INCREASING COMPREHENSIBILITY OF OBJECT MODELS: MAKING THE ROLES OF ATTRIBUTES EXPLICIT IN UML DIAGRAMS

PAULI BYCKLING1) PETRI GERDT1) LUDWIK KUZNIARZ2) JORMA SAJANIEMI1)

1)University of Joensuu, Department of Computer Science and Statistics P.O. Box 111, FI-80101 Joensuu, Finland

2)Blekinge Institute of Technology, Department of Software Engineering P.O. Box 520, SE-37225 Ronneby, Sweden
Ludwik.Kuzniarz@Bth.SE

Abstract. UML allows to produce and maintain object-oriented models of systems from different perspectives and viewpoints using the notion of a diagram. This paper describes how the comprehensibility of models can be increased by augmenting some UML diagrams with additional information about the behavior of attributes. A recent notion of “roles of variables” is used to describe various behaviors. The increased comprehensibility of two diagram types, class diagrams and sequence diagrams, is demonstrated by examples where the added role information reveals the behavior of attributes in a compact manner.

ACM CCS Categories and Subject Descriptors: D.2.2 [Software Engineering]: Design Tools and Techniques - object-oriented design methods; D.2.3 [Software Engineering]: Coding Tools and Techniques - object-oriented programming; F.3.3 [Logics and Meanings of Programs]: Studies of Program Constructs - object-oriented constructs

Key words: behavior of attributes, roles of variables, UML, comprehensibility

1. Introduction

UML (Unified Modeling Language) [Object Management Group 2006a] is widely used in software industry to model object-oriented systems at various levels of abstraction. UML has been designed with the goal of unifying the best features of various existing object-oriented modeling languages and notations, has been developed and defined by OMG (Object Management Group [2006b]), and is also accepted as an ISO (International Organization for Standardization) specification ISO/IEC 19501. However, UML is not free of problems (e.g., [Atkinson and Kühne 2002], [Dori 2002], [Fuentes et al. 2003], [Glinz 2000], [Guéhéneuc and Albin-Amiot 2004], [Kutar et al. 2002]) and its efficacy to support program understanding has been found to have limitations (e.g., [Arisholm et al. 2003], [Tilley...
Moreover, object models are becoming complex, tricky to design and hence often cumbersome, which makes them also hard to comprehend. In this paper, we are interested in UML used for constructing models of systems at the implementation level, that is models of programs. Our aim is to increase the comprehensibility of the models to programmers using the models.

UML contains a number of diagram types, each used to show the model from a certain perspective or purpose. Some diagrams describe a specific view of the static structure of a system, e.g., component diagrams describe the organization of physical software components, and class diagrams describe the class hierarchy, attributes and operations as well as relationships between classes. Dynamic aspects are illustrated in sequence and collaboration diagrams that describe message flow between objects, activity diagrams that describe control flow between activities, and statechart diagrams that describe the possible states of an object and the flow of control between the states.

None of the UML diagrams describe systematically dynamic aspects of attributes. The statechart diagram comes closest to this but it deals with the states and behavior of an entire object—rather than its individual attributes. Individual attributes are systematically described only in the class and object diagrams. The class diagram gives a static view consisting of only access and type information in addition to attribute names. The object diagram gives a static snapshot with values of attributes at some specific time. Moreover, notes can be added to any diagram, e.g., to explain program behavior with code fragments which, however, leads to a very low-level presentation.

Roles of variables [Sajaniemi 2002] form a compact classification that represents dynamic behavior of variables in a static way. For example the role stepper refers to any variable whose value changes in a predictable, systematic way, e.g., a loop counter. The set of roles is relatively small and easy to learn. Roles are actually a part of expert programmers’ tacit knowledge [Sajaniemi and Navarro Prieto 2005] and hence learning roles means only learning the names for these tacit concepts. In our experiments, programmers have learned to successfully apply the role concept in less than one hour [Sajaniemi et al. in press]. It has been found that teaching roles during elementary programming education improves students’ programming and program comprehension skills [Byckling and Sajaniemi 2006], [Sajaniemi and Kuittinen 2005], and educational institutions are currently adopting the role concept in their elementary programming courses. Thus it is supposed that in future, roles of variables will be familiar to programmers due to their basic education.

The purpose of UML diagrams is not to describe implementation details but to represent the structure and functioning of a system. A central part of these aspects should be the meaning and behavior of attributes. For example, it is important to understand whether an attribute is a unique, unchangeable entity or a counter that keeps track of the number of transactions—in the roles of variables terminology the former is a fixed value and the latter a stepper. The behavior of attributes is, however, not described systematically in UML diagrams—it can be done only by adding low-level program code fragments in notes. This paper suggests a systematic way to present this information at a higher level by adding information about the roles of variables in the diagrams. We also give examples that demonstrate how
the addition of role information increases the understandability of the diagrams.

The term “role” has been used in several contexts in object-oriented modeling. For example, Pernici [1990] presents a model where roles describe different behaviors of an object during its life-time. Object behavior is described separately for each role and the set of all roles is potentially unlimited. In a method known as Object-Role Modeling (ORM) [Halpin 1998] the world is pictured in terms of objects that play roles. In another method, OORASS (Object-Oriented Role Analysis, Synthesis and Structuring) [Reenskaug et al. 1992], a role represents the responsibilities of an object within the structure of collaborating objects. Kristensen and Østerbye [1995, 1996] present a role concept where roles are defined through perspectives, i.e., ways in which objects treat each other. Furthermore, in UML terminology a role is a named slot that represents the behavior of an element that participates in a context [Rumbaugh et al. 1999]. In all of the above uses, the set of different roles is determined ad hoc based on the properties and behavior of objects in the model under study. In the current paper, roles differ from the previous concepts in two respects. First, roles relate to the behavior of individual attributes instead of objects. Second, there is a fixed set of possible roles. This set is relatively small and it is based on tacit knowledge of expert programmers instead of more or less arbitrary properties of objects within a specific system.

The rest of the paper is organized as follows: Sect. 2 introduces the role concept and how it applies within the object-oriented paradigm. Sect. 3 describes how role information can be added to class diagrams and Sect. 4 does the same with sequence diagrams. Alternative notations and future work are then discussed in Sect. 5.

2. Roles of Attributes

Roles of variables is a recent concept that describes dynamic behaviors of variables [Sajaniemi 2002]. For example, consider the Java class Dog in Fig. 1. Objects of this class have two attributes: name and age. The attribute name does not change after initialization; its role is fixed value. The attribute age behaves differently. Once it has been initialized its future values will be known exactly: it will step upwards one by one even though we don’t know how far this counting will continue. A variable with such a behavior is said to be a stepper. Table I lists eleven roles that—according to an analysis [Sajaniemi et al. in press] of nine programming textbooks—cover 99% of variables in novice-level object-oriented, procedural, and functional programs. For the sake of paradigm independent terminology, the term data item is used instead of variable. Table I gives for each role a short, informal definition; for a more comprehensive treatment, see the Roles of Variables Home Page [Sajaniemi 2006].

A role describes the behavior of a data item. In particular, a role does not concern the way a data item is used in the program. For example, a stepper is a stepper whether it is used to reference elements in an array or in producing diagnostic output for debugging purposes. Thus, a role does not consider the way a data item is used but looks at the succession of values it will encounter and how this depends
public class Dog {
    private String name;
    private int age;
    public Dog (String n) { name = n; age = 0; }
    public void birthday () { age++; }
}

Fig. 1: A simple Java class.

Table I: Roles for object-oriented, procedural, and functional programming.

<table>
<thead>
<tr>
<th>Role</th>
<th>Abbreviation</th>
<th>Informal description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed value</td>
<td>FIX</td>
<td>A data item that does not get a new proper value after its initialization.</td>
</tr>
<tr>
<td>Stepper</td>
<td>STP</td>
<td>A data item stepping through a systematic, predictable succession of values.</td>
</tr>
<tr>
<td>Walker</td>
<td>WLK</td>
<td>A data item traversing in a data structure.</td>
</tr>
<tr>
<td>Most-recent holder</td>
<td>MRH</td>
<td>A data item holding the latest value encountered in going through a succession of unpredictable values, or simply the latest value obtained as input.</td>
</tr>
<tr>
<td>Most-wanted holder</td>
<td>MWH</td>
<td>A data item holding the best or otherwise most appropriate value encountered so far.</td>
</tr>
<tr>
<td>Gatherer</td>
<td>GAT</td>
<td>A data item accumulating the effect of individual values.</td>
</tr>
<tr>
<td>Follower</td>
<td>FOL</td>
<td>A data item that gets its new value always from the old value of some other data item.</td>
</tr>
<tr>
<td>One-way flag</td>
<td>OWF</td>
<td>A two-valued data item that cannot get its initial value once the value has been changed.</td>
</tr>
<tr>
<td>Temporary</td>
<td>TMP</td>
<td>A data item holding some value for a very short time only.</td>
</tr>
<tr>
<td>Organizer</td>
<td>ORG</td>
<td>A data structure storing elements that can be rearranged.</td>
</tr>
<tr>
<td>Container</td>
<td>CNT</td>
<td>A data structure storing elements that can be added and removed.</td>
</tr>
</tbody>
</table>
public class Sale {
    private List lineItems = new ArrayList(); /* Container */
    private Date date = new Date(); /* Fixed value */
    private boolean isComplete = false; /* One-way flag */
    private Payment payment; /* Fixed value */
    ...
}

Fig. 2: The class for a sale consisting of several line items.

public class Sale {
    private List lineItems = new ArrayList(); /* Container */
    private Date date = new Date(); /* Fixed value */
    private boolean isComplete = false; /* One-way flag */
    private Payment payment; /* Fixed value */
    ...
}

Fig. 3: The class of Fig. 2 augmented with the roles of the attributes.

on other data items.

In the object-oriented paradigm, roles apply to variables, attributes, value parameters, and objects that encapsulate a single conceptual attribute, e.g., String in Java. In the context of the current paper, however, only roles of attributes are considered. Attributes maintain the state of a class instance and understanding the behavior of attributes is crucial for understanding the behavior of objects. Roles provide a compact classification of possible behavior patterns for attributes and they can be easily expressed in documents and program code. Thus roles provide a convenient aid for the documentation and comprehension of object-oriented systems.

As an example, consider the class Sale that models a sale consisting of several line items. The code for this class starts in an object-oriented design textbook [Larman 2002, p. 315] as shown in Fig. 2. Based on this code fragment, it is impossible to say how the various attributes behave, e.g., do the attributes lineItems and payment change as the individual line items are being processed or do they express the final result only? In Fig. 3 roles are added for each attribute. It is now evident that the attribute lineItems is built during the processing of line items whereas the attribute payment is assigned only once.

In special cases, the role of an inherited attribute may be different in different sub-classes. For example, consider a class BankLoan with the attribute interestRate and two subclasses FixedRateBankLoan and VariableRateBankLoan. In the former the attribute interestRate is not changed after its initialization, hence it is a fixed value. In the latter subclass the rate can be changed in an unpredictable way, so the attribute interestRate is a most-recent holder. Understanding this
difference actually helps in understanding the class hierarchy itself.

3. Adding Role Information to Class Diagrams

A UML class diagram is a graphic presentation of the static view that shows a collection of declarative model elements, such as classes and their contents and relationships [Rumbaugh et al. 1999, p. 190]. In a class diagram, a class is usually presented with a separate compartment comprising the member attributes of the class. The information of an attribute consists of the name, a class or data type, and the visibility of the attribute. Thus the class diagram is a natural environment to present the roles of attributes along with the more traditional information.

There are, of course, various ways to include role information in class diagrams. For example, roles could be added after the type of an attribute. However, as we do not want to change the notational rules of UML, we suggest using additional compartments defined in the UML class notation. Additional compartments may be supplied to show pre-defined or user-defined model properties—for example to show business rules, responsibilities, variations, signals handled, exceptions raised and so on [Rumbaugh et al. 1999, p. 188]. The appropriate notation to represent role information in additional compartments would be the names of attributes on separate lines along with a role abbreviation (see Table I) for each attribute. The compartment may be labeled as “Roles of Attributes”.

We argue that systematic use of roles in class diagrams clarifies the general functionality and the purpose of a class by providing additional information about its key attributes, which maintain the current state of the instances of the class. Moreover, in some special cases of generalization, roles in the class diagram give substantial additional information about the implementation. For example, in cases where several classes inherit an abstract class and the behavior of the same inherited attribute differs in child classes, the role information is valuable in clarifying the difference. For these cases we suggest that an attribute whose role a super class does not want to fix is not listed in the role compartment of the super class and the role of the attribute in each child class is presented in the child classes only.

As an example, consider the class diagram in Fig. 4 that depicts a program, which implements a simple member register of a society. The society has two kinds of members: person members representing a single person, and community members representing a generic community of many persons. The program has basic member management operations such as add, remove, save, and restore. In addition, the user can search among the members by name and invoke a few basic reports. In order to keep the example simple, only a part of the methods are listed in the class diagram.

The program stores member information in the class MemberList as a collection of references to the abstract type Member. The dynamic class of a referenced instance is either PersonMember or CommunityMember, which represent the two member types. The CommunityMember class has a reference to MemberList, which it uses to store information about persons belonging to the community. The class MemberRegisterUI implements the user interface to the program. The user
interface class is connected to the class `MemberRegister`, which provides the interface to the member register and has methods for saving a record and opening previously saved records.

By looking at the traditional class view in Fig. 4, it is impossible to say, e.g., for what purpose the attribute `lastchange` of the class `MemberRegister` is used and how does it behave. For example, it might be the menu selection number of the user’s last action, the ordinal number of the latest change etc. Fig. 5, which is the same class diagram but now augmented with role information, shows the role of the attribute: it is a `stepper`. Thus it cannot store the latest action (as that would be a `most-recent holder`). In fact, it is the number of changes made so far—a typical use of `steppers`. Thus the role makes the behavior clear and suggests how the attribute
is used in the program.

In the example of Fig. 5, the abstract class **Member** has two attributes, **membership_fee** and **email**, which do not have a role within the context of the abstract class itself and, consequently, they are not listed in the role compartment. The roles are revealed in the role compartments of the subclasses, e.g., **membership_fee** is a *fixed value* in **PersonMember** and a *gatherer* within **CommunityMember**. The different roles reflect different behavior of the attribute in the subclasses: a person member has a fixed membership fee whereas the fee of a community member is gathered from its individual members. On the other hand, the attribute **email** behaves equally in the two child classes; it is a *fixed value* in both of them. It is not, however, listed in the role compartment of the abstract class, because that class has no functionality for the attribute. The role *fixed value* reveals that—due to the austerity of the model—an e-mail address cannot be changed in either subclass after its initialization. In principle, this could be inferred from the absence of the set and get methods, but often the set and get methods are not listed in class diagrams in order to save space. Thus the role of the attribute **email** makes this limitation of the system explicit.

Similarly, in cases where a child class overrides methods in such a way that the behavior of some attribute is different in the child class, role information is a powerful way to emphasize the different behavior.

The other end of the association between classes **MemberRegisterUI** and **MemberRegister** is labeled with **memberregister** that describes the (UML) role of the association end. This must not be confused with the role of the attribute **memberregister**, which has the role of a *fixed value* because it refers the same **MemberRegister** instance all the time. If the **memberregister** attribute could refer different instances at different times, its role would be *most-recent holder*. Thus the role of an attribute describes its dynamic behavior whereas the role of an association end describes an aspect of the static structure of the class relationships.

Roles can be added also to object diagrams, which are snapshot images of the system at a point in time. In object diagrams, attributes are presented similarly as in class diagrams with the exception that in an object diagram the values of attributes can also be shown. Thus the same method of additional compartments used with class diagrams can be applied to object diagrams as well. As object diagrams show object instances, all attributes have a known role even in cases where the role of an inherited attribute may differ in subclasses.

4. Adding Role Information to Sequence Diagrams

The purpose of a sequence diagram in UML is to describe interactions between objects in a sequential order, i.e., to show how runtime objects carry out a use case scenario by sending messages to each other. In a sequence diagram the chronological development and the lifeline of an object within an interaction scenario are presented explicitly and thus it is a natural environment to present also the behavior of attributes within the sequence. However, most sequence diagrams present only the stack of method activations in a particular use case scenario together with the
activation lines of objects. Single attributes can be visible in constraint notes, but the whole life cycle of an attribute within the sequence is not presented in any way.

We argue that by adding information of the behavior of relevant attributes in sequence diagrams, useful additional information will be provided. Of course, the roles may also be found in the class diagram, but to minimize the need to switch back and forth between diagrams, addition of role information for selected attributes in sequence diagrams is justified.

In our proposal the presentation of role information in sequence diagrams is twofolded. Firstly, the essential attributes are introduced along with their roles in a note box above the object. The roles describe the behavior of attributes and suggest their purpose.

Secondly, uses of attributes (i.e., references to them) and changes of attributes (initializations and updates) are presented in note boxes that contain the type of the action (refer, initialize, update), the name of the attribute and the abbreviation of its role. The note boxes are connected to the activation boxes that represent the activity that triggers the actions. The actual implementations of initialization and update actions are role dependent, thus the role abbreviations work as high level information about how an attribute is initialized or updated without the need to go into unnecessary details. For example, a gatherer may be updated with a simple addition or with some more complicated computation; for the reader of a sequence diagram it is however sufficient to know that the attribute is a gatherer that is being updated. A detailed description of the update—for example in the form of a program code fragment—would only make the basic behavior of the attribute harder to grasp.

Fig. 6 shows the sequence diagram for a scenario where a user adds a single person member to the member register. The case starts with the user invoking user interface controls that begins the addition. Then the program asks the user to enter the data of the new member. The data is sent to a MemberRegister instance with the message addMember. The method addMember creates a new PersonMember instance and inserts it into the members container of the MemberList instance using the method add. In order to find the correct insertion point, the MemberList instance asks the birth year of the new PersonMember instance using the getYear message.

The sequence diagram in Fig. 6 shows explicitly the sequence of messages activating objects, but the way method calls affect the state of an individual object can not be implied. Fig. 7, conversely, displays the sequence diagram with additional role information for essential attributes. For example, it shows how the activation of the MemberList instance affects the attribute itemcount. The role of a stepper (indicated by the abbreviation STP in the note boxes) reveals that itemcount goes through a predictable succession of values. Thus, Fig. 7 shows that within the scenario the attribute is updated and referred once, and the update is some operation typical for steppers.

Three other instances in the sequence diagram of Fig. 7 have attributes that participate in the scenario of adding a new member. The PersonMember instance named m is created as the information about the new member is sent to the MemberRegister instance. A note box added to the creation message indi-
cates that all attributes of \( m \) are initialized during creation, i.e., in the constructor. The note box recurs the role of the attributes that are initialized, in this case they all are fixed values. Thus the diagram reveals that the attributes of \( m \) are not modified after the object instantiation. Finally, the attributes \( \text{cmd\_log} \) of class \( \text{MemberRegisterUI} \) and \( \text{lastchange} \) of class \( \text{MemberRegister} \) are both updated at the end of the scenario illustrated in the diagram.

The use of role information can be applied also to collaboration diagrams as well as to sequence diagrams—the roles of the most important attributes can be shown similarly to above. However, collaboration diagrams do not show time as a separate dimension and the sequence of messages must be determined using sequence
numbers [Rumbaugh et al. 1999, p. 203]. Thus also events related to the behavior of attributes (presented in commentary note boxes) must be linked to a message with a sequence number.

We emphasize that it is not necessary to add role information to all attributes of all objects, but only on those attributes whose behavior is essential in the presented cases.

5. Discussion and Further Work

In this paper, we have presented how the behavior of attributes can be compactly described with the notion of roles of variables; and how roles can be expressed in UML diagrams. We have also given examples of class and sequence diagrams which demonstrate that the added role information really increase the comprehensibility of an object model. The example model may not be optimal and it might benefit from refactoring but this is rather common in practice and makes comprehension of models tough in general. The example diagrams give thus evidence of improved understanding obtained through the addition of role information.

Roles facilitate the communication of low-level program information at a higher level. With the use of role names, the need for low-level program code fragments in UML diagrams is decreased while the information content of the diagrams is increased. By a careful selection of attributes whose roles are described, sequence diagrams become easier to understand. In class diagrams, we recommend describing roles for all attributes. We have used class and sequence diagrams as examples for making the roles of attributes explicit but the same techniques can be applied, e.g., in adding role information into object diagrams.

The idea of assigning roles to attributes was adopted from the existing concept of roles of variables used to describe the behavior of individual variables [Sajaniemi 2002]. Our intention was to introduce the role concept into UML in a way that would be easy to comprehend: simple, intuitive and as close to its origin as possible. So the original conventions for naming of the roles were preserved. Working on this basis we have addressed two issues and related questions: where to include the role information and how to do it.

The role information should be indicated both at the place where the attribute is declared and at the place where the attribute is used and potentially updated. The first is the class definition and the second is an object participating in a collaboration. The standard diagram for defining classes is the class diagram and we decided to extend it with the definition of roles; another option would be to introduce a new diagram type for the definition of the behavioral aspects of attributes defined in class diagrams. The selected solution seems to be more convenient for a user as it makes it possible to look at various parts of a class diagram to obtain information at various abstraction levels without a need to switch between two types of diagrams. When collaborations are concerned, they can be expressed as sequence and collaboration diagrams that contain basically the same information but expressed in a slightly different way. In our proposal we decided to add role information into sequence diagrams as it is more in line with the recent changes in the UML.
specification [Object Management Group 2006a].

UML offers several potential solutions for the question of how to express role information: class compartments, notes, tagged values, and stereotypes. Stereotypes would probably be the most elegant solution from the perspective of language engineering but our main concern in this paper was to find an intuitive yet legal introduction of the basic ideas and to open the issue for further elaboration. That is why we looked at the three lighter approaches—class compartments, notes, tagged values—as a way to include roles in class diagrams. The introduction of a separate role compartment is in line with the class definition style because it groups the role information in one coherent place in much the same way as the static information of attributes and the behavior of the entire class are grouped. The use of notes or tagged values would considerably dim the class diagram and, hence, separate role compartments were selected. In sequence diagrams, on the other hand, notes are the simplest and a natural way to add information and they were chosen for sequence diagrams in order to meet our comprehension goal.

The next steps in the introduction of the concept of roles of attributes would be an attempt to formalize the concept within the UML extension framework, to investigate a possibility of extending the ideas to other modeling elements such as associations and properties in general, and to study the practical importance of the extension, that is: are software engineers willing to use the suggested new notation and do they find it helpful when working with UML diagrams?

Acknowledgements

This work was supported by the Academy of Finland under grant number 206574.

References


