Everything is an object within Smalltalk

Ernest Micklei

ELC Information Technology

Smalltalk was designed to enforce an object-oriented approach, in which programs are divided into self-contained packages of data and behavior. Every bit of code must be defined as the responsibility of a particular class, and every class inherits properties from those above it in the hierarchy. Smalltalk is a pure object-oriented, untyped programming language that provides single inheritance, polymorphism and dynamic binding for implementing any object. Learning the syntax of the language takes an afternoon, but learning the way around the Smalltalk class library can be a daunting task.

In Smalltalk, everything is an object [1]. There is no way to create, within the language, an entity that is not an object. All actions in a Smalltalk system are produced by passing messages. A message is a request for an object to perform some operation. A message can contain argument values for executing the requested operation. For instance, the expression "1 + 2" should be read as "object 1 receives the message named + with argument the object 2". Normally, it responds with an integer object 3. Every expression or statement can be explained by a sequence of message-sends to objects. The C statement

```
if (x<0)
        doLessX();
else
        doNotLessX();
```

translates in Smalltalk to

```
x < 0
        ifTrue:        [ self doLessX ]
        ifFalse: [ self doNotLessX ].
```

Ignoring the syntax differences, the Smalltalk version conforms to the message passing principle and should be read as follows. Send the object referenced by x the message < with argument object 0. This message responds with a boolean object (true or false). Next, the boolean object is send the message <ifTrue:ifFalse:> which has two block arguments (the expressions which are enclosed by squared brackets [...]). The behavior of the boolean objects provide definitions for this message. For instance, the object true, which is the sole instance of class True, implements <ifTrue:ifFalse:> in which only the first block argument will be evaluated. False, which is the class of the object false, implements the same message but evaluates the second block argument. Booleans also implement messages such as <and:>, <or:>, and <whileTrue:>.

As mentioned before, in Smalltalk everything is, or rather behaves like, an object. As a result, you even find objects for the most trivial constants such as booleans (true, false), the undefined object nil, integers, and characters. For any object in the system, there exist a class that specifies its state and defines its behavior. For instance, class Float defines the behavior of floating point numbers, class Process for creating lightweight Smalltalk processes and class Signal for part of the exception handling mechanism. Classes are objects too; they are responsible for creating new instances and keep a dictionary of compiled methods for every implemented message.

Readability

The Smalltalk language is said to be self-documenting which is partly due to the infix notation of message expressions. Instead of sepa-

rating the arguments from the message selector, the message selector may be composed by a sequence of keywords (postfixed by a colon) and arguments.

```
anAgenda        schedule: aMeeting
                on: Date today
                at: Time now + (2 hours)
                with: aPerson

Cube new frontFace leftEdge center x

aText  find: pattern startingAt: index ifAbsent: absentBlock

aCollection removeAllSuchThat: satisfactionBlock
```

By convention, variables names (instance-, temporal- and argument-) are chosen with semantic hints rather than referring to their class. When two or more words are combined to form a name, second and later initials are capitalized to improve readability. This convention applies to all names in the Smalltalk system such as object, variables, and methods.

Blocks

One of the things that makes Smalltalk more elegant and powerful than other OO languages are block closure objects. Blocks are unnamed methods. They encapsulate a sequence of actions to perform those actions at a later time, perhaps even in a different context. Blocks are created by enclosing an expression or statements by squared brackets, e.g. [index := index + 1]. Blocks may require arguments which are specified by argument names, e.g.,

```
[ :costs :eachEmployee | costs + eachEmployee salary ]
```

Blocks are evaluated by sending it the message <value> or one of its variants <value:>, <value:value:> and <valueWithArguments:>, depending on the number of arguments it is expecting. Since blocks are self-contained objects, they can be assigned to variables and be passed as arguments with messages. This kind of pluggable behavior allows for very compact and generic code. It is not just passing a pointer to a function; blocks typically have a private context determined at runtime only. In that context, references to other objects, which are known at runtime, may exist. The following example uses a block to compute 100 factorial.

```
| fac |
fac :=          [ :n | n = 1
                                ifTrue:         [ 1 ]
                                ifFalse:        [ ( fac value: n
- 1 ) * n ]
                ].
fac value: 100
```

Control structures in Smalltalk are invoked by sending messages to various objects. The boolean objects true and false provide the if-then-else machinery as mentioned before. Numbers, collections, and blocks provide the looping methods and are quite natural and consistent with the underlying object model. A more elaborate sort of for-loop comes in the form of the <to:do:by:> method. For example, to compute some CPU-consuming factorials:

```
1 to: 10000 by: 100 do: [:each | each factorial ]
```

According to the message-passing paradigm, this statement should be read as follows. To integer object 1, the message <to:by:do:> is send which takes two numbers and a block for its arguments. The implementation computes integer values for a temporary variable and evaluates the block by passing each such value as its argument.

The basic method <do:> evaluates its one-argument block for each member of the collection that receives the message. It is the most polymorphic message for the collection class hierarchy which includes classes such as Array, String, OrderedCollection, Set, and Dictionary.

Reflection

Reflection is essential to the development environment because it allows for object introspection and code simulation which are part of a Smalltalk debugger. Also many advanced programming techniques make use of reflexive behavior such as distributed computing, object databases, and user-interfaces. Basic messages defined in Object (root of the world) are the following.

class - Answer the class of the receiver
superclass - Answer the (direct) superclass of the receiver
instVarAt: anIndex - Answer the value of instance variable listed at <anIndex>
basicAt: anIndex - Answer the value of indexed variable numbered <anIndex>
perform: aSymbol - Send the receiver the unary message with selector <aSymbol>
instVarNames - Answer the collection of instance variable names of the receiver
classVarNames - Answer the collection of class variable names of the receivers class
become: otherObject - exchange the references of the owners of the receiver with that of <otherObject>

Note that these messages access private information of an object and even allow for modification without respecting the public interface of that object. In normal cases, developers do not need this kind of meta-level programming. However, there are frameworks which are difficult to implement without this reflexive behavior. The following elaborates on an example that makes use of the <perform:> message.

Perform reflection

In the expression "Date today", the message <today> is send to the class Date and returns a new Date. This mechanism is privately executed by the virtual machine (VM) of Smalltalk (see also next Section). In Smalltalk, sending a message can also be specified explicitly using the <perform:> message. The message <perform:> requires its argument to be a Symbol (a constant literal prefixed by #). The given example can be re-written to "Date perform: #today" which instructs the VM to send the message named <today> to the receiver Date. In the example, this symbol is specified by its constant or literal form. However, since the argument is yet another object, it is also allowed to use variables or other expressions that reference or evaluate to symbols. Following examples are illustrative ways to re-write the expression using the <perform:> message.

```
"variable selector"
| selectorToGetToday |
selectorToGetToday := #today
Date perform: selectorToGetToday.

"selector composition"
Date perform: ( to , day ) asSymbol.

"keyword selector"
Date today perform: #addDays: with: 1.
```

The last example illustrates how to write an expression with <perform:with:> when arguments are required. This construct has proven to be a powerful property of the Smalltalk language. It allows for creating objects whose behavior can be parameterized using message selectors. A typical example is the implementation of an interface adapting object. An adaptor provides an object with an interface transformation without changing the behavior of the object or its referencee. In one commercial Smalltalk, GUI objects require model objects to have the value-holder interface. For instance, an input field gets its display string by sending the object <value> and sets the (changed) string by sending the message <value:>. In order to connect this input field to the attribute <name> of a Person object, an AspectAdaptor can be used which is an object that implements a value-holder protocol and forwards those messages by using message selectors from the interface of the Person object.

```
AspectAdaptor >> value
"Answer the value by dispatching the request
to the receivers object using the get-selector"
^self object perform: self getSelector

AspectAdaptor >> value: anObject
"Set the value by dispatching the request
to the receivers object using the set-selector"
self object perform: self setSelector with: anObject
```

The expression below creates a new AspectAdaptor on aPerson object and passes arguments to store the message selector for getting and setting the value (aString) of an attribute (name) of the target object (aPerson). The semi-colons are used for cascading messages send to the same receiver (an AspectAdaptor).

```
( AspectAdaptor new )      object: aPerson
                           ; getSelector: #name
                           ; setSelector: #name.
```

Virtual Machine

A system designed using Smalltalk consists of three parts: the virtual machine, the standard class library and the application-specific classes and class-extensions. The virtual machine is available as a platform-dependent executable whereas the class libraries are stored as compiled classes in a platform-independent image file. Compiled classes are presented by class objects having method objects. A method contains a bytecode sequence which is a platform independent compiled representation of Smalltalk source code. Besides classes and methods, an image file typically contain many other objects that represent constant values which are initialized during development. In some way, they are static variables but their scope may be limited to a single class or class hierarchy.

Bytecode execution

Since the early implementations of Smalltalk [1], such as developed by ParcPlace (ObjectWorks) and Digitalk (Smalltalk/V), there has been a tremendous improvement of both the class libraries and the virtual machine technology. One of the first and most important improvement has been the introduction of compiled cache. Current virtual machines no longer interpret bytecodes to execute a method definition. Instead, the virtual machine compiles the bytecode language to native machine language and then executes it. To prevent compilation of frequently send messages, the native definitions are cached using a particular caching policy thus trading memory for speed. Early versions of ObjectWorks were the first examples that made use of this Deutsch-Schiffman-style dynamic translation or "JIT" virtual machines. Dynamic translation avoids the overhead of byte code dispatch by translating methods into native instructions kept in a configurable cache.

Object memory

The Smalltalk environment provides its own memory management for allocating space for objects and garbage collection, i.e., reclaiming space used by unreferenced objects. Current implementations divide the OS allocated memory into several regions for storing objects that differ for their life-cycle time or size and complexity. Together with sophisticated scavenging techniques and customizable memory policies, the background garbage collection process

is able to run effectively requiring minimal CPU time.

I do not understand the message

When a message is sent to an object, the method dictionaries associated with that object's class and its superclasses are searched at runtime. If none of the classes implement a method for the given message, the VM sends the object the message <doesNotUnderstand:> passing the original message (a Message) as its argument. Again, the VM must search for an implementation of a message and will find it in Object, which is the common superclass to most objects. The default behavior is to invoke a Smalltalk debugger since this event is considered to be a program error (note that in Smalltalk, there is no such thing as a core dump!). However, objects that override <doesNotUnderstand:> can intercept unimplemented messages at runtime, and process them differently. A typical use of this mechanism can be found in Proxies and Forwarders where messages must be delegated transparently to and from other (remote) client objects [2].

Interactive Development Environment

From its birth somewhere in 1972, the development environment has been based on a graphical user-interface (GUI). Because operating systems where not yet equipped with window systems, those Smalltalk implementations provided their own windowing system. Classes such as Window, Menu, Cursor, Controller, and Sensor represent basic objects for constructing a Window system which can still be found in the standard class library, although nowadays they are implemented as wrappers to OS-specific constructs.

The designers wanted to provide a system in which objects could live, be manipulated and defined. Although the language can be used without it, the real power is due to the combination of the language, the GUI, and an object space in which objects live, can be changed, and die. Opposed to most other application building environment, a Smalltalk application is built by extending the standard environment and finally stripping everything (objects, classes, and methods) that is superfluous. This means that applications being build can be run within the same development environment. A Smalltalk debugger can be started in any context to see what messages are send to what objects or to inspect the complete stack of the current or other running Smalltalk processes. Because the development environment itself is build using Smalltalk, it can be extended and enhanced to meet any requirements of the developer. Even the Smalltalk compiler, consisting of a scanner, parser, and code generator, is part of the system and can be reused or even changed.

Do It Yourself

I invite the reader to get a Smalltalk system and start having fun with objects. Several implementations of Smalltalk are available both on the commercial market and from public domain. One particular public domain implementation of Smalltalk is called Squeak [3] which was originally developed by Apple. An interesting aspect to mention is that it includes a Smalltalk implementation of its own virtual machine. In order to create a new VM, a dedicated C-translator is used to generate source files for compiling a new native executable virtual machine. Since the availability of Squeak to the public, many devoted Smalltalkers have compiled virtual machines for many different platforms. Other developers are using Squeak to experiment with new technologies such as morphing, hyper-literate programming, and the Internet; squeaklets are coming soon now. For commercial purposes, more mature Smalltalk implementations are used that include frameworks for object persistency, distributed computing (CORBA), interfaces to legacy systems, and full-featured graphical user-interfaces.

References

[1] A. Goldberg, *Smalltalk-80; The language and its implementation*, 1983

[2] K. Beck, *Smalltalk Best-Practice Coding Patterns*, 1996

[3] *Back to the Future*, The Squeak-team, OOPSLA 1997

Ernest Micklei completed his OOTI-programme in 1995. His passion for object technology and the Smalltalk language in particular, started in 1992 while studying modelling and simulation techniques using a Smalltalk-based application. In 1996, he co-founded ELC Object Technology, a division of ELC Information Technology. As an OO-consultant, he participates in large IS-projects and provides support in analysis, design and implementation of Smalltalk systems.