Stepwise Process for the Object Model in Booch Methodology

M. González, F. Losavio, A. Matteo, C. Metzner
Centro de Ingeniería de Software Y Sistemas ISYS, Escuela de Computación Universidad Central de Venezuela, Apdo. 47567, Caracas, Venezuela

Abstract

The goal of this work is to propose an assisted process to facilitate the correct use of the notational elements proposed in Booch object-oriented methodology for expressing the static behavior of the system. The definition of this process favors the combination of different methodologies at different stages of development, which is an important issue that can be exploited for the construction of automatic tools for software development.

1. Introduction

A well defined and expressive standard notation is a very important issue in the software development process. A standard notation allows the description of a software system architecture and the communication of results among persons involved in the development process, so that the developer may pay attention to more relevant aspects of the problem to be solved. Finally, a powerful enough notation can contribute to automatic consistency checking. A graphical notation for object-oriented (OO) analysis and design methodology is proposed in [Boo 94], as an extension of an earlier one for design, formulated by the same author [Boo 91]. The Booch extended notation is enriched with new elements to express implementation with more details and is rather oriented towards C++ [Str 93], allowing to model the logical and physical views of a system.

The goal of this work is to propose an assisted process for developing the object model, guaranteeing the correct use of the notational elements proposed in an OO methodology. This process is applied to the Booch method [Boo 94], but it could be generalized to other methodologies. It has the advantage of easing consistency checking during each stage of the development process. Moreover, even if it imposes some rigidity, it is a step towards the automatic transfer of information among different methodologies, such as Coad&Yourdon's [CY 90], OMT's [Rum&Al 91] and Booch's [Boo 94]. The process presented here provides guided or assisted construction of applications using the Booch methodology within a CASE tool integrated into the OODEST environment [LMM 94] for OO software development.

Besides this introduction and the conclusions, this paper is constituted by two main sections: Section 2 and 3, describing the systematic process for constructing the Booch object model.

1 This research is supported by the New Technology Program of the BID-CONICIT
2. PROCESS: CONSTRUCTION OF THE CLASS DIAGRAM IN THE OBJECT MODEL USING BOOCHE'S GRAPHICAL NOTATION.

The object model captures the static behavior of the objects involved in an application, and it can be expressed by means of a graphical and/or textual notation. We are concerned with the construction process of the object model using the graphical notation given in [Boo 94] for expressing the class diagram and the object diagrams. In this context we use the object model notion defined in [Rum&Ai 91] for expressing the static component of the logical view of a system. The construction of each diagram in our process is presented as a sequence of steps, that can be applied with the analysis phase as a starting point, in order to obtain a detailed design of the solution applying stepwise refinement of the initial model. An equivalence between the four initial steps of our process and the analysis object model of Coad&Yourdon in terms of the notational elements is established in [LMS 94]. The equivalence allows automatic translation of the object model into Booch notation, in order to go on with the Booch design phase. The advantage of this translation is to make use of the Coad&Yourdon analysis method, richer in guidelines than Booch's, and to continue with the much more expressive Booch design method. In what follows, the Booch notation will be described by an example taken from [LMS 94].

An Inventory Problem: The example considered deals with the inventory of software and hardware articles in a company. For each article or item, identified with an inventory number, the following characteristics are considered: a short description, the kind of model, the date of the buy, the price, the depreciation rate, the validity of the warranty, the manufacturer's name. The operational, non operational or in-repair state of the item is also considered. Each item is under the responsibility of a unique person, which may be located by office or home phone, if required. The items are placed in offices which are distributed in different buildings. Among the items which are hardware components, we distinguish the basic computer (set of elements indispensable for using the computer, such as screen, keyboard, central processing unit) and the components (such as printer, external device, etc.), which must be added to the set of elements of the basic computer. A basic computer may exist independently, while a component must be always connected to at least a basic computer.

Figure 1. Class Diagram for the inventory problem

---

\[2\]This example is a simplification of a real-life problem, formulated within the context of a joint project between the ISYS research center and the MARA VEN oil company, Petróleos de Venezuela S.A.
Figure 1 illustrates the class diagram corresponding to the inventory problem obtained applying the proposed process. In each step references will be made to this diagram in order to illustrate the different notation elements used.

A class diagram is used to show the classes and their relations, as an element of the logical model, used for expressing the logical view of a system [Boo 91]. During analysis phase, this diagram is used to indicate the roles and responsibility of the entities defining the system behavior. At design phase, this diagram is used to capture the structure of the classes constituting the system architecture.

The stepwise process is presented in what follows. In each step, some possible consistency checking will be pointed out.

**Step 1: Identification of classes and associations.**

The classes corresponding to the problem domain, and their semantic connections, are identified during this first step. A semantic connection between classes is called an association and at this level, it expresses only the existence of some kind of relationship between classes whose type (inheritance, has, use) will be defined later in our process. A class associated with itself is said to have a reflexive association. It is also possible to have more than one association among two classes, excluding the inheritance relation. In figure 1, the ITEM and FAILURE_ON_ITEM classes, related by an association, expressing the possible failures of an item, have been identified. The classes are represented by clouds (or rectangles) and the associations in this step are represented by a straight line, without adornments. Consistency checking of classes and associations names present in the dictionary are possible.

**Step 2: Identification of Roles and Keys of the associations.**

a) **role**: denotes the purpose wherein one class associates with another. The role of the class is named as a textual adornment to any association, placed adjacent to the class offering the role.

b) **key**: is an attribute whose value uniquely identifies a single target object.

This step is optional and helps to explain how one class is operating upon another.

Verifying the unicity of role and key names in the dictionary are consistency checking that can be made in this step.

**Step 3: Identification of attributes and methods.**

From the classes identified in the previous step, we proceed to identify the resources of the class, i.e. the relevant attributes and methods (called also operations or services) for each class. At this level, only the names of attributes and methods are given; in later steps more characterizations are added. There are no additional graphic notational elements for this step.
In this step, it will be checked that, for each class, there are no attributes (methods) with the same name.

Step 4: Characterization of the associations and assignment of cardinalities.

This step consists in specifying, as a first refinement, the type (inheritance, whole-part, called also has in [Boo 94] or use) and cardinality of the association. The inheritance relation defines a generalization/specialization association (BASIC_COMPUTER and HARDWARE fig. 1) represented by a directed arc, pointing to the superclass and the opposite extremity denoting the subclass. The whole-part relation is also called aggregation (STAFF_MEMBER and DIVISION fig. 1). In the notation, the extremity of the line containing a black circle denotes the class constructed as the aggregation (whole) and the opposite extremity denotes the constituent (part). The use relation defines a client/supplier association (FAILURE_ON_ITEM and ITEM); the client is denoted by a white circle and the opposite extremity of the line denotes the supplier. Besides these relations, the cardinality of the different associations, denoting the number of links between each instance of the origin class and the instances of the destiny class, are represented as follows:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>N</td>
<td>0 .. N</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 .. N</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 .. 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 .. 7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 .. 3, 7</td>
</tr>
</tbody>
</table>

Exactly one
Zero or more (a limited number)
Zero or more
One or more
Zero or one
Specific range
Specific range or exact number

Two classes may be related by several associations. However, with respect to consistency checking, we can say that it is not possible to have at the same time the associations:
- aggregation and use
- aggregation and inheritance
between two classes.

Step 5: Identification of the abstract classes

In this step, according to the inheritance relations, the classes which do not generate instances, called abstract classes, are identified (ITEM fig. 1). This kind of classes are represented by a triangle enclosing a letter A within the cloud representing the class.

Notice that at this stage, it can be checked that a leaf class in the inheritance hierarchy cannot be an abstract class.

Step 6: Grouping into class categories

A class category is an aggregation of classes. The class category may also be nested. It is focused as a partition of the logical model of the system, represented by the class diagram. Similarly as the classes, the categories are identified with a name:
Some of the classes grouped into a category may be public, i.e. they can be used outside the category. Other classes can be part of the implementation, meaning that they cannot be used by other classes outside the category. The classes within a category are considered public by default.

There may exist use relations between class categories. The categories can group basic classes, in the sense that they are associated with a great number of classes in the application to be modeled, such as the object class of Smalltalk [Gol 84]. Booch notation allows to add the global label to the category, indicating it can be used by all the others, without denoting the lines representing the associations.

In this step it could be checked that a class cannot belong to two different categories.

We have to point out that these six steps in the process are considered as the analysis phase. The equivalence of Booch's and Coad&Yourdon's notations may be appreciated at this stage. Notice that at this point we can take Coad&Yourdon class diagram, transform it into Booch's, and resume the process at step seven.

The steps presented below are a refinement of the previous steps, focused to obtain a detailed design, much closer to the implementation phase.

**Step 7: Class refinement**

In this step, new classes may be added or existing classes may be transformed into: parameterized class, instantiated class, metaclass or utility class, shown in figure 3.

![Diagram of class refinement](image)

**Figure 3: Class refinement**
a) **Parameterized class**: denotes a family of classes, whose structure and behavior is defined independently from the classes representing formal parameters. The box with dotted lines, located in the upper right corner of the class cloud, denotes the formal parameter class.

b) **Instantiated class**: is a class where the formal parameter is substituted by the actual parameter. The notation for the instantiated class is the same as for the parameterized classes, with the difference that the borders of the box enclosing the actual parameter class is a solid-line box.

In this step we have to include also the instantiation relations between the parameterized and instantiated classes; the notation provides a new element shown in the example in figure 4 below:

![Diagram](https://via.placeholder.com/150)

**Figure. 4.** The LIST class parameterized by the COMPONENT class

The dotted-line arrow points to the parameterized class (parameterized by the formal parameter class ELEMENT in the example). The instantiated class maintains a use relation with the actual parameter class (COMPONENT in the example). A parameterized class does not have instances and cannot be used as a parameter. An instantiated class defines a new class, different from all the other concrete classes of its family, with distinct actual parameters.

With respect to consistency checking, notice that an instantiated class must be related to its parameterized class.

c) **Metaclass**: is a class whose instances are classes. The arrow goes from the class to its metaclass; this kind of relation is called meta relationship. A metaclass cannot have instances, but it may be related by any kind of associations with every other class.

d) **Utility Class**: Allows the logical grouping of free procedures or subprograms, i.e. that are not members of any class identified within the problem domain, as for example, the C++ *free subprograms* [Str 93]. In this sense, the semantics of an utility class is a logically related group of operations. It may also be seen as a class containing only variables and instance operations. The utility class permits only the use relation.

Classes may be related with any utility class by whole-part and use associations, but they cannot inherit from, nor contain instances of them. Similarly, an utility class may be associated with other classes by any kind of relation, excepting the inheritance relation.
Step 8: Refinement of whole-part relations

The whole-part relation does not necessarily imply a physical inclusion of the part into the whole. Particularly, we distinguish two types of inclusions:

a) *By value*: denotes a physical inclusion of a value of the part. This implies that the creation and destruction of the parts occur as a consequence of the creation or destruction of the whole. Graphically this refinement is represented by a white box at the end of the line denoting the relation.

A possible consistency checking here can be that a part class (related by value with its whole class) does not have relations with any other classes.

b) *By reference*: denotes a physical inclusion of the reference into the part. This implies that the life time of the part and the whole are independent. Its graphical representation is a black box at the end of the line segment denoting the relation. In the example, the relations between classes DIVISION and BUILDING may be refined as inclusion by reference.

Step 9: Refinement of attributes and methods.

During Step 3 of the analysis phase only the names of attributes and methods were identified, nevertheless it is necessary to provide more detailed information on them. Besides the name, an attribute belongs to a class and occasionally may have a default value. The notation for expressing these characteristics is presented below:

\[
\begin{align*}
A & : C \\
A : C & = E
\end{align*}
\]

Only name of the attribute
Only class of the attribute
Name and class of the attribute
Name, class and default value of the attribute

The following notation is proposed for defining the profile of the methods:

\[
\begin{align*}
N(\text{Arguments}) & \quad \text{Return class, method name and formal parameters} \\
R N(A) & \quad \text{Only the name of the methods}
\end{align*}
\]

Step 10: Specification of the access mode of the associations.

Each class must indicate the access mode to its resources. This access mode can be:

- Public
- Protected
- Private
- By implementation
The private access is the default access mode; the resources are accessible for all clients. Protected access indicates that the class is accessible only for subclasses, friend classes and the class itself. Private access indicates that the class may be accessed only for the class itself and for the friend classes. And access by implementation indicates that the class is accessible only for elements which are part of the class implementation and hence are not accessible by any other class.

The access mode is a characterization of the association, nevertheless this property may also be applied to any entity nested within a class. Particularly, a class may impose access restrictions to its attributes and methods.

Step 11: Constraints specification

A constraint establishes a semantic condition which has to be preserved, i.e. a class or relation invariant which has to be satisfied in order that the system reaches a steady-state. Notationally, the constraints are represented as expressions enclosed within { }, adjacent to the classes or relations to which they are applied.

3. PROCESS: CONSTRUCTION OF THE OBJECT DIAGRAMS USING BOOCH'S GRAPHICAL NOTATION

An object diagram is used to show the existence of objects and their relationships in the logical design of a system. It represents a snapshot in time of an otherwise transitory stream of events over certain configuration of objects. Each object diagram represents the interactions (messages) or structural relationships that may occur among a given set of class instances.

In general, the analysis of a system starts from the class diagram and continues with the development of object diagrams, instantiating the classes to obtain objects and relations, to express the behavior of the system. In other approaches, such as use-case analysis [Jac&Al 92], a set of scenarios is identified. For each scenario an object diagram may be constructed, each one corresponding at analysis level to the identified objects and the relations appearing as the functionalities of the system. In this way the construction of the object diagrams can be made before the class diagrams. Moreover the last one can be obtained from the objects diagrams. During design, object diagrams are used to express the semantic of interactions between objects in the logical design of a system.

Step 1: Identification of objects and relations.

The objects are represented as a solid-line cloud and the relations as solid lines. A relation between objects can be considered as an association's instance and is called Link. The existence of a link determines a communication path for interchanging messages.
Step 2: Naming the objects

In this step, names are associated with each one of the objects in the diagram. The syntax for the names is the same presented in the definition of attributes in the class diagram (step 9)

- A Object name only
- C Object class only
- A : C Object name and class

Step 3: Roles, keys and constraints

For certain object diagrams, it is useful to restate the role on the corresponding link between objects. The use of this adornment is optional and helps to explain the operation of one object upon another. Using the same notation presented in section 2 (steps 2 and 11) the keys or constraints associated with an object or a link may be indicated.

Step 4: Visibility between objects

In certain complicated scenarios it is useful to indicate exactly how the class instances can see one another. The notational elements are similar to the elements used to represent physical containment in class diagrams, but with the addition of letters expressing the kind of visibility. The following letters are used:

- G The supplier object is global to the client.
- P The supplier object is a parameter to some operation of the client.
- F The supplier object is a part of the client.
- L The supplier object is a locally declared object in the scope of the object diagram.

These letters are written inside an open box (meaning that the object's identity is shared) or a filled box (meaning that the object's identity is not structurally shared), as for physical containment in class diagrams.

The absence of visibility indication means that the precise visibility between the two objects is left unspecified.

The general use of these symbols is the representation of whole/part relationships between objects and the representation of transitory objects passed into the object that are used in the object diagram's scenario as parameters.

Step 5: Links characterization

In this step, relations involved in message exchange are indicated. Each message can be characterized in the following manner:

a) Direction: Indicating the orientation of the message with an arrow from the client to the supplier.
b) Operation: indicates the name used to call the operation or the activated event. The syntax used to define the operation invocation is similar to the one presented in Section 2(step 9).

\[
\begin{array}{l}
N() \\
R N(\text{Arguments}) \text{Return class, method name and formal parameters}
\end{array}
\]

c) Sequence number: to show an explicit ordering of events, an operation invocation or event dispatch may optionally be prefixed with a sequence number (starting at one). The sequence number is used to indicate the relative ordering of messages. A message with lower sequence number is dispatched before messages with higher sequence numbers.

**Step 6: Message Synchronization**

Different synchronization forms between objects and the identification of active objects is expressed in this step.

The direction used in the previous step for expressing message exchange, represents a sequential communication, but a way for expressing non-sequential synchronization must be provided. The different synchronization types and the notational elements are described below.

![Figure 5: Types of message synchronization](image)

a) Balking messages passing: the client will abandon the message if the supplier cannot immediately service the message.

b) Time out: the client will abandon the message if the supplier cannot service the message within a specified amount of time.

c) Synchronous: The client will wait forever until the supplier accepts the message.

In each of these three cases the client must wait for the supplier to completely process the message (or abandon the message) before control can resume. However, it is also possible to represent asynchronous messages, in this case the client sends the event to the supplier for processing, the supplier queues the message, and the client then proceeds without waiting for the supplier.

4. **Conclusions**

The proposed process enhances a systematic application of the notational elements defined in Booch's methodology [Boo 94], in order to construct the object model of applications, ensuring their correct use and consistency checking in each step. The absence of an explicit guide for the use of the graphical notation in Booch's methodology, has motivated the definition of our process. The defined process is a step towards the automatic transfer of information among different methodologies. In the
context of an automated tool, the process can be implemented for assisting developers throughout the analysis and design phase in the definition and refinement of class and object diagrams.

5. REFERENCES


Martha GONZALEZ received the Computer Science License in 1991, from the Univ. Central de Venezuela; she is MSc student in Computer Science and Instructor Professor at the School of Computer Science, Faculty of Science, Univ. Central de Venezuela. At present she is member of the ISYS Research Center and her research interests are in methodologies for software development and formalisms for object specifications.
E-mail: mig@conicit.ve.

Francis LOSAVIO received the Doctor degree in Computer Science in 1991 and a 3ème. Cycle Doctor degree in Computer Science in 1985 from the Univ. Paris-Sud, Orsay, France. She also received a MSc degree in Computer Science from the Univ. Simón Bolívar, Venezuela in 1983. At present, she is a Titular Professor at the School of Computer Science, Faculty of Science, Univ. Central de Venezuela and coordinates the ISYS Research Center. Her research includes software engineering environments, methodologies for software development, automatic program construction and formal specifications.
E-mail: flosavio@conicit.ve.

Alfredo MATTEO received the Doctor degree in Computer Science from the Univ. Paul Sabatier, Toulouse, France in 1984. At present, he is an Aggregate Professor at the School of Computer Science, Faculty of Science, Univ. Central de Venezuela and is member of the ISYS Research Center. His research includes software engineering environments, methodologies for software development, formal specifications and algorithmic complexity.
E-mail: amatteo@conicit.ve.

Christiane METZNER received the Master degree in Computer Science from the Univ. Simón Bolívar, Caracas, Venezuela in 1988. She is an Assistent professor at the School of Computer Science, Faculty of Science, Univ. Central de Venezuela and is member of the ISYS Research Center. At present she is director of the School of Computer Science. Her research includes software engineering methodologies, formal specifications and algorithmic complexity.
E-mail: cmetzner@conicit.ve.