OBJECT-ORIENTED SPECIFICATION IN SOFTWARE DEVELOPMENT

Simona MOTOGNA[*]

Received: 20.11.1995
AMS subject classification: 68N15, 68N05

REZUMAT. - Specificarea obiect-orientată în dezvoltarea software. Specificarea formală joacă un rol important în dezvoltarea de sisteme informatice largi şi complexe. Pe de altă parte, programarea orientată obiect s-a dovedit în ultimii ani ca fiind un instrument cu beneficii clare în dezvoltarea de produse informatice. Scopul acestei studiu este de a propune o metodă de specificaţie orientată obiect bazată pe descompunerea, abstractizarea şi încapsularea sistemului, oferind şi posibilitatea reutilizării.

1. Introduction. We propose an object-oriented specification method which will support widespread reuse and respects the following principles: a specification, in general, must be formal, understandable, as well as abstract and implementation-independent. Reuse of the software components is possible in the same problem or for other similar problems.

The design of a component influences its potential for reuse, but a good design is not always sufficient. The expression of the design is equally important. The specification of a component must be achieved such that the implementation and use of that component meet the following conditions:

- understand easily, but exactly which is the component functionality;
- choose freely among multiple, efficient implementations;
- certify that an implementation satisfies the specification requirements.

[*] *"Babeş-Bolyai" University, Faculty of Mathematics and Computer Science, 3400 Cluj-Napoca, Romania*

2. Specification Features. Object-oriented features can be successfully used to describe abstract entities, and this is exactly what a specification must do - to study the abstract behavior without any constraints about computer architectures. That's why, object oriented concepts like data abstraction and encapsulation (components will be considered classes) will be used in this specification method, as well as inheritance and polymorphism to support software reuse.

The idea of this method is based on Eiffel [4]. The principal reasons for choosing Eiffel are:

- it is an object-oriented language, so it offers data abstraction, encapsulation, inheritance and polymorphism;

- it has a set of assertions which can express the conditions that an operation has to satisfy.

We shall shortly overview which are these assertions, which in this case will be specified in conditions.

- Preconditions will be specified through a **requires** clause and represent the conditions under which the operation will function correctly.

- Postconditions will be specified through an **ensures** clause and represent the conditions assured after performing the corresponding operation.

If the precondition of an operation holds before an operation is invoked, then the postconditions will be guaranteed to hold when the operation completes, assuming a correct implementation of the specification.

In addition we suppose that each parameter of an operation has one of the following modes: *conserves, uses, produces* or *modifies*:

- the *conserves* mode indicates that the parameter value will remain unchanged during the operation performing (like the invariant assertion in Eiffel);

- the *uses* mode indicates that a parameter value is used by the operation and that the initial value is not modified;

- the *produces* mode indicates that the obtained value is relevant or that the parameters has a value only after performing the operation;

- the *modifies* mode is used when a parameter has an input value that is modified and returned by an operation.

These modes are only specification notations and should not be confused with parameter passing mechanisms. They are included in the specification only to increase the understandability of it.

The interface of a class describes what the component provides and these are the only data and operations which are visible to other components.

The reuse of the component in defining other components of the same system is achieved using inheritance, specified by an *inherits* clause. Reusing this component in another system is easy to achieve since the specification of a component is encapsulated.

In order to understand this method of specification, the following example will be considered:

Specification of a generic **Stack**

```
class Stack[Item]
Type content: sequence of Item

interface
 init, empty, push, pop, top
end;

operation init
        parameters:{produces c:content
                            }
        ensures c = []
end_operation

operation empty
```

```
        parameters:{conserves c:content
                        produces b:Bool
                        }
        ensures (b = c = [])
end_operation

operation push
        parameters:{modifies c:content
                        uses i:Item
                        }
        ensures ( not empty ) ∧ (c=c+[i])
end_operation

operation pop

        parameters:{modifies c:content
                        produces i:Item
                        }
        requires (not empty)
        ensures (c=c-[i])
end_operation

operation top
        parameters:{conserves c:content
                        produces i:Item
                        }
        requires (not empty)
        ensures (not empty)
end_operation

end_class --class Stack
```

This example shows a way of specifying stacks regardless of the elements type. A stack component must provide operations as creation (init), a test: is the stack empty? (empty), the classical operations push, pop and top. Instead of defining these operations as procedures or functions, the parameters and their mode are specified separately, between braces. Then, the preconditions and postconditions are described.

In general, a specification of a component will be:

```
class <class_name>
inherits <class_name>                    // the class from each
                                               //inherits
Type ...                                       // user defined types
interface                                // operations exported
 <op1>,<op2>,...
end;

operation <name_op>
     parameters:{[conserves <var>:<type>,..]
                   [uses <var>:<type>,..]
                   [produces <var>:<type>,..]
                   [modifies <var>:<type>,..]
                  }
     [requires (cond1) [∧ (cond2) ...]]
     [ensures (cond1) [∧ (cond2) ...]]
end_operation
...                                            // the same format for each
                                               // operation
end_class
```

The complete syntactic definition of the language is given in Appendix A.

It's easy to observe that this specification method is not intended to suggest any implementation method. The purpose of specification is, on the contrary, to give more choices of implementation.

3. Conclusions. The specification method described above meets the criteria regarding formal specification, and also offers flexibility and security. The idea of this specification was found in [2], but lacks in information regarding the operations specification. The model proposed has added some techniques in order to give a more clear specification for each operation included in a class.

The purpose of it is to fit between the object-oriented analysis and design and an object-oriented implementation.

This study represents more an idea which can be applied with fruitful results, especially in reusing software components.

The specification method is implementation independent. Any object-oriented programming language can be used for implementation, but this is not a requirement: this kind of specification can be used for any other language without object oriented features, transforming these class definitions in user defined types and the operations in procedures and functions. The dezadvantage of such a language is that properties like inheritance, polymorphism and encapsulation are not available and the programmer has to find a way to "translate" the given specification using the language facilities.

Appendix A: Syntactic Definition

The syntactic definition is given in extended BNF (Backus-Naur Form). The following notational conventions are used:

- the keywords are bold;
- <> item enclosed in angle brackets are required
- [] item enclosed in square brackets are optional
- { } items enclosed in braces may be repeated zero or more times
- // everything that follows up to the end of line denotes comment

```
<class_description> ::= class <class_name> [[<formal_type_parameters>]]
                          <inheritance_declaration>
                          Type <type_definition>
                          <interface_declaration>
                          <operation_description>
                          end_class

<class_name> ::= <identifier>
<formal_type_parameters> ::= <identifier_list>
<identifier_list> ::= <identifier> {, <identifier>}

<inheritance_declaration> ::= inherits <class_list>
<class_list> ::= <class_name> {, <class_name>}

<type_definition> ::= <identifier> : <type>
```

```
<type> ::= Integer | Bool | Sequence of <identifier> |...*

<interface_declaration> ::=   interface <op_name> {, <op_name>}
                              end
<op_name> ::= <identifier>

<operation_description> ::= <operation_item> {, <operation_item>}
<operation_item> ::=     operation <op_name>
                         parameters: {[conserves <var_decl_list>]
                                        [uses <var_decl_list>]
                                        [produces <var_decl_list>]
                                        [modifies <var_decl_list>]
                                      }
                         [requires <condition_list>]
                         [ensures <condition_list>]
                         end_operation
<var_decl_list> ::= <var_list> : <type>
<var_list> ::= <var_name> {, <var_name>}
<var_name> ::= <identifier>
<condition_list> ::= <condition> { ∧ <condition>}
<condition> ::= <bool_exp> | not <bool_exp> |
                <bool_exp> V <bool_exp> |<bool_exp> ∧ <bool_exp>
<bool_exp> ::= <exp> <rel_operator> <exp> | <op_name>**
<rel_operator> ::= < | > | = | <> | <= | >=
<exp> ::= <var_name> | <constant> | <exp> <operator> <exp>
<operator> ::= + | - | * | /
<constant> ::= <number>
<number> ::= <digit>{<digit>}
<digit> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0
```

* It can be completed with other types, when it is necessary

** Note that an operation which returns a boolean value can be considered an boolean expression

REFERENCES

[1] L. Cardelli, P. Wegner - On understanding types, data abstraction and polymorphism, Computing Surveys, 17(4), pp.471-522.

[2] Y Cheon, G. T. Leavens - The Larch/Smalltalk Interface Specification Language, ACM Transactions on Software Engineering and Methodology, July 1994, vol. 3 no.3, pp. 221-253.

[3] B. Meyer - Object-Oriented Software Development, Prentice-Hall, 1988

[4] P. Wegner - Concepts and Paradigms of Object-Oriented Programming, OOPSLA'89 Keynote Talk, OOPS Messenger, vol.1,no.1, pp. 7-87, 1990