On Types, Instances, and Classes in UML

Guy Genilloud,[1] Alain Wegmann[1]

[1] Institute for computer Communication and Application (ICA)
Swiss Federal Institute of Technology (EPFL)
CH-1015 Lausanne, Switzerland
icawww.epfl.ch
{alain.wegmann, guy.genilloud}@epfl.ch

1 Introduction

The semantics of UML [3] is for a large part given in English. The drawback of English is its ambiguity, which makes it unclear and ambiguous. But the benefits of English are its power and flexibility, which allow it to deal with a slightly inconsistent terminology. Formalizing UML will imply recognizing the occurrences of these inconsistencies and coping with them in a systematic way. In this paper, we show some problems with the UML terminology and we propose an approach to deal with these problems.

2 Problems with the UML Terminology

In UML the terms *instance*, *type*, *class* take their meanings according to the classical object-oriented programming terminology. This leads to the following problems:

1. The UML glossary makes *type "a stereotype of class that is used to specify a domain of instances (objects)…*". As a consequence, it is impossible to speak of types of use cases, of types of messages, of types of actors, etc. By speaking of *"type-like*" elements, UML recognizes the existence of a more general concept of type than what is defined in its glossary. However, there is no definition of "type-like" in UML.

2. UML speaks of a type-instance dichotomy, saying that "*Many or most of the modeling concepts in UML have this dual character, usually modeled by two paired modeling elements, one represents the generic descriptor and the other the individual items that it describes. Examples of such pairs in UML include: Class-Object, Association-Link, Parameter-Value, Operation-Call, and so on. To every instance concept, there is a related type concept, and vice-versa"* [3, section 3-12]. We believe that it is not "many or most;" it is all instance concepts which (potentially) have a type. Or conversely, there can be no type concept if there is no instance concept. UML makes exceptions to this rule: "actor," "interface," and "role" are defined as specifications and have no equivalent instance concepts. We believe each of these omissions to be erroneous. For example, we have shown in [1] that thinking of role instances is essential to understand this modeling concept. And in [4], we

have shown that modeling at the level of actor instances is very useful for specifying execution and multiplicity constraints.

3. The UML glossary is not systematic in its approach to defining concepts, and does not always use terms according to their definition. For example, UML defines *"message"* as a specification concept but also speaks of "*sending a message*." But "sending a specification" is not what is meant. The fact that UML does not always use "stimulus" (the defined term for the occurrence of a message) in these cases shows how impractical it is to choose different terms for the instance and the type. As another example, component, unlike similar concepts, is defined as an instance concept. And UML has no term or definition for the related specification concept. These lacks of systematic in defining concepts make the UML terminology excessively complex to understand and to use.

4. Class is defined as *"A description of a set of objects that share the same attributes, operations, methods, relationships, and semantics."* This definition is ambiguous about whether a class is a set of objects (the English meaning of a class) or is a description of the common features of these objects (the OO meaning of a class). UML tends to use the term class with both meanings, with the consequence that the semantics of a class diagram are unclear.

3 The ODP-based Approach for the Terminology

The RM-ODP Foundations has a very systematic approach to terminology0207 Tc 0.2082 Ts2e Terminology

The1R0 3 0 TD 0.1395 Tc 0.798 Tw (ODP Foundatio4about whe2stance a

Class (of <X>s): *The set of all <X>s satisfying a type … [2, Def. 9.8]*

3.2 Templates and Instantiations

In making models, we rarely specify instances directly. Rather we define templates and we instantiate them. Template and instantiation are generically defined as:

<X> Template:

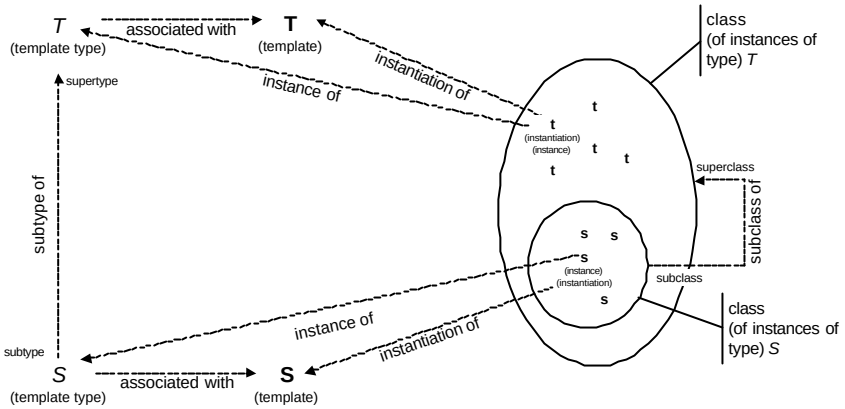

Figure 1: Relationship between templates, instantiations and instances.

3.4 Inheritance vs. Subtyping

The RM-ODP Foundations make it clear that an inheritance relationship (implied by the incremental modification of a template) is not necessarily a subtyping relationship. The details are given in the definition below.

> ***Derived class/base class****: If a template A is an incremental modification of a template B, then the template class CA of instances of A is a derived class of the template class CB of instances of B, and the CB is a base class of CA.…*
>
> *Classes can be arranged in an inheritance hierarchy according to derived class/base class relationships. This is the interpretation of* ***inheritance*** *in the ODP Reference Model. If classes can have several base classes, inheritance is said to be* ***multiple****. If the criteria prohibit suppression of properties from the base class, inheritance is said to be* ***strict****.*
>
> *It is possible for one class to be a subclass of a second class without being a derived class, and to be a derived class without being a subclass. The inheritance hierarchy (where arcs denote the derived class relation) and the type hierarchy (where arcs denote the subtype or subclass relation) are therefore logically distinct, though they may coincide in whole or in part. [2, Def. 9.21]*

4 Discussion

4.1 Refactoring the UML Terminology

We think that refactoring the UML terminology is an essential step to make it more consistent and simpler to understand. If the semantics of UML is to be given in UML, then making UML more consistent is likely to be useful.

The main problem with the UML terminology is the lack of systematic in defining the terms. The duality type-instance is not dealt with for all terms, and the base terms are arbitrarily defined as templates (or specifications) or as instances. Using the generic definitions of the RM-ODP, that we presented above, can be a good basis for refactoring the UML terminology, or more modestly, to analyze the existing terminology and the meta-model.

4.2 The UML Type System

As a foundation for object-based modeling or programming languages, the RM-ODP has an extremely generic type system. It talks about the relationships between templates and types, between supertypes and subtypes, and between base classes and derived classes. Which templates are typed, and what their types are, is left for the specific language to define. The only constraint imposed by the RM-ODP is to think of types in terms of predicates, which in a sense excludes languages whose type systems are inconsistent.

UML can be used for many different modeling purposes, from enterprise modeling to code generation into a specific target language. On that perspective, UML is similar to the RM-ODP. Yet UML seems to have a very different approach, much less generic, to defining its typing system. Is the UML approach adequate for its purpose, or should it be made more general in the ODP way?

The UML approach to typing is in fact not clear at all. How can it be explained on the basis of the RM-ODP Foundations? And what can we learn from this?

5 References

1. Genilloud Guy, Wegmann Alain, “A Foundation for the Concept of Role in Object Modelling,” EPFL Technical Report (in preparation)

2. ISO/IEC ITU-T, “Open Distributed Processing – Basic Reference Model – Part 2: Foundations,” Standard 10746-2, Recommendation X.902. 1995. (http://isotc.iso.ch/livelink/livelink/fetch/2000/2489/Ittf_Home/PubliclyAvailableStandards.htm).

3. OMG, *Unified Modeling Language Specification*, Version 1.3, June 1999, www.omg.org

4. Wegmann Alain, Genilloud Guy, “The Role of “Roles” in Use Case Diagrams,” EPFL Technical Report (in preparation).