Design by Contract 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   Israel Repiso
Dpto. de Lenguajes y Ciencias de la Computación. Univ. de Málaga
Pza. El Ejido, s/n. Málaga 29014. Spain
e-mail: pimentel@lcc.uma.es

Abstract

In the context of object-oriented programming languages we propose an extension of the Smalltalk environment which provides for the possibility of constructing software by applying the *Design by Contract* methodology, introduced by Meyer as a way of developing reliable software. In this sense, both correctness and robustness are increased by using the mechanisms incorporated to Smalltalk as a consequence of the proposed extension. Thus, assertions are used to guarantee the preconditions and postconditions of methods and the stability of the state of the objects, while a mechanism is provided in order to handle the exceptional situations. The Smalltalk programming philosophy is maintained, and the extension does not alter the basic concepts distinguishing this language from the rest of object-oriented languages. This means that the new possibilities incorporated to the environment are accessible by invoking appropriate methods on certain objects. Basic constructions of the language do not need to be modified, so avoiding special sections in the method syntax. The management of execution control has been handled by using the Smalltalk language itself in a transparent way. Therefore, the extension may be easily applied to other Smalltalk systems.

1 Introduction

Smalltalk [2] was designed with the purpose of defining a language and an associated programming environment which acted as an interface between models

*This work has been supported by the proyect CARM PCT 94/67

of the mind and those of the computer, based on the model of communicating objects [3]. 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 as a very powerful technique to improve software quality, favoring reuse and extensibility. Eiffel [6] is the language that perhaps best illustrates this view. It is a pure object-oriented language like Smalltlak, but it is strongly typed and combines object-oriented concepts with the characteristics of large-programming languages, such as assertions and exception mechanisms. These two aspects are the basis for applying the *Design by Contract* methodology [8], which was proposed by B. Meyer in the context of the Eiffel language.

Object-oriented programing is proposed as a methodology capable of solving some of the difficulties concerning software development, such as extensibility and reusability. However, some other aspects must also be dealt with. Among these, we are interested in those related to software reliability, taking into account the two possible points of view: correctness and robustness. In fact, one of the drawbacks of Smalltalk when it is compared with Eiffel is the absence of mechanisms specially designed to cover these two aspects. In order to solve this inconvenience we have attempted to improve the Smalltalk environment in the same way as Eiffel, by proposing an extension which provides a number of characteristics directed to specify assertions and handle exceptions. It must be clear that we do not intend to construct a new Smalltalk system, but to enrich an existing one.

The extension we propose for Smalltalk is based on the possibility of specifying assertions in different ways, and establishing execution control in the case of exceptional situations. So, pre and postconditions of methods, class invariants, or loop variants and invariants can be defined, with the aim of improving documentation, correctness and debugging of classes. However, the capability of specifying assertions is not enough; it is also convenient to establish the behavior of the system when assertions are not safisfied. In [1] this is solved by activating the debugger with information about the violated assertions. Nevertheless, we are interested in providing a way of preserving execution control after an assertion has been violated. In other words, our objective is not only behavior specification, but also fault-tolerant software construction in Smalltalk.

The more usual way of expressing fault-tolerance is the use of mechanisms explicitly dedicated to this aim, such as exceptions handling. This kind of mechanism allows for the description of elegant and clear algorithms to solve normal situations, whereas exceptional situations are relegated to external code sections. Exception handling is based on dividing the domain of an operation (or method) into two parts: the standard domain, and the exceptional domain. So, if an operation is invoked (i.e. if a message is sent to an object) in an initial state belonging to the exceptional domain, the execution of the operation (i.e. message sending) raises an exception, which may be handled or not. A way of distinguishing between exceptional and normal situations is by means of assertions. Therefore, assertions and exception handling are two complementary mechanisms.

Thus, notions directly related to software quality like correctness and robustness are incorporated into the Smalltalk context, such that software development in this language can progress in a more disciplined way.

The fact of having available quality environments 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 proposal, having a more appropriate framework to develop software. In addition, as is stated in [10], the most appropriate languages to be used as the first object-oriented programming language are Eiffel and Smalltalk, because both were designed as pure object-oriented languages. If we are also interested in introducing object-oriented methodology from the perspective of large-software development, we are convinced that the most appropriate language is Eiffel. However, the ease of constructing prototypes, joined to the low cost of the environments based on Smalltalk compared with Eiffel systems, and the higher price of the machines supporting them, makes the Smalltalk language more accessible for introductory courses of software engineering in academic environments. So, we think our extension improves the use of Smalltalk in the teaching of object-oriented software construction.

In the following section we will discuss the drawbacks of Smalltalk programming which are motivating our work, and we outline the approaches that we have chosen to alleviate them. Then we will describe the Eiffel-like assertion mechanism embedded in Smalltalk. Next, we will explain the approach we propose to construct fault-tolerant software in Smalltalk, and we relate it with assertions. 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 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), offering a rich hierarchy of classes and system tools. However, Smalltalk, unlike Eiffel, was not designed from the viewpoint of the object-oriented paradigm as a software engineering technique, so it presents some drawbacks which make large-software construction more difficult. Nevertheless, we think it is possible to improve it, in the way described in Eiffel, with extensions that attenuate these drawbacks. So, different aspects of Smalltalk like typeless style, non-possibility of defining private methods, non-support of the assertion concept, or the lack of a methodology promoting reliable class development, can be improved by extending the class hierarchy provided by the environment [1]. In the present work we describe the solution concerning the two last mentioned problems, i.e. using assertions and reliable class construction.

In order to produce Smalltalk software components with a precise specification, we have enriched it with Eiffel-like assertions. As is quoted in [9], "without assertions it is not possible to produce truly industrial software components". The same as Eiffel, debugging and documentation tools will be based on assertions. In particular, the debugging process takes advantage of the role played by assertions as a methodical way to annotate the classes with debugging statements.

Assertions combined with exceptions handling also will allow the application of the *Design by Contract* methodology in the Smalltalk context. This methodology, described in detail in [8] using Eiffel, was devised in a parallel way to Smalltalk with the aim of building reliable software. It is based on the *Software by Contract* metaphor 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 exhibit the expected behavior (stated as a post-condition 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. A mechanism for explicit exceptions handling, which has been incorporated into Smalltalk, will allow the treatment of situations in which assertions are not satisfied.

In short, our proposal of extending Smalltalk with assertion and exception handling mechanisms will help to developers in the building of reliable classes.

3 Including assertions in Smalltalk

By using assertions, the Smalltalk programmer will be able to include some relevant aspects of the specification at different levels: class level (class invariant), the message level (pre and post-conditions), and method level (loop invariant and variant), in the same way as an Eiffel programmer can. So, assertions provide a good mechanism by which to specify, debug and documentate a class, also in a Smalltalk environment, such that the *Design by Contract* methodology can be applied, by including the handling of exceptional cases by means of *a priori* and *a posteriori* approaches [7]. Assertions can be monitored in run-time, with the same potential as in Eiffel.

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

```
Object subclass:#BankAccount
```

```
  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. Some of the methods defined in this class could be `deposit:`, `withdrawal:`, `balance:` and `changeCode:`, whose meanings are easily deduced from their names.

We will describe the assertion mechanism incorporated into Smalltalk below. The extension allowing the use of assertions in this way are intended to substantially modify the class `ClassHierarchyBrowser` which provides the user interface.

3.1 Class invariant

We have added to all class creation methods, a new parameter to specify 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 is appropriate to modify the structure of the class descriptor, although the implementation could 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. So, the definition of the class `BankAccount` can be made in our proposal by sending a message to the root class `Object`, as follows:

```
Object subclass:#BankAccount
  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 checking must be made to guarantee that a valid structure is being used, and that involves instance or class variables previously declared. 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 establishes that the account balance will be greater than zero and that the nationality of the account owner has to be spanish.

3.2 Pre and post-conditions

Assertions in our Smalltalk environment are composed by boolean expressions, including the same extensions provided by Eiffel, i.e. `old`, referring to the state of an object before executing the method, and `noChange`, to indicate that a given object has not changed its state.

Pre and post-conditions are used to specify the behavior 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 post-conditions. We have used the same keywords as in Eiffel: `require` and `ensure`, to specify the pre and post-conditions of a method, respectively. In fact, 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. every action in the system is a consequence of sending a message to an object) is maintained. The pre and post-conditions are included in the code as Smalltalk blocks. The aspect of a Smalltalk method containing pre and post-conditions could be illustrated by the `withdrawal:personalCode:` operation:

```
withdrawal:pesetas personalCode:code
  ''Withdraws money from the account
  if the code is right''
  self require:
    [balance>=0 & pesetas<50000].
  self ensure:
    [((personalCode <> code) & self noChange) or:
    [(self old) balance = (balance + pesetas)]].
  (personalCode = code)
    ifTrue:  [balance ← balance-pesetas.
             ↑balance]
```

The meaning of pre and post-condition appearing in `withdrawal:` are evident from the text of the class. In the post-condition the extension `old` (appearing as a message on the current object) is used to refer to the state of the object before the method is executed. The message `balance` sent to the object returned by `self old` has the same nature as the method `nationality`.

When the system saves a method, previous to being translated onto bytecodes, a series of messages is added to the method code intended to check assertions in run-time. These control messages depend on the kind of assertion used to specify the behavior of the method, e.g. `withdrawal` really will be stored as:

```
withdrawal:pesetas personalCode:code
  ''Withdraws money from the account
  if the code is right''
  self chekInvClass: [balance > 0].
  self require:
    [balance>=0 and: [pesetas<50000]].
  self ensure:
    [((personalCode <> code) & self noChange) or:
    [(self old) balance = (balance + pesetas)]].
  (personalCode = code)
    ifTrue:  [balance ← balance-pesetas.
             ↑self return:  [balance]].
```

In this case, the two first control messages check the class invariant and the pre-condition. The third control message only stores the post-condition to be later tested by executing the method **return:**. The class invariant will be checked before and after executing the body of the method. The message **checkInvClass** checks if the invariant is satisfied and it pushes the object state on an **LinkedStack** instance. The message **return:** pops the state of the current object from the stack if the invariant and post-condition are successfully evaluated. In addition, it returns the argument. A previous analysis is needed to construct the code which is actually saved. This is done in a transparent way.

Loop specification is also possible by means of the invariant and variant loop. In this case, it was only necessary to add a new method to the **Context** class, named **invariant: variant: whileFalse:**. It is also possible to check assertions everywhere in the code, by means of a **check:** message applied to the **self** entity. So, an expression as **self check: aBlock** will check if the assertion given by **aBlock** is satisfied at that point.

Concerning methods to create instances 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 **BankAccount** is an example of a method creating instances:

```
openAccount:pts owner:aString
  ''Creates a banking account''
  ↑((self new)
    initialize:
      (Array with:pts with:aString))
```

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

```
openAccount:pts owner:aString
  ''Creates a banking account''
  ↑((self new)
    initialize:
      (Array with:pts with:aString))
    chekcInvClass:[(balance>0) and:  [owner nationality = 'spanish']].
```

Note the `Account` class invariant is included in the block argument of `chekcInvClass:`. It means that instance methods initializing an object must be defined such that they ensure the class invariant. That is, all instance variables involved in the invariant have to be correctly initialized.

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 they can provide a good approach to construct fault-tolerant software (e.g. by supporting an exception handler.)

We have taken into account the redefinition rule [8] which regulates the combination of the assertions with the method redefinition: the precondition must be weaker than the original one, and the post-condition stronger. In this sense, we provide similar mechanisms as in the last version of Eiffel, by considering the new methods: `requireElse:` and `ensureThen:`; such that its arguments are added (by the convenient logic operators) to the assertions corresponding to the redefined method.

3.3 Assertions and Debugging

As it is quoted in [8], assertions provide a debugging and testing mechanism more appropriate than the traditional techniques, since it is based on the software component specifications (pre and post-conditions).

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 a window is shown indicating the kind of error produced. After this, the user can run the Smalltalk system debugger in order to proceed conveniently. The state restoring is made by using the global stack which is maintained by the checking assertion system. 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 on which has been applied some message, maintaining the order in which they have been executed. Objects stored by this stack are instances of the class `ObjectContext` which is defined by three instance variables: one of them containing a deep copy of the object in whose context a message is being processed (it allows the restoring of the state), another one stores a reference to the current object, and the last one stores the block containing the post-condition (which must be evaluated after the message is processed.)

The class `ClassHierarchyBrowser` has been also 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 has been also modified. Basically, these classes have been: `ClassBrowser`, `Debugger`, and `MethodBrowser`. It has been also necessary adding new methods to the class `Behavior` and to its subclasses `Class` and `MetaClass`.

We must emphasize that our system is portable to other ones no including this assertion mechanism. On the one hand, the communication from our system to a non-extended environment is granted, since the exporting process of the source code from a class text to a file includes the possibility of using a filter which removes 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*.

4 Fault-tolerance in Smalltalk

Smalltalk, as initially defined, cannot be considered an adequate language for fault-tolerant software construction. Unfortunately, there are not independent mechanisms to recover execution control when a run-time error is produced. By 'independent' we mean not interacting with the user. To solve this problem, the Smalltalk environment might be extended with some Eiffel-like characteristics concerning exception handling, but still maintaining the basic philosophy of Smalltalk.

Exception handling is one of the most extended mechanisms used in specifying fault-tolerant software. The idea of this kind of construction is based on decomposing the operation domain into two disjoint subdomains: that corresponding to normal computations, and that corresponding to abnormal situations. These situations, which are not contemplated in the standard behavior of the operation, are solved by raising an exception, which might or might not be handled. If the exception is dealt with the computation error can be avoided. In the other case, the exception will be propagated, by delegating the execution control to the environment.

Essentially, there are two different models for exception handling [4]. The most frequently used is the *termination model*, which is characterised thus: when an exceptional situation is produced during the computation of an operation, this is then stopped, and execution control is passed to the exception handler. This method is much more simple than the *continuation model*. where the execution control is returned to the original operation when the exception has been dealt with. Most languages incorporating some kind of exception handling present a termination model: Clu [5], Ada, Eiffel [6]. Although it is possible to find some proposals which relax the complexity of the continuation model [12], we have adopted the termination model in our proposal.

On the other hand, we have defined *disciplined* exception handling, in the sense described by Meyer in [6]. So, there are only two ways of finishing the execution of a method: a) successfully, and verifying all postconditions, or b) by propagating the exception to the environment (invoking method), after the stable state of the object has been restored.

As in Eiffel, our proposal incorporates the possibility of handling exceptions at two different levels. In this way, it should be possible to define blocks to handle exceptions at either method and class level. In the last case, the block defining the method of handling the exception will only be activated when the method corresponding to the currently processed message does not contain an exception handling block itself.

These exception handling class level blocks are specified in a similar way as the class invariant is, that is:

```
Class subclass: #Subclass
  instanceVariableNames:
  classVariableNames: ‘’
  poolDictionaries: ‘’
  classRescue: ‘’
  classInvariant: ‘’
```

The argument corresponding to `classRescue:` is a block containing the sequence of actions to execute when an exception is produced, which is not handled by the method where it was raised.

In a similar way, we can use the method `rescue:` in the body of any method, which must be sent to the `self` entity. The syntax has also been relaxed to allow the using of `rescue:` as the name of a special section.

By maintaining the similarity to Eiffel, we have also added the `retry` method, whose objective is again to execute the method in whose context the exception was raised.

The fault-tolerance obtained by the proposed mechanism covers two different aspects: external and internal fails. The first kind of fail is derived from operating system and hardware errors, whereas the second one is due to assertion failures.

To illustrate the way assertion failures are dealt with by the exception handling mechanism, we can consider the following example, redefining the `withdrawal: personalCode:` method:

```
withdrawal:pesetas personalCode:code
  ‘‘Withdraws money from the account
  if the code is right’’
```

```
  self require:
    [balance>=0 & pesetas<50000].
  self ensure:
    [((personalCode <> code) & self noChange) or:
    [(self old) balance = (balance + pesetas)]].
  self rescue:
    [balance ← (self old) balance].
  (personalCode = code)
    ifTrue:  [balance ← balance - pesetas.
             ↑balance]
```

Thus, if an error is produced during the execution of a `withdrawal: personalCode:` method, either external (e.g. storage device error) or internal (the amount to be withdrawn is greater than the balance), then the `rescue:` message provides the possibility of restoring the object state. In this example, the method does not fulfil its contract (determined by the post-condition). Therefore, although the exception has been partially handled, it must be propagated.

However, on some occasions retrying the execution of a method might be convenient after an exception is raised. We can consider, for instance, a class representing an automatic cash desk, containing an instance method `introduce:`, which models the actions to be made when a client introduces a card:

```
introduce:card
  ``Models the actions to be made when a
  card is introduced in the cash desk''
  |account code amount|
  account ← bank getAccount:(card info).
  code ← device getPersonalCode.
  amount ← device getAmount.
  account withdrawal:amount personalCode:code.
  self rescue:
    [ device message:'Please, try it again !'.
    self retry ]
```

In this second example we are assuming the cash desk is connected with a bank, and there is an input/output device, which are represented by two instance variables: `bank` and `device`, respectively. Once account information is obtained from the `card`, and the account is located through the bank connection, some additional information is demanded from the client by using an input/output device: the code and the amount to be withdrawn. After that, the corresponding message is sent to the account. At this point, if an exception is derived from the execution (e.g. if the amount is greater than the balance), then a message is displayed and the method is retried from the beginning. Observe that any exception is raised when the code does not correspond with the actual code. Such a situation has no effect.

A more sophisticated definition for `introduce:` might be the following:

```
introduce:card
```

```
''Models the actions to be made when a
card is introduced in the cash desk''
|account code amount|
(failures < 3)
  ifTrue:
    [(account ← bank getAccount:(card info).
    code ← device getPersonalCode.
    amount ← device getAmount.
    account withdrawal:amount personalCode:code.
  ifFalse:
    [device extractCard.  failures ← 0].
self rescue:
  [ device message:'Please, try it again !'.
  failures ← failures+1.
  self retry ]
```

The instance variable **failures** is used as a counter, and we are assuming its value is 0 in stable states. Each time an exception is raised because a wrong amount was introduced, the variable is increased, and the method is executed again from the beginning by sending the message **retry** to **self**.

The issues we have followed to enable the end of message processing are those defined in Eiffel.

4.1 Implementation issues

In this section we are going to explain some details concerning the implementation of exception handling at the method level. The class exception handler is implemented by considering a class variable, which is evaluated when an exception is raised and the corresponding method does not contain any exception handler.

When a new method is defined (class or instance), the browser provides a new pattern message incorporating the possibility of defining an exception handler for the method. The protocol of the `Object` class has been extended to present these characteristics. In addition, some other classes have been newly defined and extended, to manage execution control when an exception has to be raised and handled. We want to do this without modifying the Smalltalk virtual machine. So, each message sending has to be (transparently) preprocessed, in order to include the additional behavior, given by the eventual exception handling. To do this transparently for the user, a filter hiding all internal controls is also necessary.

The preprocessing of a method including assertions and/or an exception handler needs the correct sequence of messages on the current object. The messages invoked are different depending on the kind of method. So, we distinguish three different situations: a) the method does not contain an exception handler, b) the method contains one, but it does not retry, and c) exception handling could cause the method to retry.

As in assertion checking, we need to store the system's history, and we also use a linked stack, but on this occasion, elements of this stack are instances of a descendant class, which has five components: the three components of the parent class, a boolean variable indicating if the handler includes `retry`, and the exception handling block.

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, in particular a way of incorporating assertions in the class and method definition, and a mechanism for exception handling in the Smalltalk context.

Although Smalltalk and Eiffel's philosophies are rather different, we believe that some of the facilities which were introduced into Eiffel to obtain a more reliable software construction might be also applied in Smalltalk. And it is worth noting that this might be done in a very easy way, due to the flexibility of Smalltalk.

With this extension, the design by contract methodology stops being the exclusive preserve of Eiffel.

We think our assertion mechanism can be also applied to the object-oriented database 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.

On the other hand, our environment may also have educational interest since it allows the use of Smalltalk in problem solving courses, such as is described in [10], and where the object-oriented paradigm is introduced as a software engineering technique.

References

[1] M. Carrillo-Castellón, J. García-Molina and E. Pimentel. "Eiffel-like assertions and private methods in Smalltalk", in *TOOLS 7*, Prentice-Hall, 1994.

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

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

[4] J.L. Knudsen. "Better exception handling in block structured systems", *IEEE Software, 17, 2*, pp. 40–49 ,1987.

[5] B.H. Liskov and A. Snyder. “Exception handling in Clu”, *IEEE Transactions in Software Engineering, 5, 6*, pp. 546–558, 1979.

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

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

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

[9] 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.

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

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

[12] S. Yemini and D.M. Berry. “A modular verifiable exception handling mechanism”, *ACM Transactions on Programming Languages and Systems, 7, 2*, pp. 214–243, 1985.