IMPORTANCE OF OBJECT CONSTRAINTS IN SOFTWARE DEVELOPMENT

Meena Sharma¹ and Rajeev G Vishwakarma²

¹Department of Computer Engineering, IET-Devi Ahilya University, Indore, India
meena@myself.com

²Department of Computer Science & Engineering, SVITS, Rajiv Gandhi Technical University, Indore, India
rajeev@mail.com

ABSTRACT

The Object constraints can be described as the expressions that are used to insert important data in object oriented models. The Object Management Group founded a worldwide standard for object-oriented analysis and design artifacts specifically the diagrams. The specification and standard, known as the Unified Modeling Language, comprises model diagrams and the allied and associated semantics. Unified Modeling Language is meant for modeling and, the Object Constraint Language is the specification and standard for specifying expressions. These expressions add essential, crucial and critical information to object-oriented models and other object modeling workproducts. In Unified Software Development Process we have analysis & Design discipline to have complete architecture and design of the system. Analysis & Design discipline is followed by implementation discipline. The activities of the implementation phase are mainly captured in construction phase. In unified software development life cycle the expressions in the design model are forward engineered to produce or generate the source code. The source code that is generated depends a lot on the platform and the technology that are selected for the software development. We have investigated how the expressions are developed and incorporated in the models in an electronic business solution. Further object constraint language is used to form the expressions that are attached to the object oriented model and artifacts. Then models are forward engineered to generate the code. In a nutshell we produce code from abstract, models or diagrams making use of object constraints language that is to achieve round trip engineering format between model and code. In this paper we have generated expressions from object constraint perspective for a reward point system as applied to a customer in e-business. We have also evolved the Context Definition, Initial Values & Derivation Rules, Query Operations, Attributes and Operations in this regard.

KEYWORDS

UML, OCL, Model, Object Oriented Analysis & Design, Unified Software development Process, Context Definition, Initial Values & Derivation Rules, Query Operations, Attributes and Operations

1. INTRODUCTION

This This part deals with the essential elements that we need to write object constraints. This should be noted down that we can mark the object constraints with the help of the unified modeling language. We consequently hereby explain Expressions, Types, and Values. In Object Constraint Language (OCL), every value, either it is an object, or a component instance, or a datavalue, has a particular type, which defines the process or function that can be applied to the object. Types in OCL are divided into two groups- Predefined types, as defined in the standard library, (including the: Basic types and Collection types) and User-defined types.

The predefined basic types are Integer, Real, String, and Boolean. Their descriptions are similar to those in many acknowledged languages. The preclassified collection types are Collection, Set, Bag, OrderedSet, and Sequence. They are utilized to state the accurate results of a navigation through associations in a class diagram. We need to be familiar with these types to write more

complex expressions. User-defined types, such as Customer or RewardScheme, are defined by the user in the UML diagrams. Every model element that can be instantiated (for example each class, interface, component, or datatype) - in a UML diagram becomes a type in OCL also automatically. Each OCL expression either a user-defined type or a predefined OCL type symbolizes a value; consequently, it has a type as well. Each OCL expression has a result: the output that results from evaluating the expression. The type of the result value is equivalent to the type of the expression. OCL marks the difference between value types and object types. Both are types, and both denote instances, but there is one significant differentiation: value types define instances that never modify. For example, integer with the value 10 will never change its value that is it will not become an integer with a value of 100. Object types, or Classifiers, stand for types that define instances. The value of the instances can be changed over time and states. An instance of the class Customer can amend the value of its attribute (for example name) and even then it may remain the same instance. For instance, Martin Fowler [Fowler, 97] calls object types as reference objects and value types as value objects.

One more main quality of value types involves identity. For value types, the value identifies the instance, consequently the name. Two incidences of a value type that have the same value are by definition and other by the same instance. Two incidents of an object type are the same instance if they have the same object identity. In other words, value types take account of value-based identity; object types encompass reference-based identity. Value types include the predefined basic types and the predefined collection types of OCL as well. The user defined types may be classified by either value types or may be object types. UML datatypes are value types that include enumeration types. Object types include UML classes, components, and interfaces are object types. This part deals with the essential elements that we need to write object constraints. This should be noted down that we can mark the object constraints with the help of the unified modeling language. We consequently hereby explain Expressions, Types, and Values. In Object Constraint Language (OCL), every value, either it is an object, or a component instance, or a datavalue, has a particular type, which defines the process or function that can be applied to the object. Types in OCL are divided into two groups- Predefined types, as defined in the standard library, (including the: Basic types and Collection types) and User-defined types.

The predefined basic types are Integer, Real, String, and Boolean. Their descriptions are similar to those in many acknowledged languages. The preclassified collection types are Collection, Set, Bag, OrderedSet, and Sequence. They are utilized to state the accurate results of a navigation through associations in a class diagram. We need to be familiar with these types to write more complex expressions. User-defined types, such as Customer or RewardScheme, are defined by the user in the UML diagrams. Every model element that can be instantiated (for example each class, interface, component, or datatype) - in a UML diagram becomes a type in OCL also automatically. Each OCL expression either a user-defined type or a predefined OCL type symbolizes a value; consequently, it has a type as well. Each OCL expression has a result: the output that results from evaluating the expression. The type of the result value is equivalent to the type of the expression. OCL marks the difference between value types and object types. Both are types, and both denote instances, but there is one significant differentiation: value types define instances that never modify. For example, integer with the value 10 will never change its value that is it will not become an integer with a value of 100. Object types, or Classifiers, stand for types that define instances. The value of the instances can be changed over time and states. An instance of the class Customer can amend the value of its attribute (for example name) and even then it may remain the same instance. For instance, Martin Fowler [Fowler, 97] calls object types as reference objects and value types as value objects.

One more main quality of value types involves identity. For value types, the value identifies the instance, consequently the name. Two incidences of a value type that have the same value are by definition and other by the same instance. Two incidents of an object type are the same instance if

they have the same object identity. In other words, value types take account of value-based identity; object types encompass reference-based identity. Value types include the predefined basic types and the predefined collection types of OCL as well. The user defined types may be classified by either value types or may be object types. UML data-types are value types that include enumeration types. Object types include UML classes, components, and interfaces are object types.

2. OCL

OCL is a kind of modeling language that facilitates construct software models. It is identified as a customary add-on to the UML. UML is the Object Management Group (OMG) standard for object-oriented analysis and design. The expressions developed in OCL rely on the types like the classes, interfaces, and so on. These types are defined in the UML models and diagrams. The use of OCL therefore requires that we should have input of some features of UML.

Expressions written in OCL attach essential information to object-oriented models and other artifacts pertaining to object modeling. This information often cannot be put across in a diagram. Earlier in UML, this expression writing was limited to constraints, where a constraint is defined as a control or restriction on one or more values or their part of an object-oriented model or system. Later on in UML 2, the perception and view is that more and more additional information should be included in a model along with the constraints alone. With the help of UML and OCL we can associate constraints and information in a model for defining queries, referencing values, or stating conditions and business rules in a model. In order to write these expressions in a clear and unambiguous manner we make use of OCL.

The OCL version 2.0, has been formally classified in the Object Constraint Language Specification [OCL03] and it has been very well accepted by the OMG. The, models should be good quality, firm, dependable, and logical. Using the mixture of UML and OCL, we can build such models.

3. UNIFIED DEVELOPMENT

We have researched and produced an object constraint code framework for an e-business application in unified paradigm. Object constraints are employed as the formalization of the specifications in the model. The Unified Process (UP) is not just a process, but rather an expandable framework which may be tailored and customized for specific organizations or projects. Now we describe the important characteristics of UP. It is iterative and incremental. The Unified Process is an iterative and incremental development process. The Elaboration, Construction and Transition phases are divided into several time boxed iterations. The Inception phase may also be partitioned in to more than one, iterations for a large project. Each iteration results in an increment or executable, which is a release of the system that has additional or improved and enhanced functionality compared to the earlier release. Even though most iterations will comprise work in most of the development process disciplines (Business Modelling, Requirements, Analysis & Design, Implementation, Testing and others) the relative work, tasks and importance will change over the life of the project. UP is use case driven. In the Unified Process, use cases are the key elements to capture the functional requirements and to characterize the contents of the iterations. Each of the iterations needs a set of use cases or use case scenarios from requirements discipline to the implementation discipline, test and deployment.

UP is architecture centric. The Unified Process persists that architecture resides at the heart of the software development life cycle and project team's efforts to accomplish the system. Since no

single model is adequate to wrap up all aspects of a system, the Unified Process allows multiple architectural models and views. One of the most significant deliverables of the unified process is the executable architecture baseline. This architectural baseline is developed during the Elaboration phase. This partial implementation of the system is necessary to validate the architecture and serve as a basis for rest of the development. UP is risk focused. The Unified Process necessitates the project development team to focus on dealing with the majority of critical risks at an early stage in the project life cycle. The deliverables of each iteration, particularly in the Elaboration phase, have to be chosen in order to make certain that the maximum risks are addressed and attended first. Now we will focus on the fact why we should necessitate the object constraint.

4. CONTEXT DEFINITION

The following The context definition identifies and states the model entity for which the OCL expression is defined. Typically this is a class, interface, datatype, or component. Sometimes it may be an operation, and rarely is it an instance. It is always a particular element defined in a UML model or diagram. This element is termed the context of the expression. OCL expressions can be integrated and incorporated simply in the model straight forward in the diagrams. OCL expressions can also be made available in a separate text file. In both the cases it contains a context definition. In the UML diagram, a dotted line is used to symbolize the context definition that connects the model element and the OCL expression. In Figure 1, we show the expressions and their contexts.

Figure. 1. Expressions & their contexts

5. CASE STUDY

As a real world case study of an ebusiness, we have modelled a computer software system for a imaginary company called Award and Reward (A&R). A&R manages rewardreward point programs for business organizations that offer their customers various kinds of shopping and service rewards. Often, the extras take the form of reward points or air miles, petrol, gas and many others. But other rewards on the basis of rewards points collected are possible as well. For example: discounted rates, a luxury sedan rental for the equal price as a standard rental car, additional or ad-on service on an airline, and so on. Anything a company is willing to offer can be a service rendered in a reward program. Figure 1 shows the UML class model that A&R employs for most of its customers The model shown here is a classic platform independent model from the

perspective and context of software architecture. It does not demonstrate any dependency on the programming language to be applied to make the system.

The key and fundamental class in this model is RewardProgram. A system that administers one reward program will hold only one instance of this class. In the case of A&R, many instances of this class will be present in the system. A company that offers its customers membership in a reward program is called a ProgramaAssociate. Several companies can join the same program. In this case, customers who join the reward program can make good profit from services offered by any of the member companies.

Every customer of a program associate can join the reward program by submitting a form and receiving a membership card. The objects of class Customer represent the persons who have joined the program. CustomerCard class represents membership card. Where, the card is issued to one person. Card utilization is not checked, so one card could be used for all the members of the family or business. Most reward pro-grams allow customers to save reward points. Each individual program associate chooses when and how many reward points are allocated for a particular purchase. Accumulated reward points can be used to "buy" specific services from one of the program associate. To account for the reward points that are gathered by a customer, every membership can be associated with a RewardAccount. A variety of transactions on this account are potential. For example, in the RewardProgram, Silver and Gold CustomerCard has four program partners: a food chain, a line of petrol pumps, and an airline service. At the food chain outlet, the customer can use reward points to pay the food bill. The customer earns five reward points for any regular purchase over the amount of Rs. 100. The petrol pump stations offer a discount of 5 per cent on each petrol filling. Customers can save reward points for free flights with the airline company. For each flight that is usually paid by the customer, the airline offers two reward point for each 20 miles of flight.

In these circumstances, there are two kinds of transactions. First, there are transactions in which the customer accumulates reward points. In the model (Figure 2, this type of Transaction is modelled by a subclass of the class Transaction called Earning). Second, there are transactions in which the customer uses reward points for purchasing. In the diagram of class model, they are represented by instances of the subclass Redeem of Transaction class. The earnings generated by the customers need to be recorded. This is recorded as two simultaneous transactions on the RewardAccount, one Earning and one Redeeming for the same number of points.

6. INITIAL VALUES & DERIVATION RULES

A very fundamental addition to the diagram shown in Figure 2 is to take rules in account that state initial values for attributes and association ends. Initial value rules can be stated very comfortably and easily. First of all, we have to point out the class that has got the attribute or association end. This class is termed as the context. Then we describe the expression that states our initial value rule. For instance, initial value of reward points in a RewardAccount will always be zero points. Similarly the initial value of the validity of the card at the time of issue will be – valid card only.

```
context RewardAccount::points
init: 0
context CustomerCard::valid
init: true
```

Another important and crucial part of the model is the specification for establishing the value of derived elements. A model possibly will hold derived attributes and derived associations. For

both derived attributes and derived associations, an invented *derivation rule* can be specified. A derivation rule is a rule that specifies the value of a derived element. Once more, the context of the expression is the class that has got the attribute or association end. The expression comes after the context that states the derivation rule. For example, the derived attribute `printedName` of *CustomerCard* is decided on the basis of the name and title of the card owner.

```
context CustomerCard::printedName
derive:owner.title.concat(' ').concat(owner.name)
```

7. QUERY OPERATIONS

Initial Values & Derivation Rules Operations that do not change the state of the system are termed as query operations. They just return a value or return set of values. It is very difficult to give definition of the result of query operations in UML diagram. Therefore we write or define body expressions in OCL. In the context we provide the operation name, parameters, and return type (its signature). For instance, assuming that the class *RewardProgram* has a query operation `getServices`. `getServices` returns all services proposed and offered by all program associate in this ebusiness:

```
context RewardProgram::getServices(): Set(Service)
body: partners.deliveredServices->asSet()
```

In the above example, the association-end `deliveredServices` of *ProgramAssociate* that has got a set of *Services* is used. For all instances of *ProgramAssociate* that are associated with the *RewardProgram* instance of which the operation `getServices` is called. The result of query operation will be the sets of *Services* that are collected and combined into one set. In the body expression, the parameters of the operation are to be used. For instance, assume that we are looking for an advanced version of the `getServices` operation. This operation takes as a parameter a program associate object and returns the services delivered by the parameter object, if it is an associate in this program. In this case, the refined and advanced operation can be specified as follows:

```
context RewardProgram::getServices(pa: ProgramAssociate):
Set(Service)
body: if partners->includes(pa)
      then pa.deliveredServices
      else Set
    endif
```

The result of this query operation is the set of *Services* held by the parameter `pa`, or an empty set if the parameter instance of *ProgramAssociate* is not an associate of the *RewardProgram* for which the query operation is called. This should be noted that derived attribute and a query operation that has no parameters are the same only difference is of notation. The query operation requires that it is written using braces.

Figure. 2. Class Model for an eBusiness

8. DEFINING ATTRIBUTES & OPERATIONS

This way of defining the attribute results in a derived attribute. The name and type of the attribute, and the derivation rule are included by the expression that defines the attribute. For example, we might want to introduce an attribute called earnings in class *RewardAccount*, which would be the result of the sum up of all the amount attributes of all the transactions on the account as defined in the class *Transaction*. This attribute can be specified and defined by the below given expression. The code fragment after the equal sign states the derivation rule for the new attribute:

```

context RewardAccount
def: turnover : Real = transactions.amount->sum()

```

When the derivation rule is applied, its output will be a single real number that is computed by summing the value of the amount attribute in all transactions associated with the *RewardAccount* instance that holds the newly defined attribute. We can define the operation in OCL; we should note that the operation that is defined in OCL is always will be a query operation. The code fragment that represents the expression just after the equality sign defines the body expression.

Let us consider for an instance, we might wish to introduce the operation `getServicesAsPerLevel` in the class `RewardProgram`. This query operation returns the set of all delivered services for a particular level in a reward point program:

```
context RewardProgram
def: getServicesAsPerLevel(levelName: String): Set(Service)
    =levels->select(name=levelName).availableServices->asSet()
```

The effect and output of the body expression is computed and taken from a collection and selection of the levels associated with the instance of `RewardProgram` for which the operation `getServicesAsPerLevel` is called. It returns only the services available from the *ServiceLevel* whose name is equal to the parameter `levelName`.

9. SUMMARY

This way Unified modeling language is critical to model driven architecture. We have investigated that OCL is critical to and why UML alone is not enough. We investigated how OCL can be incorporated into model in order to forward engineer correct code in ebusiness solution. To demonstrate the use of object constraints along with the modeling we took a case study for an ebusiness enterprise that is based on a reward program. There are many advantages to make UML model crisp and more specified by applying OCL to real-world modeling challenges. Using a combination of UML and object constraints allows developers to realize the effective, consistent, and coherent models. Finally we have all the complete and correct code in hand to satisfy our customer. We have investigated that looking in to constraints applied to the modeling we can improve the quality of the product. It is obvious that improving the quality of modeling definitely improves the quality of analysis and design. It is found during our research that when we apply tracing back, we found lesser bugs in the model (with constraints) as compared to the more number of bugs when we traced back to model from code without using object constraints.

REFERENCES

- [1] Akehurst D. H., Bordbar B.: The Unified Modeling Language, Modeling Languages, Concepts and Tools, 4th International Conference, Toronto, Canada, (2001).
- [2] Balsters H.: Modeling Database Views with Derived Classes in the UML/OCL-framework, In proceedings of <<UML>> 2003 "Modeling Languages and Applications", San Francisco, California, USA, (2003)
- [3] Blaha M., Premerlani W.: Object-Oriented Modeling and Design for Database Applications, Prentice-Hall, (1998)
- [4] Booch G.: Object-Oriented Analysis and Design with Applications, 2nd ed, Benjamin/Cummings, (1994)
- [5] Booch G., Rumbaugh J., Jacobson I.: The Unified Modeling Language User Guide, Addison-Wesley, (1999)
- [6] Carnegie Mellon University/Software Engineering Institute, The Capability Maturity Model: Guidelines for Improving the Software Process, Addison-Wesley, 1995.
- [7] Clark T., Warner J.: Object Modeling with the OCL: The Rationale behind the Object Constraint Language (Lecture Notes in Computer Science), Springer, (2002)
- [8] Coleman D., Arnold P., Bodoff S., Dollin C., Chilchrist H., Hayes F., Jermaes P., Object-Oriented Development: The Fusion Method, Prentice-Hall, (1994)
- [9] Cook S., Daniels J.: Designing Object Systems—Object Oriented Modeling with Syntropy, Prentice-Hall, (1994)
- [10] D'Souza D. F., Wills A. C.: Objects, Components, and Frameworks with UML: The Catalysis Approach, Addison-Wesley, (1999)
- [11] UML/EJB Mapping specification, Java Community Process Document JSR26, (2001)

- [12] Eriksson H., Penker M.: Business Modeling with UML, Business Patterns at Work, John Wiley & Sons, (2000)
- [13] Fowler M.: UML Distilled: Applying the Standard Object Modeling Language, Addison-Wesley, (1997)
- [14] Graham I.: Migrating to Object Technology, Addison-Wesley, (1995)
- [15] Jacobson I., Booch G., Rumbaugh J.: The Unified Software Development Process, Addison-Wesley, (1999).
- [16] Kleppe A., Warmer J., Bast W.: MDA Explained; The Model Driven Architecture: Practice and Promise, Addison-Wesley, (2003)
- [17] Liskov B., Wing J.: "A Behavioral Notion of Subtyping," ACM Transactions on Programming Languages and Systems, Vol. 16, No. 6, pp. 1811–1841,(1994)
- [18] Meyer B.: "On Formalism in Specifications," IEEE Software, January (1985)
- [19] Meyer B.: Object-Oriented Software Construction, Prentice-Hall, (1988)
- [20] Meyer B.: "Design by Contract," in Advances in Object-Oriented Software Engineering, Prentice-Hall, pp. 1–50, (1991)
- [21] Meyer B.: "Applying Design by Contract," IEEE Computer, October (1992)
- [22] Object Constraint Language Specification, version 1.1, OMG document (1997)
- [23] Object Constraint Language Specification, version 2.0, OMG document (2006)
- [24] Object Constraint Language Specification, version 2.3 Beta, OMG document (2011)
- [25] Pinet F., Duboisset M., Soullignac V.: Using UML and OCL to maintain the consistency of spatial data in environmental information systems, Elsevier (2007)
- [26] Richters M.: A Precise Approach to Validating UML Models and OCL Constraints, Logos Verlag Berlin, (2001)
- [27] Rumbaugh J, Blaha M, Premelani W, Eddy F., Lorensen W.: Object-Oriented Modeling and Design, Prentice-Hall, (1991)
- [28] Booch G., Rumbaugh J., Jacobson I.: Unified Modeling Language Reference Manual, Addison-Wesley, (1999)
- [29] Selic B., Gullekson G., Ward P. T.: Real-Time Object-Oriented Modeling, John Wiley & Sons, (1994)
- [30] UML 1.1 Specification, OMG documents, (1997)
- [31] UML 2.0 Specification, OMG documents, (2005)
- [32] UML 2.2 Specification, OMG documents, (2007)
- [33] UML -ISO Released Versions of UML, Ver 1.4.2, (2005)
- [34] Warmer J., Kleppe A.: The Object Constraint Language: Getting Your Models Ready for MDA, Addison-Wesley Professional, 2 edition (2003)