IDEA GROUP PUBLISHING

701 E. Chocolate Avenue, Suite 200, Hershey PA 17033, USA
Tel: 717/533-8845; Fax 717/533-8661; URL-http://www.idea-group.com

ITP4863

Meta-Model Patterns in Object-Oriented Analysis

Frank Devos and Eric Steegmans
Department of Computer Science, Katholieke Universiteit Leuven
Celestijnenlaan 200A, 3001 Leuven, Belgium
{frank.devos,eric.steegmans}@cs.kuleuven.ac.be

ABSTRACT

A major purpose of analysis is to represent precisely all relevant facts, as they are observed in the external world. A substantial problem in object-oriented analysis is that most modelling languages and methods are more pointed at building computational models than conceptual models. It is a very blind assumption that concepts that are convenient for design can also be applied during analysis. Meta-model patterns describe the adequateness and the use of concepts in modelling languages. In this paper, concepts as queries, attributes, invariants and preconditions will be evaluated as possible building stones for conceptual modelling. Furthermore, a new concept is proposed to specify properties or behaviour where in more than one object is involved.

1. INTRODUCTION

A major purpose of analysis is to represent precisely all relevant facts, as they are observed in the external world. A substantial problem in object-oriented analysis is that most modelling languages and methods are more pointed at building computational models than conage (UML) [13], are more pointed at building computational models than conceptual models. As defined in [1, p.4], a computational model describes a software product and is the output of design activities. Perhaps the most difficult aspect of analysis is avoiding software design [3]. It is a very blind assumption that concepts that are convenient for design are also valid for analysis. As stated in [8, p.144], it is unwise to try to design requirements modelling languages by merely adopting programming language ideas.

Meta-model patterns describe the of use concepts in modelling languages whereas patterns rather describe useful models. Meta-model patterns can also introduce new concepts and forbid others. As an example, Fowler [6, p.305] introduces a new concept for modelling the history of values.

Do these patterns augment the quality of a conceptual model? From our point of view, meta-model patterns for analysis help to model reality and form part of an analysis method. An analyst has to focus on the domain problem rather than concentrating on possible concepts and alternatives to model reality. Meta-model patterns for analysis lead the analyst in building conceptual models instead of computational models. These patterns promote the separation of concerns during analysis and design.

A good analysis method imposes rules on the concepts it offers. A decision of an analyst on *how* to represent an observed fact indicates a lack in the analysis method. From our point of view, the ultimate goal of an analysis method is to guide the analyst specifying precisely all relevant real world facts without any design or modelling concern.

2. A META-MODEL PATTERN FOR SPECIFYING PROPERTIES

In this paper, a property is defined as a relationship between a class and a data type. The state of an object of a class can be changed; the state of a data value of a data type can never be changed [13, p.2-113].

Queries and attributes are competing concepts at the level of analysis. They can be both used to model a relationship between a class and a data type. Queries offer advantages compared with its competitor. The advantages are shortly described in this pattern. Attributes are not needed in object-oriented analysis. This simple pattern helps the analyst to model properties and promotes the maintainability of the model. It eliminates the possible modelling alternatives for an analyst.

Borgida [2, p.1] states that an analyst should not model the way data is stored in a computer. In our opinion, an attribute is rather a design-oriented and programming concept. An attribute suggests data representation while data representation is clearly a design issue.

The concept of a query is semantically richer than the concept of an attribute. Every relationship between a class and a data type can be expressed as a query but not as an attribute. The analyst has no choice between an attribute and a query when explicit arguments are involved in modelling a property. Only queries can have explicit arguments. We are in favour of using as few concepts as possible in the analysis phase for simplicity reasons.

Some analysts use queries to model derived properties and attributes to model non-derived properties. However, this way of modelling of both types of properties compromises the extendibility of the conceptual model. When a non-derived property becomes a derived property, an attribute has to be deleted and a query has to be introduced. The interface of the class is changed, although, the property as such still exists. The transformation from a non-derived property in a derived property is a well-known phenomenon in case a model is extended. Consequently, methods where the interface of a class does not change when the property becomes derived are preferred.

Notice that in this paper an association was not considered as an alternative to model a property. In [14, p.170] the authors argued that for relationships between classes associations are used because it is important to see the relationship in both directions while for relationships between a class and a data type the latter is usually subordinate to the class and has no knowledge of it.

In the next patterns, the queries will be modelled in the first compartment of the class. The second compartment of a class will only contain events.

3. A META-MODEL PATTERN FOR SPECIFYING MESSAGES

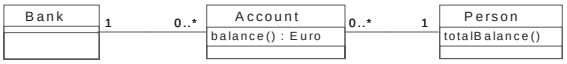
One of the most specific concepts in object orientation, aside inheritance, is that one can send messages to an object. This principle is known as the *message paradigm*. The message paradigm forces the software engineer to associate a message with a class. This leads to a more consistent structure.

The message paradigm in object-oriented analysis works well as long as one object is involved. Problems arise when none or more than one object is involved in a message. The next pattern describes how to model messages where in more than one object is involved. The question rises to which class the message will be associated.

Consider the example in the above figure. Assume that a property returning the total balance of all the accounts of a specific person at a specific bank is relevant. The query totalBalance() could be attached to

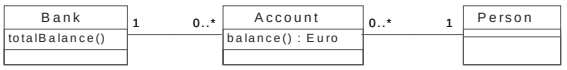
Fig. 1. *Messages with Implicit and Explicit arguments or N-ary Messages*

Alternative One



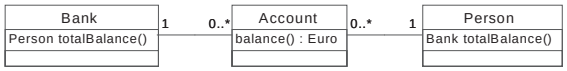
context Person :: totalBalance(bank : Bank) post:
result= self.account()®select(account|account.bank() = bank).balance()®sum

Alternative Two



context Bank :: totalBalance(person : Person) post:
result=self.account()®select(account|account.person()= person).balance()®sum

Alternative Three



context Bank and Person :: totalBalance() : Euro post:
result=(selfBank.account()®intersection(selfPerson.account()).balance()®sum

the class Person or to the class Bank. The choice of the context during analysis will be rather arbitrary. Also looking at the different specifications of the postconditions will not help: the specifications are each other's mirror image.

Two solutions are possible, modelling the same external world. In our opinion, the decision to which class a message with more than one object involved belongs, adds nothing new to the model about the facts in the external world. This decision is clearly a design decision and should not be taken during object-oriented analysis.

For that reason messages with more than one implicit argument are introduced. These messages are called binary messages or more general *n-ary messages* [4, p.184]. Binary messages in conceptual models are illustrated for the bank case in the third alternative. Aside the introduction of a new concept, the pattern forbids the use of objects as explicit arguments. As a result, the analyst has only one possible way to model the requirements of the domain. Notice that the use of class-scoped messages is also forbidden. This excludes the alternative of a class-scoped message with two explicit arguments attached at the class Account.

Instead of one context class, there exist two context classes for the binary message totalBalance(). The implicit arguments are referenced by selfBank and selfPerson. More important is that also the specification for the postcondition is changed. Instead of starting with one object and navigating through the model, the postcondition has two starting objects. This way of modelling is more declarative and abstract in the sense that it does not suggest an implementation strategy.

A typical application of this concept is the linking of two or more objects. Consider the relationship among persons and cars. How should the acquisition of a car by a given person be reflected in term of messages? Is a person buying a car? Or is a car bought by a person? Should there be a message applicable to a person, to a car or to both? When using n-ary messages the answer is already given and the question becomes irrelevant.

One could argue that an analyst does not have to make a choice between different classes if a data value instead of an object is used as an explicit argument. In the example, one could have associated a message totalBalance(nameOfTheBank : String) to the class Person, assuming that the name of the bank is unique. In our approach, other patterns exist to avoid this modelling option.

4. A META-MODEL PATTERN FOR SPECIFYING CONSTRAINTS

In the external world, an analyst observes human or business-imposed, physical or legal laws, rules and regulations. The UML provides the Object Constraint Language (OCL) to express constraints in a precise way [13, p.6-2]. The OCL offers preconditions, postconditions and invariants as (traditional) concepts for expressing constraints [13, p.6-5]. Notice that the use of invariants, pre- and postconditions during the analysis phase suggests in no way an implementation strategy [13, p.6-1].

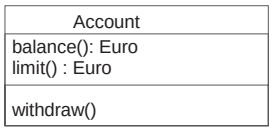
During analysis, an invariant is defined as a feature that must be true at each moment in time, without determining how and when this feature must be controlled [7, p.110] [15, p.393]. Due to the operational nature of design, the definition of an invariant during the design phase has to be less restrictive: an invariant must be true at all times, except during the execution of an operation. Warmer and Kleppe, the main authors of the OCL, seem to adopt this design view on an invariant. Their initial statement that "an invariant must be true all the time" [16, p.4] has been rephrased as "the invariant must be true upon completion of the constructor and every public method but not necessarily during the execution of methods" [10].

The differences in the definition of the concept of an invariant illustrate the need for different concepts during analysis and design. Both phases have different goals. Because of these different goals, different concepts are needed.

As defined, at the level of analysis an invariant must be satisfied at any time. However, problems arise as soon as events must be specified. The specification of an event can contradict the specification of an invariant. Two main approaches exist to solve this problem. The approaches are illustrated in the second figure.

A first approach consists in a more detailed specification of the event that eliminates the contradiction. The conditions for the effect of the event will be stricter. These extra specifications violate the requirement of modelling a real world fact only once. If the specification of n events can possibly contradict the invariant, the real world restriction is represented in n+1 places: once as invariant and n -times as a condition for the effect of the event. This approach does not support adaptability and extendibility. Traditionally, these conditions are modelled with an implication or a precondition.

Fig. 2*. Alternatives respecting the invariant*



context Account inv: *balance >= limit*
context Account :: withdraw(amount : Euro)

First Approach
post: *(balance()@pre – amount >= limit()) implies balance() = balance()@pre – amount*
— or
pre: *balance() - amount >= limit()*
post: *balance() = balance()@pre – amount*
Second Approach
— the principle of non-violation is adopted.
post: *balance() = balance()@pre – amount*

The last approach is to adopt the principle of non-violation. This principle states that an event, which is about to violate an invariant, does not change the state of the model. This principle supports also the adaptability and extendibility of the model. When invariants are added to the model, the specification of events must not be changed. In addition, events can be easily added to the model without examining all possible conditions under which they may violate invariants. This approach is less design and operational oriented than the first one. To avoid possible contradictions or extra specifications the principle of non-violation will be adopted.

5. CONCLUSIONS

Real world facts are modelled during analysis whereas software systems are modelled during design. This makes it a blind assumption that concepts for design are also suitable for analysis. Furthermore, this distinction between analysis and design causes difficulties giving one valid definition of a concept for both phases of the software life cycle. Meta-model patterns in object-oriented analysis try to describe the adequateness and the use of concepts for analysis purposes. These patterns form part of an analysis method and helps the modeller to eliminate design concerns during analysis.

REFERENCES

1. BOOCH G., *Object-Oriented Analysis and Design with Applications*, 1994, The Benjamin/Cummings Publishing Company, Inc.

2. BORGIDA A., "*Features of Languages for the Development of Information Systems at the Conceptual Level*", IEEE Software, 2(1), January 1985, pp. 63-73.

3. DAVIS A., *Software Requirements: Objects, functions and states*, 1993, Prentice-Hall.

4. DEVOS F., STEEGMANS E., *The Message Paradigm in Object-Oriented Analysis*, The 4th International Conference on the Unified Modeling Language, LNCS 2185, 2001, pp.182-193.

5. DIESTE O., JURISTO N., MORENO A., PAZOS J. and SIERRA A., *Conceptual Modelling in Software Engineering and Knowledge Engineering: Concepts, Techniques and Trends*, Handbook of Software Engineering and Knowledge Engineering, World Scientific Publishing Company, 2000.

6. FOWLER M., *Analysis Patterns, Reusable Object Models*, 1997, Addison-Wesley.

7. GOGOLLA M. and RICHTERS M., *On Constraints and Queries in UML*, The Unified Modeling Language, Physica-Verlag, 1998, pp. 109-121.

8. GREENSPAN S., MYLOPOULOS J. and BORGIDA A., *On formal Requirements Modeling Languages: RML revisited*, International Conference on Software Engineering, 1994, pp. 135-147.

9. HENNICKER R., HUSSMANN H. and BIDOIT M., *On the Precise Meaning of OCL Constraints*, Advances in Object Modelling with OCL, LNCS 2263, 2002, pp. 69-84.

10. KLASSE OBJECTEN, *Errata for The Object Constraint Language, Precise Modeling with Unified Modeling Language*, 2002, http://www.klasse.nl/english/boeken/ocl-intro.html.

11. LISKOV B. and WING J., *A Behavioral Notion of Subtyping*, ACM Transactions on Programming Languages and Systems, Volume 6, November 1994, pp.1811-1841.

12. MEYER B., *Object-Oriented Software Construction*, 1997, Prentice Hall.

13. OBJECT MANAGEMENT GROUP, *Unified Modeling Language Specification*, Version 1.4, 2002.

14. RUMBAUGH J., JACOBSON I., BOOCH G., *The Unified Modeling Language Reference Manual*, 1999, Addison-Wesley.

15. VAN BAELEN, S., LEWI, J., STEEGMANS, E. and SWENNEN B., *Constraints in Object-Oriented Analysis*, Object Technologies for Advanced Software, Lecture Notes in Computer Science, Volume 742, 1993, pp. 393-407.

16. WARMER J. and KLEPPE A., *The Object Constraint Language, Precise Modeling with Unified Modeling Language*, 1999, Addison-Wesley.