.

Defining a selector as abstract means that subclasses must provide a definition for the method. It is the equivalent of the Smalltalk-80 convention self subclassResponsibility. A class that includes abstract methods, either locally or through inheritance, is an abstract class and cannot be instantiated.

If a class inherits an abstract method from one of its superclasses, the abstract implementation will not conflict with any nonabstract implementation inherited from any other superclass. Abstract methods cannot cause method definition clashes.

Defining a selector as undefined removes the inherited selector from the behavior of the object. It is the analog of the Smalltalk-80 convention self shouldNotImplement.

Visibility

Each method includes a visibility attribute; it is either a public or a private method. The default is public. Information-hiding, the encapsulation of implementation details private to an object, is a key principle of object-oriented programming. Therefore, private methods have long been a convention of Smalltalk-80. Modular Smalltalk provides support for this convention by building it into the semantics of the language.

When sent to an object, a private selector is understood if the class description of the receiver is the same as the class description of the sender.

Private selectors are inherited by subclasses, which can therefore use them. The visibility of a message selector (whether it is public or private) is inherited separately from the body of the method. Subclasses can inherit a method body while overriding its visibility attribute.

Conclusion

We believe that Modular Smalltalk will prove to be an effective tool for the construction of complex applications. It is easier for novices to learn, while remaining similar enough that current Smalltalk programmers will be able to learn it swiftly and easily. In addition, it supports current software engineering practices for the following reasons:

- It allows the delivery of stand-alone applications.
- It allows the protection of proprietary source code.
- It supports change management and version control, so that teams of programmers can work together on a large project without collisions.
- It allows for varying implementations, each of which can be optimized for different purposes.

Acknowledgements

Many of the ideas incorporated into Modular Smalltalk grew out of a long series of language design discussions with Will Clinger, Ralph London and Steve Vegdahl. Mark Ballard was a major contributor to the initial language design and along with Brian Wilkerson built the first experimental implementation. Kit Bradley has provided continuing managerial support for what has often appeared to be a radical project. Finally, Lauren Wiener helped us make this paper and the preliminary language specification real.

References

[Booc83] Booch, Grady, *Software Engineering with Ada*, Benjamin/Cummings, Menlo Park, CA. 1983

[Born81] Borning, Alan H., "The Programming Language Aspects of ThingLab," *CM Transactions of Programming Languages and Systems*, 3(4), pp. 353-387, Oct. 1981.

[Born82] Borning, Alan H., and Daniel H. H. Ingalls, "Multiple Inheritance in Smalltalk-80," *AAAI Proceedings*, 1982, pp. 234-237.

[Born87] Borning, Alan, and Tim O'Shea, "Deltatalk: An Empirically and Aesthetically Motivated Simplification of the Smalltalk-80 Language," *ECOOP Proceedings*, 1987.

[Cox86] Cox, Brad J., *Object-Oriented Programming An Evolutionary Approach*, Addison-Wesley, Reading, Massachusetts, 1986.

[Deut86] Deutsch, L. Peter, private communication

[Gold83] Goldberg, Adele, and David Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading Massachusetts, 1983.

[Gold84] Goldberg, Adele, *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, Reading, Massachusetts, 1984.

[Inga83] Ingalls, Daniel H. H., "The Evolution of the Smalltalk Virtual Machine", *Smalltalk-80: Bits of History, Words of Advice*, ed. Glenn Krasner, Addison-Wesley, Reading, Massachusetts, 1983.

[Kay77a] Kay, Alan C., "Microelectronics and the Personal Computer," *Scientific American*, September 1977, pp. 230-244.

[Kay77b] Kay, Alan C. and Adele Goldberg, "Personal Dynamic Media," *Computer*, March 1977, pp. 31-41.

[Robs81] Robson, David. "Object-Oriented Software Systems", *Byte*, August, 1981, pp. 74-86.

[Vegd86] Vegdahl, Steven R. "Moving Structures between Smalltalk Images," *OOPSLA Proceedings*, 1986, pp. 466-471. Also published in *SIGPLAN Notices*, vol. 21, no. 11, November 1986, pp. 466-471.

[Wilk86] Wilkerson, Brian C., *Inheritance Mechanisms for Smalltalk-80*, Technical Report CR-86-57, Tektronix, Inc., Beaverton, Oregon, August 1986.

[Wirf88] Wirfs-Brock, Allen, and Brian C. Wilkerson, *Variables Limit Reusability*, Technical Report SPT-88-07.

[Wirt84] Wirth, Niklaus, "Programming in Modula-2", *Texts and Monographs in Computer Science*, 2nd ed. David Gries, Springer-Verlag, Berlin 1984.

Appendix

This appendix contains example code for the module PlayingCards. The code below is formatted in an informal publication syntax.

```
Module 'PlayingCards'
    "This module defines four named objects –
    CardSuits, CardRanks, Card and CardDeck
    – that are used to implement the functionality
    of a deck of playing cards. Only the class
    CardDeck is exported."

  imports Object from 'Kernel'
  imports List from 'Collections'
  imports UniformDistribution from
  'ProbabilityDistributions'

  CardSuits -> #('heart' 'club' 'diamond' 'spade')
    "The symbol '->' means 'is defined as'."

  CardRanks -> 1 to: 13

  Card -> Class
    refines Object

  instance behavior

    accessing

      variable suit suit: (private)
            "Answer and set the suit of the
      receiver. The suit should be an element of
      <CardSuits>."

      variable rank rank: (private)
            "Answer and set the rank of the
      receiver. The rank of jacks, queens and
      kings is 11, 12 and 13, respectively."

      value
        "Answer the face value of the receiver."

        ↑self rank min: 10

    testing

      = aCard
        "Answer <true> if the receiver represents
      the same card as <aCard>."

        ↑self suit = aCard suit
          and: [self rank = aCard rank]

  class behavior

    instance creation

      suit: suitName rank: rankIndex
        "Answer an instance of the receiver
      whose suit is <suitName> and whose rank
      is <rankName>."

        | card |
        card := self new.
        card suit: suitName.
        card rank: rankIndex.
        ↑card

CardDeck (public) -> Class
    refines Object

  instance behavior

    accessing

      variable cards (private) cards: (private)
            "Answer and set the ordered
      collection of cards remaining in the
      receiver."

      initialize (private)
        "Initialize the receiver."

        self cards: List new

      addCard: aCard (private)
        "Add <aCard> to the receiver."

        self cards add: aCard

      deal
        "Deal the top card off of the receiver."

        ↑self cards removeFirst

      shuffle
        "Shuffle the cards remaining in the
      receiver."

        | random |
        random := UniformDistribution from: 1 to:
      self cards size.
        1 to: self cards size
          do:
            [:source |
            | target temp |
            target := random next.
            temp := self cards at: source.
            self cards at: source
              put: (self cards at: target).
            self cards at: target put: temp]
```

class behavior

instance creation

new

"*Answer an instance of the receiver containing all 52 standard playing cards.*"

```
| deck |
deck := super new initialize.
CardSuits
  do:
    [:suit |
    CardRanks
      do:
        [:rank |
        deck addCard: (Card suit: suit
rank: rank)]].
↑deck
```