Eiffel-like assertions and private methods in Smalltalk

Manuela Carrillo-Castellón Jesús García-Molina *
Dpto. de Informática y Automática. Univ. de Murcia
Santo Cristo,1. Murcia 30001. Spain
e-mail: molina@dia.um.es

Ernesto Pimentel
Dpto. de Lenguajes y Ciencias de la Computación. Univ. de Málaga
Pza. El Ejido, s/n. Málaga 29014. Spain
e-mail: pimentel@ctima.uma.es

Abstract

In this paper, we describe the current state of a project whose aim is the extension of the Smalltalk environment by incorporating mechanisms and tools promoting the development of quality software. We have included in the Smalltalk context a number of possibilities, which were not available in the original definition of the language. In this sense, assertions can be specified in our extension, in an Eiffel-like style, so gaining all benefits that the use of assertions bring with them. We also propose an approach to declare private methods, so that the information hiding capability of Smalltalk is improved. These mechanisms are integrated with a new browser and documentation tools increasing the understanding of the class behaviour. The extension has been made by maintaining the Smalltalk philosophy; that is, all elements incorporated to get the new characteristics are defined only using Smalltalk components.

1 Introduction

Smalltalk [Goldberg 83] was designed with the purpose of defining a language and an associated programming environment which acted as an interface between the models of the mind and those of the computer, based on the model of communicating objects [Ingalls 81]. Its origins go back to the beginning of the seventies and for many years has been the principal reference to explain the meaning of object-oriented concepts. Smalltalk facilitates rapid prototyping of applications because it is a dynamically typed language with an interpreted environment that includes powerful and robust tools.

By the middle of the last decade object-oriented programming began to be considered

*The work of Jesús García was partially supported by a grant PCT91/04 from "Consejería de Cultura, Educación y Turismo de la Comunidad Autonóma de la Región de Murcia"

as a very powerful technique to improve software quality, favoring reuse and extensibility. However, Eiffel [Meyer 88] could be the language that better illustrates this view. The same as Smalltalk, it is a pure object-oriented language, but it is strongly typed and combines object-oriented concepts with the characteristics of large-programming languages, such as assertions and exception mechanisms. Eiffel becomes an elegant notation for writing legible programs due to its simplicity of design. This language has been successfully accepted in the academic community but it has not spread through the industrial community yet, because it lacks quality programming environments.

We have attempted to improve the Smalltalk environment in the same way as Eiffel, by proposing mechanisms and building tools, improving several features such as: class understanding, class design, software components specification, debugging and testing programs and class documentation, among others. So, we have augmented Smalltalk with the following characteristics:

a). Assertions checking mechanisms.

b). Classes can include private operations.

c). A browser that improves the understanding of the code, alleviating the illegibility caused by the typeless style, and also including the application and specification concepts, as is quoted in [Wu 90].

d). Automatic generation of classes documentation.

It should be clear that we do not intend to construct a new Smalltalk system, but to enrich an existing one, as is the case of Smalltalk/V [Digital Inc. 91], though our work is portable on any Smalltalk system.

The fact of having quality environments available has promoted the expansion of the Smalltalk users community. We believe this community will be able to take advantage of the improvements offered by our project, having a more appropriate framework to develop software.

In the following section we will discuss the drawbacks of Smalltalk programming which motivate our work, and we outline the approaches that we have chosen to alleviate them. Next, we will describe the Eiffel-like assertion mechanism embedded in Smalltalk. Then, we will explain a modification of the Smalltalk/V environment for distinguishing between public and private operations. Finally, we will establish some conclusions and future work.

2 Drawbacks of Smalltalk

The Smalltalk language is simple due to the fact that it is based on very few concepts (object, class, method, message, polymorphism and inheritance) and to the uniformity provided by the model of communicating objects through messages. The environment supplies a WIMP interface style (windows, icons, menus and pointing device), offers a rich hierarchy of classes and the system tools enable access to the class text in run-time, since the classes are also objects.

But Smalltalk, unlike Eiffel, was not designed from the view point of the object-oriented paradigm as a software engineering technique, so it presents some drawbacks which make software construction more difficult. However, we think that it is possible to improve it, in the way exemplified by Eiffel, with extensions that attenuate these drawbacks. We started our project with this idea, trying to approach the following problems of Smalltalk: a) the typeless style, b) the non-separation between public and private features (method in the Smalltalk terminology), c) the non-supporting of the assertion concept, d) the lack of a methodology promoting the development of reliable classes.

The lack of types declaration makes the methods rather unreadable, so that it is very difficult to capture the meaning of a class only by reading it. We have built a new browser based on [Wu 90] to facilitate the understanding of the classes. As well as distinguishing among abstract vision (specification) and concrete vision (implementation) of a class, this browser also includes the concept of application (separating classes included in an application from classes of the system), and it also allows the addition to the class specification of information which cannot be deduced from the text of the class, such as variable type, object type returned by a method or if a class is abstract.

We have added some documentation tools which are similar to short and flat commands from Eiffel, as well as others intended to favour the sharing of classes among Smalltalk systems, having our extensions available or not.

In order to produce Smalltalk software components with a precise specification, we have enriched it with Eiffel-like assertions. As is quoted in [Meyer 92c], "without assertions it is not possible to produce truly industrial software components". The same as Eiffel, debugging and documentation tools are based on assertions. The debugging process especially takes advantage of the role played by the assertions as a methodical way to annotate the classes with debugging statements.

The assertions also allow the application of the "Design by Contract" methodology in the Smalltalk context. This methodology, described in detail in [Meyer 92a] using Eiffel, was devised in a parallel way to Smalltalk with the aim of building reliable software. It is based on the metaphor "Software by Contract" that proposes the inclusion of proper assertions into the classes. Assertions establish a contract among a class and its clients. This contract forces the client to fulfill certain conditions each time he makes use of a class service (the client has to check that the precondition, stating the requirements of any call to a routine, is satisfied); on the other hand, the class designer has to ensure that each routine will have the expected behaviour (stated as a postcondition expressing the effect of a such routine). The contract also need to take into account the concept of class invariant or property that must be satisfied by all instances of a class in every stable state.

In short, our assertion mechanism embedded in Smalltalk will help developers in the building of reliable classes and in the distribution of classes with a proper specification.

In Smalltalk, all methods defined in a class are considered part of its protocol or interface, and it is not possible to declare private methods. It implies a violation of the information hiding principle that all object-oriented languages should support. We have modified the way of storing the source code of the methods in order to implement a non-selective version of method privatization. This implementation permits the declaration of private methods, such that they can only be used within the class including them; that is, when they are sent to the current object (the *self* entity).

3 Including assertions in Smalltalk

By using assertions, the Smalltalk programmer will be able to include some elements of a specification at the class level (class invariant), at the message level (pre and postconditions) and at the method level (loop invariant, loop variant) in the same way as an Eiffel programmer. So, assertions provide a good mechanism to specify, debug and document a class, in a Smalltalk environment.

The "Design by Contract" methodology is fully applicable, except in handle exceptional cases in which the approaches *a priori* and *a posteriori* [Meyer 92a] are not valid, because we have not developed an exception mechanism yet. Assertions can be monitored in run-time, existing the same possibilities as in Eiffel. So, we have considered the following situations:

a). All the assertions are activated.

b). Only the preconditions are activated.

c). Preconditions and postconditions are activated.

d). All the assertions are deactivated.

For the same reasons presented in [Meyer 92a], by default, only the preconditions are activated.

Throughout this paper we will use the class *Account* as a simple example of class, whose instances represent banking accounts. The definition in Smalltalk of this class could be:

```
Object subclass: #Account
  instanceVariableNames:
    'balance owner personalCode'
  classVariableNames: ''
  poolDictionaries: ''
```

where the state of each instance is represented by the value of the instance variables: *balance*, *owner* and *personalCode*, storing the available money, the owner and the personal number of the account, respectively. Methods defined in this class could be *deposit:*, *withdrawal:*, *balance:* and *changeCode:*, whose meanings are easily deduced from their names.

We will describe below the assertion mechanism included in Smalltalk. This extension is meant to substantially modify the class *ClassHierarchyBrowser* which provides the interface with the programmer.

3.1 Class invariant

We have added to all the class creation methods, a new parameter specifying the invariant of the class. With respect to the storage of the invariant, since it is common to all the instances of a class, we outline two possibilities. The first one was to assign it to a class variable, and the other one to store it as a new field of the class descriptor. We chose the first one since we did not consider it appropriate to modify the structure of the class descriptor, though the implementation might have been simpler.

The class invariant is specified when the class is defined, and it is made by using a block which contains the boolean expression defining the invariant. The syntax used to express the conditions are fixed by the methods defined in the class *Boolean* and its subclasses. So, the definition of the class *Account* could be made in our proposal by sending a message to the root class *Object*, as follows:

```
Object subclass: #Account
  instanceVariableNames:
    'balance owner personalCode'
  classVariableNames: ''
  poolDictionaries: ''
  classInvariant:
    '[(balance>0) and:
    [owner nationality = 'Spanish']]'
```

The class invariant appears as a string containing a block, since a syntactic check must be made to assure that its structure is right, and that involves instance or class variables previously declared. The invariant checking is made by executing a convenient code when an instance of the class is created, and before and after a method is called.

In the example above the class invariant of *Account* establishes that the account balance will be greater than zero and that the nationality of the account owner has to be Spanish. It may seem strange that one of the conditions appears as block and the other does not, but this is necessary because of the implementation of the logical operators 'and' and 'or' in Smalltalk/V. The conjunct which appears as a block, includes the sending of the message *nationality* to an object which represents the account owner. Actually, it is not desirable that an assertion includes messages, for the obvious reasons explained in [Meyer 92a]: they must only involve methods of unimpeachable quality, that do not change the state of the current object or give rise to errors. However, in Smalltalk the interface of a class only contains methods, and the only way of consulting the value of an instance variable is defining a method that returns the object that it references. Supposing that the instance variable *owner* references to objects of the class *Person*, the method *nationality* included in this class could be:

```
nationality
  ↑nationality
```

Obviously, these kinds of methods do not cause problems since they are "safe functions", that is, they have not side effects.

We are also working on the possibility of expressing more complex conditions in the class invariant (e.g. by iterating on structures), but we are not developed it. It could be got by using enumerating messages in the class *Collection*, having as argument the block which defines the invariant. If this capability is achieved the assertion language of Eiffel will be improved.

3.2 Pre and postconditions

Assertions are composed by boolean expressions, including the same extensions that are provided in Eiffel: *old*, to refer the state of an object before executing the method; and *nochange*, to indicate that a given object has not changed its state, i.e. the *old* state coincides with the current state.

Pre and postconditions are used to specify the behaviour of a method. The new template that the system browser presents to the programmer, when he decides to add a new method, is the following one:

```
messagePattern
  ''comment''
  |temporaries|
  require: [..].
    < statements >
  ensure: [..].
```

In this template two new clauses appear, indicating the possibility of including pre and postconditions in the method. We have used the same keywords as in Eiffel: *require* and *ensure*, to specify the pre and postconditions of a method, respectively. This kind of assertion also must be included in the code as Smalltalk blocks. So, the aspect of a Smalltalk method containing pre and postconditions could be illustrated by the *withdrawal* operation on the class *Account*:

```
withdrawal:pesetas
  ''Withdraws the amount pesetas from
  the banking account''
  require:
    [balance>=pesetas and:
        [pesetas<10000]].
  balance ← balance-pesetas.
  ↑balance
  ensure:
    [(self old) balance
        = balance+pesetas]
```

The meaning of pre and postcondition appearing in *withdrawal* are evident from the text of the class. In the postcondition the extension *old* (appears as a message on the current object) is used to reference the state of the object before the method is executed. The *balance* message sent to the object returned by the expression *self old* has the same nature that the method *nationality* presents.

Really, the clauses *require:* and *ensure:* are not special sections in the method code. They are messages which are sent to the current object (i.e. *self*) when it is convenient. So, the Smalltalk philosophy (i.e. all actions are made by sending of messages to objects) is maintained. The system allows the using of these messages not refering the entity *self*, only as a syntactic sugar.

When the system saves a method, previous to its translation into bytecodes, a series of messages is added to the method code intended to check assertions in run- time. The aspect of these control messages depends on the assertions controlling the behaviour of the method, and also on the class invariant, since it must be also checked before and after the method is executed. So, for example, the *withdrawal* method will be really stored as:

```
withdrawal:pesetas
  ''Withdraws the amount pesetas from
  the banking account''
  self chekInvClass: [balance > 0].
  self require:
    [balance>=pesetas and:
        [pesetas<10000]].
  balance←balance-pesetas.
  self ensure:
    [(self old) balance
        = balance+pesetas].
  ↑self return: [balance]
```

In this case, the control messages check the precondition, the postcondition and the class invariant. The last one will be checked before and after the execution of the body of the method is executed. The message *checkInvClass* checks if the invariant is true and it pushes the object state in a stack, which is a global variable for the class *Stack*, named *StackAssertion*. The message *return:* checks the invariant and if this is true, it executes a pop operation on the stack, so removing the state of the current object. The messages *require:* and *ensure:* check the pre and postconditions, respectively. Obviously, the manipulation of these control messages is transparent for the programmer.

Assertions can be checked everywhere in the code by means of a *check* message applied to the *self* entity. So, we can use expressions as *self check: [cond]*, which will check if the assertion *cond*, which is passed as argument, is satisfied at that moment.

Loops specification is also possible by means of the invariant and variant loop, such that the *gcd* could be redefined in the class *Integer* by:

```
gcd:b
  ''Greatest common divisor of b and
  the receiver object''
  |x y|
  require: [self>0 and:[b>0]]
  x←self.
  y←b.
  [x=y] invariant: [x>0 and:[y>0]]
        variant: [x max: y]
        whileFalse:
             [(x>y)
                    ifTrue: [x ← x-y]
                    ifFalse: [y ← y-x]
             ].
  ↑x
```

In the instance creation methods it is necessary to verify that the invariant is fulfilled for the created object. In Smalltalk the creation methods are not declared explicitly, but they are a special case of class methods. Therefore each time a class method is stored the user is asked if it creates instances or not.

In the text of a class method which creates an instance the symbol "↑" must appear before the message creating the instance, to indicate that it is the object returned by the method. The class method *openAccount* of the class *Account* is an example of a method creating instances:

```
openAccount:pesetas owner:aString
  ''Creates a banking account with an
  initial balance given by pesetas,
  and an owner called aString''
  ↑((self new)
    initialize:
      (Array with:pesetas with:aString))
```

So, when a creation method is saved, the symbol "↑" is looked for, and the method *checkInvClass*, whose argument is a block, is applied on the returned object. In the example above, the argument is an empty block, since there is not any postcondition. Internally, the method *openAccount:owner:* will be stored as:

```
openAccount:pesetas owner:aString
  ''Creates a banking account with an
  initial balance given by pesetas,
  and an owner called aString''
  ↑((self new)
    initialize:
      (Array with:pesetas with:aString))
    chekcInvClass:[].
```

Note the *Account* class invariant, namely *balance* is positive, is not included in the block argument of *checkInvClass:*. This is because Smalltalk is not statically typed, and it does include a default initialization for instance variables. So, the value checking for the *balance* variable could imply a run-time error when it has not been previously initialized by an instance method. Nevertheless, it does not mean that the class invariant can be violated when an object is created, because any instance variable initialization must be made by using an instance method, which will contain its corresponding class invariant checking.

It is worth noting the special messages *require*, *ensure:*, or *check:*, cannot be directly emulated by an *ifTrue:* or similar messages. The differences are not only syntactical. In fact, the new predefined methods do not only check boolean expressions; they also make some other actions, as: storing the old state of objects, raising error messages related with the assertion violation, ...; and we think they can provide a good approach to construct fault-tolerant software (e.g. by supporting an exception handler.)

3.3 Assertions and methods redefinition

The "design by contract" methodology promotes a better use and understanding of inheritance and dynamic binding. We have taken into account the redefinition rule [Meyer 92b] which regulates the combination of the assertions with method redefinition: the precondition must be weaker than the original one, and the postcondition stronger. In this sense, we provide similar mechanisms as in the last version of Eiffel, by considering the new methods: *requirelse:* and *ensurethen:*; such that its arguments are added (by the appropriate logic operators) to the assertions corresponding to the redefined method.

3.4 Assertions and Debugging

As is quoted in [Meyer 92b], assertions provide a debugging and testing mechanism more appropriate than the traditional techniques, since it is based on the software component specifications (pre and postconditions).

In our case, when the execution of a message produces an error, because of assertion failure, the initial state of the receiver object is restored, and the user is shown a window indicating the kind of error produced. After this, the user can run the Smalltalk system debugger in order to act conveniently and to continue the execution. To restore the state of an object the *StackObject* stack is used. Since at a given instant, one or more methods will be in an intermediate point of its execution, this stack will store the states of the objects to which has been applied some message, maintaining the order in which they have been executed. *StackObject* stores objects of the class *ObjectContext* which has three instance variables: one of them stores a deep copy of the current object, allowing the restoring of the state, the second one stores a reference to the current object, and the last one stores the block containing the postcondition.

The class *ClassHierarchyBrowser* has also been modified so that a) the option of saving a method adds the control messages corresponding to the assertions checking, and b) the visualization of a method is done with the format shown by the template seen above, eliminating the auxiliary code. In a parallel way, all classes implementing the rest of tools of the environment have also been modified. Basically, these classes have been: *ClassBrowser*, *Debugger*, and *MethodBrowser*. It has also been necessary to add new methods to the class *Behaviour* and to its subclasses *Class* and *MetaClass*.

We must emphasize that our system is portable to other ones not including this assertions mechanism. On the one hand, communication from our system to a non-extended environment is guaranteed, since the exporting process of the source code from a class text to a file includes the possibility of using a filter which removes the control messages, and all the auxiliary information generated by our environment. On the other hand, the extended system can also import classes developed by other systems. All this has been made possible by modifying the protocol for the class *ClassReader*. In addition, some tools are also supplied in order to simulate the functional character of *short* and *flat* commands of Eiffel, automatically generating documentation intended for the clients of a class.

4 Privatization of methods in Smalltalk

An object-oriented language must be able to establish in an explicit way the interface of a class, that will be composed by the operations and attributes considered visible by the class designer. In Eiffel, it is possible to decide if an attribute or an operation is visible only for a group of classes (in an extreme case, to the whole system). Out of the attributes exported by a class, the clients can only consult its state, that is, they cannot modify it directly.

In Smalltalk, the interface of a class is considered to be composed by all the methods of the class, and it is not possible to restrict the accessing to them; that is, every method is considered public. If we take into account that in Smalltalk the only way to give a value to an instance variable is by using an instance method, then the limitation in relation with the information hiding becomes obvious. It is a common practice in this language to associate two methods to each instance variable: one for consulting its value, and another one for changing it.

We have modified Smalltalk/V so that it allows the declaration of private methods and public attributes in consulting mode. Each time the programmer adds a new class to the system, a method is generated for each instance variable with the same name, and returning its value (if the instance variable is declared private the corresponding method is not created).

In a similar way, when a new method is defined in a class it is possible to decide if it will be public or private by means of a pair of buttons. If the method is stored as private, then the prefix "private" is added, automatically, to its name in order to differentiate it from the public protocol, but this handling is totally transparent to the programmer.

We have considered a non-selective version of the privatization mechanism so that a private method is considered as such for all the class clients, including itself, because of the lack of typing in Smalltalk; so, it is not possible to establish statically a selective privatization.

The selectors of the methods belonging to the private protocol are stored internally with a different name to the one given by the user (e.g. by adding the prefix "private"), so the clients of a class can only use the public protocol. In this way, when they are used from a client class a typical error message will be produced: "message not understood". However, private methods can also be applied on the entity *self*, and these situations must not be considered erroneous. To avoid the generation of an error when messages are sent to *self*, the system has to recognize if the corresponding method is public or private. But, when a method is created, the entities and methods used in its definition may be unknown. To solve this problem the prefix "private" is added to all selectors applied on *self*, and each time a public method is defined a method with the same selector, but preceded by the prefix, is automatically created, whose body only contains a message invoking the public version on the entity *self*.

The privatization of methods has been made possible by modifying a number of classes. So, the class *ClassHierarchyBrowser* was modified to change the interface aspect, as well as the storage process. On the other hand, the class *Behaviour* provides a new protocol to separate public methods from private ones. In a similar way, the tools *ClassBrowser*, *Debugger* and *MethodBrowser* have been conveniently updated to present the new functional character.

5 Conclusions and future work

Our main aim has been extending Smalltalk with new mechanisms and tools, in order to increase programmer productivity. In this paper, part of our project has been described, that is, a way to incorporate assertions in the class and method definition, and an approach to hide components have been proposed. Different mechanisms to specify assertions have been included to apply the "design by contract" methodology in the context of Smalltalk. We believe this methodology can substantially help Smalltalk programmers to develop reliable software. Moreover, assertions are very useful in debugging and automatic documentation. In this sense, we are now interested in the inclusion of an exception handling mechanism to also exploit the usefulness of assertions in relation with the construction of fault-tolerant software.

We think our assertion mechanism can be also applied to the object-oriented databases management system, GemStone, since it uses Smalltalk as a data model. We are studying the way we could adapt our work to such a database system.

It has not been possible to include a selective version of the method privatization, because of Smalltalk is not statically typed. Actually, our mechanism facilitates the privatization but it does not prevent it, since the user of a Smalltalk system could easily verify the prefix of the private methods, so avoiding this privatization approach.

Our environment also has an educational interest since it allows the use of Smalltalk in problem solving courses, such as is described in [Wiener 93], and where the object-oriented paradigm is introduced as a software engineering technique.

References

[Goldberg 83] A. Goldberg and D. Robson. *Smalltalk-80: The language and its implementation*, Addisson-Wesley, 1983.

[Ingalls 81] D.H. Ingalls. "Design Principles behind Smalltalk", in *Byte*, August, 1981.

[Meyer 88] B. Meyer. *Object-Oriented Software Construction*, Prentice- Hall, 1988.

[Meyer 92a] B. Meyer. *Eiffel: The Language*, Prentice-Hall, 1992.

[Meyer 92b] B. Meyer. "Design by Contract", in D. Mandrioli and B.Meyer (eds), *Advances in Object-Oriented Software Engineering*, Prentice-Hall, 1992.

[Meyer 92c] B. Meyer, "The New Culture of Software Development: Reflections on the Practice in Object-Oriented Design", in D. Mandrioli and B.Meyer (eds), *Advances in Object-Oriented Software Engineering*, Prentice-Hall, 1992.

[Digital Inc. 91] Digital Inc. *Smalltalk/V Windows Tutorial and Programming Handbook*, 1991.

[Wiener 93] R. Wiener and L. Pinson. "OOP: An academic Perspective", in *Journal of Object-Oriented Programming (Education and Training Supplement)*, May 1993.

[Wu 90] T. Wu. "A better browser for object-oriented programming", in *Journal of Object-Oriented Programming*, November 1990.