DOI: 10.2298/CSIS110716006M

Towards Understanding of Classes versus Data Types in Conceptual Modeling and UML

Dragan Milićev

University of Belgrade
Faculty of Electrical Engineering, Department of Computing
P.O. Box 35-54, 11120 Belgrade, Serbia
dmilicev@etf.rs

Abstract. Traditional conceptual modeling and UML take different vague, ambiguous, and apparently incompatible approaches to making a distinction between two different entity types – classes and data types. In this paper, an in-depth theoretical study of these ambiguities and discrepancies is given and a new semantic interpretation is proposed for consolidation. The interpretation is founded on the premise that populations of the two kinds of entity types are defined in two substantially different ways: by intensional (for data types) and extensional (for classes) definitions. The notion of a generative relationship set is introduced to explain the role of specific relationship types that are used to define populations of structured data types by cross-combinations of populations of the related entity types. Finally, some important semantic consequences are described through the proposed interpretation: value-based vs. object-based semantics, associations vs. attributes, and identity vs. identification. The given interpretation is based on runtime semantics and allows for fully unambiguous discrimination of the related concepts, yet it fits into intuitive understanding and common practical usage of these concepts.

Keywords: conceptual modeling, Unified Modeling Language (UML), formal semantics, class, data type, entity, relationship, object identity, identification, association, attribute.

1. Introduction

Ever since their introduction with Entity-Relationship (ER) modeling [4], the notion of *entity type*, along with its more recent and object-oriented successor concept of *class*, have had a central role in conceptual modeling and programming in general. In addition, classical conceptual modeling recognizes the concept of *data type* as a special kind of entity type, and fairly clearly and seemingly unambiguously describes the distinction between data types and the other entity types [17, 6]. Unfortunately, as it will be shown in

this paper, this distinction does not seem to be precise enough to allow clear separation and proper use of the two kinds of entity types. The lack of unambiguous criteria for discriminating between data types and the other entity types inevitably causes uncertainty to modelers about which of the two concepts should be used in each particular case, or about what the original semantic intent was behind a particular use of one or the other in a model.

On the other hand, the Unified Modeling Language (UML) has adopted the analogous separation of *classes* and *data types* in a yet more explicit way, clearly emphasizing a few distinctive characteristics of classes versus data types [16]. However, as it will be discussed in this paper, the definitions of the distinction between classes and data types appear to be completely different from those widely used in classical conceptual modeling. Such a situation unnecessarily increases the confusion in understanding of these notions and their use in practice. This is yet another of many existing parts of the UML's definition that contributes to its main drawback – the lack of a complete semantic formalization. Many concepts of UML, especially in its early versions, have not had definitions precise enough to be interpreted unambiguously. In other words, many parts of a syntactically correct UML model can still be interpreted in different ways by different readers. This has been recognized as the main obstacle for UML to become a machine-interpretable, i.e., executable language. Instead, UML served predominantly for recording ideas, sketches, and design decisions in early phases of software analysis and design, without an ambition of encouraging unambiguously interpretable models.

Recent model-driven development (MDD) trends in software engineering [22, 23, 14] have dramatically increased the importance of formalizing the semantics of UML, as the key prerequisite for its switch from a solely descriptive to an unambiguously interpretable and executable language. As a result, a major revision of UML has emerged in its version 2 [16], making a significant step towards precise semantics of concepts. Much effort has been made to clarify the meaning of the widely adopted concepts in the very UML 2 specification, as well as in other attempts before and around it [1, 3, 5, 7, 8, 9, 14, 15, 17, 18, 19, 24]. However, the distinction between classes and data types has not yet been fully clarified and consolidated with the conception in classical conceptual modeling.

In this paper, we describe apparent inconsistencies in and incompatibilities between the definitions given in classical conceptual modeling and in UML, address these issues, and propose consolidation, attempting to get closer to a fully clear understanding of these classical and widely used notions. We also show how the understanding of these notions affects the understanding of associations versus attributes in UML. We believe that the improvements in the common understanding of these notions can lead to better communication between software designers and developers, proper and certain use of these concepts in conceptual modeling, as well as their coherent implementation in environments for building and running executable models.

The rest of the paper is organized as follows. The next section gives basic definitions and assumptions in the field as the foundations for the discussions that follow. Section 3 brings an overview of the treatment of the concepts relevant for the subject matter in classical conceptual modeling and in UML, and reveals some ambiguities, discrepancies, or seeming contradictions. Section 4 gives an informal analysis of the subject matter that leads to the proposed solution in an intuitive way. Section 5 formalizes the proposed interpretations through precise definitions. Section 6 summarizes some semantic and practical consequences of the presented interpretations. The paper ends with conclusions.

2. Basic Concepts and Assumptions

2.1. Entity Types

To avoid terminological confusion and ambiguity, we will use the terms *class* and *data type* adopted from UML to refer to the two different kinds of classifiers of relevance in this discussion, and the term *entity type* from classical conceptual modeling and ER [4, 17] to refer to both (or any). In other words, entity type will be used to refer to a generalization of the concepts of class and data type, while the division of entity types into classes and data types is assumed to be a partition, i.e., a covering and disjoint specialization.[1]

As a starting definition, we can say that an entity type is a concept whose instances at a given time are individual entities that exist in the domain at that time. This is a reasonably precise and, with some variations, widely adopted definition. For example, this is a variation of the definition from classical conceptual modeling [17], with one intentional modification: the definition in [17] requires that instances of every entity type be identifiable, while we will revisit this aspect in this paper. Put another way, entity types represent sets of individual entities. These sets are generally variable in time. Variability of entity types over time is one of the fundamental assumptions in conceptual modeling. Entities can begin or cease to exist, or they can be reclassified to other entity types [17, 6, 14]. A more formal definition of entity types as variable sets can be found in [6].

[1] We will not use the UML term *classifier* for such a generalizing concept, because there are many other kinds of classifiers in UML, such as collaborations or use cases, which are out of scope of this discussion.

In order to make this definition slightly more practical and allow some more precise semantic interpretations that will be given in the paper, we will refine this definition and say that an entity type represents a set of entities that exist in the *system* or in the *domain* in *runtime*. By referring to runtime, we imply an inherent dichotomy between *design time* (or modeling time) and *runtime* that exists in practice. The exact practical meaning of these two terms certainly depends on a concrete framework or tooling in case, but traditionally, they refer to the activities of data schema design or modeling (for design time), and data manipulation or model execution (for runtime). It should be noted that the distinction between the two is mostly ontological and does not necessarily imply different physical times; in reality, these two may refer to the same physical time, meaning that the two activities (of system design and its execution) may take place simultaneously. For example, many database management systems and other frameworks allow modifications of the database schema or model during the exploitation of the system; others may need to interrupt the exploitation of the system in order to recompile and redeploy the schema/model, and to restart the execution environment, but without affecting the existing entities that survived the interruption. Anyhow, we consider the runtime as an absolute temporal scope of all entities that may exist in a certain system, and outside which the entities cannot exist due to the very nature of the considered system or the technology used. That might be, for example, the lifetime of a certain installation of a database system, or an execution of a program in the classical sense.

A *constant entity type* is an entity type whose set of instances is constant, i.e., invariable (immutable) in runtime [17, 6].

2.2. Relationship Types

For similar reasons as for entity types, before we discuss and clarify the semantics of attributes and associations in UML, we will use a generic term *relationship type* taken from classical conceptual modeling and ER. A relationship type is a concept whose instances at a given time are individual relationships between entities that are considered to exist in the domain at that time [17]. Again, we have omitted the requirement in the definition given in [17] that individual instances of a relationship type be identifiable in order to allow interpretations like the one given in [14, 15] and to align it with the definition given in UML 2 [16], where instances of relationship types (i.e., associations in UML) cannot be identified. We do not see any significant semantic or practical impact of this modification.

A relationship type of degree $n \geq 2$ consists of an ordered set of n participants, whereby a participant is an entity type that plays a role in the relationship type [17]. We will write $R(p_1{:}E_1, ..., p_n{:}E_n)$ to denote a relationship type named R, with participant entity types $E_1, ..., E_n$, playing roles $p_1, ..., p_n$ respectively. Note that $E_1, ..., E_n$ do not have to be different entity types, because the same entity type can play several roles in the same relationship

type. Roles must be, naturally, pairwise different. We may omit the role p_i played by participant p_i:E_i either because it is obvious or it is the same as the name of E_i; then it is assumed that p_i is the same as E_i. For example, *Reads*(*reader*:*Person*, *Book*) is the same as *Reads*(*reader*:*Person*, *book*:*Book*).

A relationship of type R has a form of a set {<p_1:e_1>, ..., <p_n:e_n>}, sometimes also referred to as a *tuple*, where e_1, ..., e_n are instances of their corresponding entity types E_1, ..., E_n. In classical conceptual modeling, all relationships of a certain relationship type that exist at a certain moment in runtime form a set of distinct tuples [17]. A more recent interpretation [14, 15] defines relationship types as concepts whose instances at a certain moment in runtime form a bag of tuples (i.e., a multiset), in order to support advanced notions of uniqueness of roles in UML 2. The discussions in this paper are independent of the interpretation used.

A binary relationship type R(p_1:E_1, p_2:E_2) defines two inverse mappings [5, 6, 14, 15]:

- p_1 that maps an instance e_2 of E_2 to a set (or bag in [14, 15]) of all those and only those instances e_1 of E_1 for which {<p_1:e_1>, <p_2:e_2>} is in R at any particular moment in runtime;
- p_2 that maps an instance e_1 of E_1 to a set (or bag) of all those and only those instances e_2 of E_2 for which {<p_1:e_1>, <p_2:e_2>} is in R at any particular moment in runtime.

It is easy to see that e_1 is in $p_1(e_2)$ if and only if e_2 is in $p_2(e_1)$. The dynamicity of the sets of instances of entity and relationship types during runtime implies that these mappings are generally also variable in time [5, 6].

A relationship type R(p_1:E_1, ..., p_n:E_n) is *constant* with respect to a particular participant p_i if the instances of R in which an instance e_i of E_i participates are the same during the temporal interval in which e_i exists, for each e_i of E_i. A relationship type is constant if it is constant with respect to all its participants [17]. Obviously, when a binary relationship R(p_1:E_1, p_2:E_2) is constant with respect to e.g. p_1, it means that the mapping $p_2(e_1)$ is constant during the lifetime of e_1 for each e_1 of E_1.

2.3. Populations and Actions

We will refer to the set of instances of an entity type E that exist at a certain moment in runtime as the *population* of E.[2] Similarly, the set or bag of instances of a certain relationship type R that exist at a moment in runtime will be referred to as the *population* of the relationship type.

[2] Sometimes also called the *extent* of E. However, we will use the term *extent* for a slightly different concept.

Populations of entity and relationship types are assumed to be dynamically changed during runtime by means of *actions*. Actions are atomic units of behavior that affect populations of entity and relationship types. In this context, we are focused on a generic set of elementary actions on entity and relationship types, without going deeper into their formal semantics as they are not directly relevant to the conclusions of this paper:

- *Create a new instance of one or more given entity types*, which adds a new entity *e* to the system and to the populations of the given entity types. (In general, an entity can be an instance of more related or unrelated entity types.)
- *Delete an existing entity e*, which removes *e* from the populations of its entity types and from the entire system. Deletion of an entity implicitly removes all relationships in which that entity participates.
- *Reclassify an existing entity e*, by removing the entity from the population of zero or more given entity types and adding it to the population of zero or more entity types. Removing an entity from the population of an entity type implicitly removes all relationships in which that entity participates with a role of that entity type.
- *Create a new instance of the given relationship type R*, which adds a new relationship $\{<p_1:e_1>, ..., < p_n:e_n>\}$ to the population of *R*.
- *Delete an existing relationship*, which removes a given relationship $\{<p_1:e_1>, ..., < p_n:e_n>\}$ from the population of its relationship type.

While the system's overall state is defined in terms of populations of its entity and relationship types, the system's behavior is defined in terms of executed actions. More detailed and formal treatments of actions and their semantics can be found in [6, 14].

Constant entity and relationship types have immutable populations. This means that actions on constant entity or relationship types that would change their populations are illegal.

2.4. Identification

An *identifier* of an entity *e* is an expression, written in some language, that unambiguously denotes *e* [17].[3] In this paper, most relevant are identifiers built from binary relationship types that have particular properties, referred to as *reference relationship types*, as defined in [17]. A more general approach to building identifiers through so-called *observation terms* is presented in [10].

[3] In general, identifiers may have temporal or other kinds of scopes, meaning that the same identifier (being an expression) may denote different entities at different times or in different scopes. We will not consider this case, as it is orthogonal to the discussions in this paper.

We will here provide a semi-formal definition of observation terms, which is sufficient for the discussions given in this paper; for a fully formal definition, the reader is referred to [10].

Definition (Observation term). *Observation term* of E (denoted with $\tau(E)$) is an injective function τ: $\pi(E) \rightarrow S_\tau$, where $\pi(E)$ is the population of an entity type E at any moment in runtime, and S_τ is a certain fixed set:

$$(\forall e_1, e_2 \in \pi(E))(\tau(e_1)=\tau(e_2) \Rightarrow e_1=e_2).^4$$

Even less formally, $\tau(E)$ is a function that maps an entity $e \in \pi(E)$ to an atomic or structured value (formally bound by a certain set S_τ) so that $\tau(e)$ uniquely identifies e. Observation terms are more general than simpler identifiers normally used in practice, such as reference relationship types, because $\tau(E)$ may map an e to an arbitrarily deeply structured value whose components may be obtained by more complex mappings than simple relationships, like queries. Of course, reference relationship types (both simple and compound) are special cases of observation terms. In [10], Gogolla gives a detailed definition of how $\tau(E)$ can be constructed, i.e., how S_τ can be defined.

By its definition, $\tau(e)$ represents an identifier of e, meaning that it implies an inverse function f_τ: $S_\tau \rightarrow \pi(E)$ that maps a value from S_τ to an entity e, for a subset of S_τ for which it is defined. It is important to note that $\tau(E)$ is an injection that does not have to be a surjection, meaning that there does not have to be an $e \in \pi(E)$ for each element s of S_τ so that $\tau(e)=s$. This is trivially true because the set $\pi(E)$ can be dynamically changed in runtime, while S_τ is fixed. Therefore, in general, f_τ is a partial function, defined for a subset of S_τ. The function f_τ is the *identification function* that can be used for identifying entities in $\pi(E)$.

For a comprehensive study of identification and formalization of conceptual modeling in general, the reader is referred to [17, 10].

[4] The formal semantics of equality is a very subtle and difficult topic; for example, see a very interesting perspective and conclusions described in [13]. This is why we do not go deeper into that matter. We may simply note that the operation "=" may mean equality (up to isomorphism) or identity in different contexts. It seems that our observations do not require a fully formal clarification of this issue and that the interpretation of $e_1=e_2$, where e_1 and e_2 are (references to) entities, is that e_1 and e_2 represent the same entity (identity).

3. Overview of the Matter

3.1. Classes vs. Data Types in Classical Conceptual Modeling

Classical conceptual modeling defines *data type* as a constant entity type whose instances, called *values*, can be identified by literals, whereby literals are strings (i.e., sequences) of some symbols (e.g., characters) [17, 6]. The same value of a certain data type can often be identified by several different literals; for example, both 1.0 and 0.1E+1 denote the same value one of a floating point data type. Although this definition sounds intuitively clear and reasonably practical, it is still formally vague and conceals several important issues.

First, isn't it possible that an instance of a class (i.e., of an entity type that is not a data type) be identifiable by a literal? For example, one can imagine a system where each instance of a constant class is assigned (e.g., at design time) a literal that identifies it, so identifiability through literals is not a sound discriminator between the two notions. (It will be shown later that this is exactly the case with enumerations.) On the other hand, is it really necessary that each instance of every possible data type (e.g., a structured one) be identifiable by a literal? What is the real difference between data types and constant classes then?

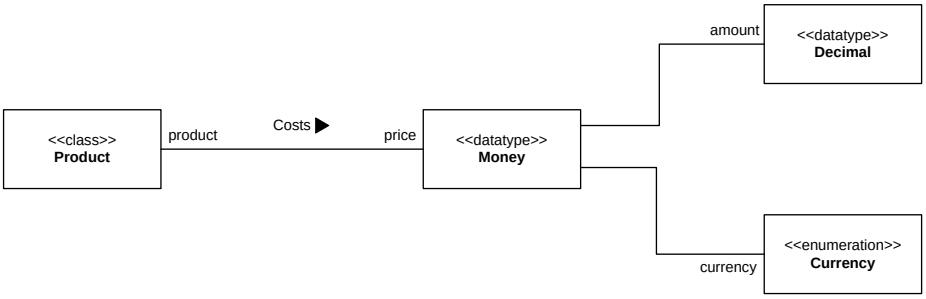


Figure 1. A constant entity type *Money* with a non-constant relationship type with *Product* and constant relationship types with *Decimal* and *Currency*

Just being a constant entity type is not a sufficient discriminator for data types, simply because classes can also be constant. Data types often imply constancy of some (but not all) relationship types. Figure 1 gives an example. The data type *Money* is a structured entity type that represents a certain amount of money expressed in a given currency; these two properties of *Money* are modeled with the two respective relationship types with *Decimal*

and *Currency* data types.[5] Obviously, these two relationship types are constant with respect to *Money*. One should not change the pair of instances of *Decimal* and *Currency* related to a certain instance of *Money*, because it would change the very identity of that instance of *Money*. Indeed, if it were the opposite, one could relate two instances of *Money* with the same pairs of *Decimal* and *Currency*, thus making two undistinguishable and fundamentally the same instances of *Money*. In fact, these two relationship types form a compound reference [17] of *Money*, so this situation should not occur. On the other hand, however, there are usually non-constant relationship types in which a data type participates. The relationship type *Costs* is obviously non-constant with respect to *Money*. However, to the best of the author's knowledge, there is not a clear explanation and understanding of these aspects in the open literature, especially of which relationship types of a data type are or are not constant (with respect to that data type).

Beeri [2] gives one of the most profound analyses on the difference between classes (i.e., objects as their instances) and data types (i.e., values as their instances) in the context of a formal foundation of object-oriented databases. However, his discussion is mostly restricted to primitive data types. Using the example of numbers, Beeri gives a few criteria for differentiating data types from classes (referred to as abstractions in his discussions); we first quote each criterion and then give our comments on it:

"1. Numbers are a universally known abstraction, and have the same meaning to all people. Abstractions such as employees are application specific."

This is correct for the case of most general and application-independent data types. However, applications very often introduce quite specific, derived data types, typically structured ones.

"2. Numbers have names in the name space. In contrast, application specific abstractions normally do not have names."

We agree that objects of application-specific abstractions (classes) normally do not have names as their identifiers, but as discussed above, there is not any practical or formal reason against having names, as also clearly indicated in the same Beeri's paper. There are many examples of such cases, and this paper will later argue that enumerations are actually constant classes with named objects.

[5] One can notice that *Money* is actually a reified ternary relationship type between *Product*, *Decimal*, and *Currency*. It might be regarded so, but the reification (i.e., the promotion of the ternary relationship type into the entity type with the three binary relationship types) has been done for the purpose of reusability of the data type *Money* in other places in a broader model, such as for properties of other entity types.

"3. Numbers (and their names) are built into the system; they need not be defined. Object abstractions need to be introduced to the system by definitions."
This is actually a distinction between built-in and user-defined (application-specific) entity types, primarily determined by the construction of a particular language or model, but not a distinction between data types and classes. We believe that this distinction is orthogonal to the distinction between classes and data types, as both classes and data types can be built-in as well as user-defined. For example, a structured derived data type such as *Money* in Figure 1 can be either built-in or user-defined, depending on the concrete implementation of the language or framework.

"4. The information carried by a number is in itself. ... In contrast, the interesting information about application specific abstractions is carried in relationships they have with other objects and values."
We argue that this is a correct distinction between primitive data types on one hand and both classes and structured data types on the other, but not between classes and data types. For example, a structured derived data type such as *Money* in Figure 1 is an example of a data type whose interesting information is truly in its relationships with *Decimal* and *Currency* and not in itself.

Beeri also concludes that "none of these distinctions is absolute" and that "both objects and values are objects in the general intuitive sense." In this paper, we will stay aligned with this conclusion, but will try to provide a more precise and unambiguous distinction between classes and data types.

3.2. Classes vs. Data Types in UML

The confusion becomes even worse when the definitions of data types in UML get involved [16]. In UML, instances of classes are entities (called *objects*) with inherent identity, while instances of data types are "pure values" that have no identity: "A data type is a special kind of classifier, similar to a class. It differs from a class in that instances of a data type are identified only by their value. All copies of an instance of a data type and any instances of that data type with the same value are considered to be the same instance. Instances of a data type that have attributes (i.e., is a structured data type) are considered to be the same if the structure is the same and the values of the corresponding attributes are the same. If a data type has attributes, then instances of that data type will contain attribute values matching the attributes." [16, p. 60]

This "definition" is extremely vague. First, the construction that "all copies of an instance of a data type and any instances of that data type with the same value are considered to be the same instance" sounds logically problematic indeed. Second, it is unclear what "values" really are. Then, in case of structured data types, the definition partially relies on the notion of attributes, which is even more unclear, as it will be discussed soon. Finally, even if the notion of attribute is taken to be just a relationship type as in [17],

it is unclear which relationship types have to relate the given two instances of a data type to the same sets of other instances in order to be treated as "having the same value." For example, for the data type *Money* in Figure 1, if two instances of *Money* have relationships with the same instances of *Decimal* and *Currency*, respectively, they obviously have to be interpreted as having the same "value". However, these two same instances of *Money* obviously may be related to different instances of *Product*, stating simply that these two products have the same price, but the two instances of *Money* are still having the same value, although they do not have the same relationships (structure). This observation indicates a certain hidden difference between the meaning and purpose of different relationship types relating the data type *Money*.

An elaborate discussion and clarification of the meaning of these statements and their practical implications can be found in [14]. In brief, every object of a class has its own identity, which is an inherent characteristic of that object that distinguishes it from any other object of the same or any other class. The identity of an object is inherent, meaning that it is ensured by the very existence of the object, and does not require any special activity by the modeler or user in design or runtime. For example, one instance of class *Person* is, by default, different and can be distinguished from any other instance of the same class or any other class. How it can be distinguished is a matter of a concrete technique, notation, and implementation. It is only important to emphasize that two objects are distinguishable simply because they are two separate identities by their nature and existence (that is, because they have been created by two separate executions of the Create Entity action). The identity of an object is its inherent characteristic, and is independent of any relationships (e.g., properties of the object or any other part of its state). In other words, two objects can have exactly the same relationships and exactly the same state, but they are still two different identities. For example, two instances of the class *Person* may have the same name, home address, date of birth, and all other properties, but they are still treated as two different and independent entities. In addition, the identity of an object is not affected by any modification of its properties.

Unlike class instances, instances of data types do not have their identity in UML's interpretation — that is, they are pure values. Two instances of the same data type with all properties having equal values cannot be distinguished. Being pure values without identity, instances of data types in UML are immutable, meaning that operations of data types do not have side effects and do not change values of their owner instances, but are pure functions that can only produce new instances and not change the existing ones.

To conclude, the two approaches taken in classical conceptual modeling and in UML obviously differ significantly. (The only apparent commonality is that operations of data types cannot modify their owner instances.) Both rely on the notion of value, but what is really "a value"? In UML, one can create many "equal" instances of a data type by Create Entity actions, while data

types are constant in classical conceptual modeling and such actions should be treated as illegal.

3.3. Associations vs. Attributes

The confusion is similar with distinguishing associations and attributes as different subkinds of relationship types.

Olivé [17] clearly points out that attributes are very similar to binary relationship types and are not really needed at the conceptual level. In fact, an attribute is a binary relationship type $R(p_1{:}E_1, p_2{:}E_2)$ in which participant p_2 is considered to be a *characteristic* (or *property*) of E_1 (with E_2 being the type of the attribute), or p_1 a characteristic of E_2 (with E_1 being the type of the attribute). Therefore, an attribute is like a binary relationship, except that users and designers add the interpretation that one participant is a characteristic of the other. In associations, on the other hand, the order of the participants does not imply any priority or subordination between them.

Obviously, except from the rather subjective interpretation by the humans, there is not any formal distinction between attributes and associations. "Thus it is not clear whether attributes should be used or when," concludes Olivé [17, p. 76].

Quite similarly, standard UML [16] does not impose any significant semantic difference between binary associations and attributes either. It simply provides them as different kinds of modeling elements, without giving any clear and formal difference in their runtime semantics. For example, one can use either or even both at the same time to model a specific binary relationship type between certain entity types. In other words, mostly similarly to classical conceptual modeling, standard UML treats attributes and associations just as two syntactic variations of the same background concept of a binary relationship type. Obviously, having such a situation does not help modelers and model readers too much, because it increases confusion in which modeling concept to apply or how to interpret the intention of the model designer when reading a model.

Instead of sound criteria on the use of attributes, some guidelines exist only. One guideline, well known in conceptual modeling and also suggested by standard UML [21], is that types of attributes should be entity types defined outside the scope of the considered model, while associations should be used to relate entity types that are defined within the considered model. This is, however, not a strict guideline; it just helps modelers make models easier to understand.

A formal and executable profile of UML named OOIS UML and described in [14] gives more precise and strict discriminating characteristics between the two notions: associations are relationships that may involve only classes as their participants, while attributes are binary relationship types whose one participant is always a data type and represents the type of the attribute, while the other can be either a class or a data type and represents the owner of the attribute. In other words, the type of an attribute may only be a data

type. The profile also describes some additional semantic implications, briefly summarized as follows:

- Objects of classes are compared for equality by reference, while instances of data types are compared for equality by value.
- As they can have only attributes, instances of data types have implicit copy semantics that deeply (recursively) copies values of their attributes. Objects of classes do not have implicit copy semantics.
- Objects of classes have an explicit lifetime – they are explicitly created and destroyed by executing actions. Instances of data types have an implicit lifetime – they are created by executing actions, but are destroyed implicitly, when they are not referred to any more.
- Operations of classes may modify the objects' state. Operations of data types must not modify values of their attributes; they must be pure functions that possibly produce new values.

The semantics given in [14] is strict, clear, and unambiguous. The author has applied the profile and the rules in many industrial projects and has found the interpretation practically useful. However, it is not obvious whether the described two approaches are compatible and how they relate to each other. Although there are practical and intuitive rationales behind the definition given in [14], its formal background was still unclear.

All the described approaches agree that data types can have attributes too. Such data types are called *structured* in UML. Attributes of data types are considered to be immutable characteristics of their owner instances.

3.4. Summary

The presented discussions are summarized in Table 1 that gives an overview of the criteria used to discriminate classes vs. data types in classical conceptual modeling (including Beeri's guidelines), standard UML, and OOIS UML.

It is useful to mention briefly that a similar confusion exists in programming in classical object-oriented programming languages, such as C++ or Java. There, objects of classes (as the only kinds of supported entity types) have their inherent identity that is based on their very existence in time and space – the computer memory. Unrelated to their state, objects can be distinguished by their technical identifiers – references (pointers) that conceptualize objects' physical location in computer memory. However, programmers usually feel a need for what is clearly recognized in conceptual modeling and UML as a data type, so they introduce "special kinds of classes" that they usually call "immutable," rely on their "value-based equality comparison, copying, argument passing, etc.," and must deal with implicit lifetimes of data type instances or explicit lifetimes of objects (although the language sometimes supports only one of them).

As a conclusion, these two approaches (in classical conceptual modeling and in UML) to distinguishing between classes and data types on one hand,

as well as between associations and attributes on the other, may seem unrelated or even contradictory, or at least vague and confusing at first sight. We will try to consolidate them and form a unified and consistent understanding of the notions. In fact, we will show that both approaches are actually correct and compatible, although not complete and fully formal, and will provide complete formalizations of all related notions.

Table 1. Summary of the discriminators of classes vs. data types used in classical conceptual modeling and UML

	Class	*Data Type*	*Issue*
Conceptual modeling	Variable?	Constant	Why cannot a class be constant as well?
	Not identifiable by a literal?	Identifiable by a literal	Why cannot a class be identifiable by a literal?
	Domain/application-specific	Universally known abstraction	A user-defined, domain-specific, structured data type is not a universally known abstraction.
	User-defined	Built-in	Both classes and data types can be built-in and user-defined. This is an orthogonal distinction.
	Interesting information carried by relationships	Interesting information carried by the value	Not true for structured data types.
Standard UML	Instances have identity	Instances do not have identity, but are pure values	What do “values” exactly mean?
	Equality by identity	Equality by structure/value	Definition for structured data types based on the unclear notion of attribute. The isomorphism of relationships/attributes unclear.
	Instances are generally mutable	Instances are immutable	No issues.
OOIS UML	Instances have identity	Instances do not have identity, but are values	What do “values” exactly mean?
	May take part in associations	May have attributes only	No issues.
	May not be types of attributes	May be types of attributes	No issues.
	Equality by identity (reference)	Equality by attribute values (recursively)	No issues.
	No implicit copy semantics	Implicit copy semantics	No issues.
	Explicit lifetime of instances	Implicit lifetime of instances	No issues.
	Instances are generally mutable	Instances are immutable	No issues.

4. Analysis

Before reaching precise definitions, we will carefully analyze the practical use and intents behind of what are traditionally referred to as classes and data types (including primitive and derived data types, structured data types, and enumerations).

4.1. Populations of Classes and Data Types

We deem that the central difference between different subkinds of entity types is the way how the population of an entity type is defined.

For entity types that we traditionally treat as data types, the population is typically defined *intensionally*. That is, the set of instances of a data type is defined by a formula or a set of rules or constraints that are necessary and sufficient conditions for belonging to the set. For example, the population of a primitive type *Integer* can be defined as the set of integer numbers that can be represented with a certain binary format, e.g., a 32-bit two's complement, which defines the set of integers in the range from -2^{31} to $+2^{31}-1$. Similarly, the population of the data type *String* is the set of all logically unlimited arrays of characters, including the empty array.

On the other hand, the population of a typical class is defined *extensionally*: for each particular instance of a class in its population, it is directly and explicitly stated that the instance belongs to the population, basically by executing a Create Entity or Reclassify Entity action; the instance belongs to the population of the class until it is destroyed or reclassified by another action.

4.2. Structured Data Types

The given simple example of data type *Integer* addresses only a subset of simple data types that are traditionally called *primitive* data types. These are types that are not defined in terms of other data types; they exist ab initio. *Derived* data types are those that are defined in terms of other data types. The way how (populations of) derived data types are defined predominantly depends on the language constructs and features, but we can list some of the most general approaches:

1) Defining a subset of another data type or of a union of other data types. The subset can be defined intensionally, by a rule, or extensionally, by enumerating instances that form the subset.

2) Through a set and functional calculus, by defining a data type whose instances are sets or functions of other data types, or with algebraic specifications of abstract data types [11, 12]. Such types include different kinds of collections, sets, bags, maps, and other non-atomic types used in different languages.

3) Defining structured data types by composition of other entity types.

In this paper, we will focus on and confine to the semantics of the third subcategory – structured data types only, as they are most typical derived data types used in conceptual modeling. There, a derived data type is defined in terms of relationship types with other entity types. The data type *Money* in Figure 1 represents one such example. Another simple and intuitive example is a data type *Point* with two relationship types (*Point*, *x*:*Decimal*) and (*Point*, *y*:*Decimal*) that define the coordinates of a point in a two-dimensional space.

Let us carefully consider the semantics of structured data types, actually, the intent of the modeler who defined them and the meaning she obviously had in mind. The populations of such entity types are defined intensionally, in terms of a set of some other entity types and relationships with them. For the two given examples, this looks like the following:

- For each pair of existing instances of *Decimal* and *Currency*, there exists exactly one (one and only one) instance of *Money* that is related to the two instances of *Decimal* and *Currency*.
- For each pair of two existing instances (possibly the same) of *Decimal*, there exists exactly one (one and only one) instance of *Point* that is related to the two instances of *Decimal* that play the *x* and *y* roles.

Optionally, in a more general case, such a definition may include a constraint that reduces the set of instances obtained by a simple Cartesian product of the related populations.

Consequently, unlike other relationship types in which a certain data type participates, such relationship types have a special meaning: they are used to define the population of the entity type by the (optionally reduced) Cartesian product of populations of the related entity types. Due to their special role in the definition of an entity type *E*, we will call such a set of relationship types the *generative set of E* and denote it with $\gamma(E)$. Consequently, the relationship types in $\gamma(E)$ are always constant with respect to *E*. For the example shown in Figure 1, the relationship types of *Money* with *Decimal* and *Currency* form the generative set of *Money*, because they are used to "generate" and identify instances of *Money*, and are therefore significantly different from other relationship types of *Money*, such as the one with *Product* whose instances do not affect the population of *Money*.

It can be easily concluded that if all other entity types that participate in the relationship types in $\gamma(E)$ are constant, *E* is also constant. However, one can imagine a more general case where there is a relationship type *R* in $\gamma(E)$ with a non-constant class *C* whose instances are dynamically created and destroyed by actions during runtime. The definition of *E*'s population remains the same, but the population is not constant any more. Namely, when an instance *c* of *C* is removed from *C*'s population, all instances *e* of *E* related to *c* by a relationship type from $\gamma(E)$ should also cease to exist, as their existence is defined in terms of their relationships with instances of *C*. Thus, the population of *E* changes over time, as the population of *C* changes. As a

conclusion, an entity type *E* is constant if and only if $\gamma(E)$ relates *E* with constant entity types only.

Consequently, if the notion of the generative set is to be taken as the basis for defining populations of structured data types, then the previous conclusion calls for:

- either a restriction that $\gamma(E)$ for a structured data type *E* cannot relate *E* with a non-constant entity type; in that case, structured data types are constant entity types;
- or for a revision of the claim that data types have to be constant entity types; as just shown, if $\gamma(E)$ for a data type *E* relates *E* with a non-constant entity type, then *E* is also non-constant.

We will take the first approach in this paper in order to stay aligned with a commonly accepted conception that data types are constant entity types.

4.3. Constant Classes

In a traditional sense, as well as in UML, an ordinary, non-constant class specified in a conceptual model designates a set of objects that can potentially exist in the system's object space in runtime. This (typically infinite) set of potential objects is basically defined by intensional semantics, in terms of the features that objects of the class will have; all objects and only objects that have the features of a specified class are potential instances of that class. On the other hand, the actual set of objects that exist in the system's object space at a particular point in runtime is defined by extensional semantics, in terms of explicit Create, Reclassify, and Destroy Entity actions that have been executed up to that point. Such set of objects is always finite. In order to ensure clear disambiguation, we will refer to the former (the usually infinite set of potential objects defined intensionally) as the *extent* of the class or entity type in general, and to the latter (the set of actually existing objects at a certain point in runtime) as the *population*. One important corollary of this definition is that the population of a data type is always equal to its extent, that is, the population covers the entire extent and is constant. This is a fundamental observation and assumption that underpins the further ideas presented in this paper.

As opposed to a non-constant class, the extent and population of a constant class are the same, and are defined in design time rather than in runtime. In fact, this observation conceals a subtle potential inconsistence. Namely, objects of classes are created in an explicit, extensional manner, by executing Create Entity actions. Execution of actions is naturally bound to runtime. Hence, objects of constant classes cannot be "created in design time," but can only be "specified" (defined) in an extensional manner in design time, for example by so-called instance specifications of UML [16] (see, for example, creational object specifications in [14] as an advanced example). Anyhow, these specifications are manifested as actions being executed in runtime. Every successfully executed Create Entity action in turn

modifies the population of the entity type. However, if a class is constant, it does not allow such actions. Therefore, totally constant classes, in this purely theoretical sense, actually cannot exist.

It seems that this inconsistence, however, is purely conceptual and does not have a significant practical impact. Constant classes may be defined as those whose instances are created in some strictly defined and controlled periods of runtime, e.g., at system's startup or by certain restricted administrative, initialization, or configuration procedures only.

An alternative resolution could be a relaxation of the notion of extensional population definition: an instance specification, made in a model and in design time, does not have to result in an action of creating an instance in runtime, but may be conceptually treated as a formal statement that such an instance (with its optionally specified relationships) merely exists, with the lifetime that has not its "creational beginning nor ending during runtime" but spans over the entire system's lifetime, and is a member of the population of its entity type ab initio.

This is just the case with enumerations: in essence, an enumeration is nothing more than a constant class whose instances can be referred to by literals specified in design time. Indeed, if an enumeration can take part in relationship types (either constant or non-constant with respect to itself) and can have other features (such as operations), there is not any semantic difference between a constant class and an enumeration, as both must have populations defined extensionally, in design time. This is really the case in some languages, such as Java, where an enumeration is nothing but a constant class with objects enumerated in the program and identified by literals. UML does not prevent enumerations from having relationship types and other features as well. However, in UML, "enumeration is a kind of data type, whose instances may be any of a number of user-defined enumeration literals" [16, p. 67]. Except from being influenced from some older programming languages, such as C/C++, and the semantics and implementation of enumerations in them, there seems to be no rationale behind the decision that enumerations are data types. We argue that this is yet another design flaw of UML that makes its semantics vague. From our perspective, the extensional nature of the definition of an enumeration's population is a much stronger qualifier that classifies it into classes, as opposed to the identifiability of its constant set of instances through literals, which is a characteristic applicable to constant classes as well.

In general, a constant class can take part in constant as well as in non-constant relationship types (with respect to itself); it seems that there is not a valid reason for any restriction in this respect.

5. Definitions

In this section, we provide a summary of the conclusions from the previous analysis in a form of precise definitions. Our definitions are based on the following observations described informally so far:

- Extents of both kinds of entity types are defined intensionally. Populations, on the other hand, are defined differently: extensionally for classes and intensionally for data types. This reflects our basic orientation to providing clear runtime semantics.
- Data types are constant entity types and their populations are equal to their extents.
- Populations of structured data types are defined as all allowed cross-combinations of different relationships of the relationship types from a generative set.

In the discussions that follow, we use the following notation:

- $\varepsilon(T)$ denotes the extent of an entity or relationship type T, that is, the (possibly infinite) set of all possible instances of T allowed by the model and constraints in it.
- $\pi(T)$ denotes the population of an entity or relationship type T, that is, the set of all existing instances of T at a certain moment in runtime; $\pi(T)$ obeys all implicit and explicit constraints defined by the model.
- $\pi(S)$ denotes the entire system's population, that is, populations of all entity and relationship types from the considered conceptual model at a certain (but the same) moment in time; the populations altogether obey all implicit and explicit constraints defined by the model.
- In symbols for entities, subscripts differentiate types of those entities, while superscripts differentiate instances within the same entity type.

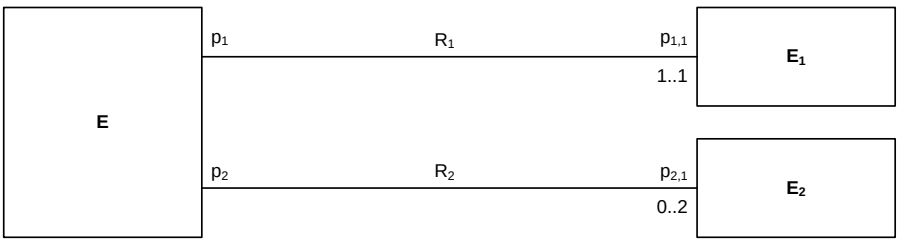


Figure 2. An example of an entity type E and its generative set $\gamma(E)=\{R_1, R_2\}$

Prior to giving a general definition of generative set, we first develop it for a simpler case with binary relationship types shown in Figure 2. The example shows an entity type E and its generative set $\gamma(E)$ consisting of two binary relationship types R_1 and R_2. There are also some cardinality constraints (called multiplicities in UML) associated with the relationship types, so that R_1 is total and functional with respect to E ($p_{1,1}$: $\pi(E)\rightarrow\pi(E_1)$), while R_2 is neither total nor functional ($p_{2,1}$ is multivalued and partial because of the 0..2

constraint). In addition, we may assume a set C of domain-specific constraints defined in the model, in a form of predicates that take entities as parameters.

The idea of the generative set $\gamma(E)=\{R_1, R_2\}$ is that there is exactly one $e \in \pi(E)$ for each valid combination of relationships with all existing entities of E_1 and E_2 so that all cardinality constraints and all predicates in C are also satisfied. For example, if there is only one existing instance e_1^1 of E_1 and two existing instances e_2^1 and e_2^2 of E_2, there are exactly four instances of E, related to instances of E_1 and E_2 as given in Table 2, assuming that C holds for all of them.[6]

Table 2. The population of entity type E defined by its generative set $\gamma(E)$ in Fig. 2

Entity of E	*Relationships of R_1*	*Relationships of R_2*
e^1	$\{<p_1:e^1>,<p_{1,1}:e_1^1>\}$	None
e^2	$\{<p_1:e^2>,<p_{1,1}:e_1^1>\}$	$\{<p_2:e^2>,<p_{2,1}:e_2^1>\}$
e^3	$\{<p_1:e^3>,<p_{1,1}:e_1^1>\}$	$\{<p_2:e^3>,<p_{2,1}:e_2^2>\}$
e^4	$\{<p_1:e^4>,<p_{1,1}:e_1^1>\}$	$\{<p_2:e^4>,<p_{2,1}:e_2^1>\}$, $\{<p_2:e^4>,<p_{2,1}:e_2^2>\}$

Just as another example, if the constraint at $p_{1,1}$ were 0..1, the number of instances of E would be eight: four of them would be related as shown in the given table, and the other four would be unrelated to any instance of E_1 and related to the instances of E_2 as shown in the table.

In our formal definitions, we introduce the notion of *relationship projection*: informally, this is a tuple obtained from a relationship by omitting a certain role. For example, the projection of the relationship $\{<p_1:e^1>,<p_{1,1}:e_1^1>\}$ from Table 2, for the role p_1, denoted with $\Pi(\{<p_1:e^1>,<p_{1,1}:e_1^1>\},p_1)$, is $\{<p_{1,1}:e_1^1>\}$, while $\Pi(\{<p_2:e^4>,<p_{2,1}:e_2^2>\}, p_2)= \{<p_{2,1}:e_2^2>\}$.

Similarly, the projection of an entity e for the given role p, denoted with $\Pi(e,p)$, is the set of projections of all relationships in which e participates with the role p. For the same example in Table 2, $\Pi(e^3,p_2)=\{\{<p_{2,1}:e_2^2>\}\}$, while $\Pi(e^4, p_2)=\{\{<p_{2,1}:e_2^1>\}, \{<p_{2,1}:e_2^2>\}\}$.

Finally, the projection of the given entity e for the entire ordered set of given roles p_1, p_2, ..., p_n, is an n-tuple consisting of projections for each role: $\Pi(e, <p_1, p_2,..., p_n>)=<\Pi(e,p_1), \Pi(e,p_2), ..., \Pi(e,p_n)>$. For the same example in Table 2, $\Pi(e^4,<p_1,p_2>)=<\{\{<p_{1,1}:e_1^1>\}\}, \{\{<p_{2,1}:e_2^1>\},\{<p_{2,1}:e_2^2>\}\}>$.

The purpose of the notion of projections is to allow for considering and comparing different possible and well-formed configurations of relationships

[6] We assume that the roles have the semantics of unique association ends in UML as discussed in [14]. If a role is non-unique, the result is slightly different, but the core idea remains the same.

for the generative set of a certain entity type *E*, independently of concrete entities of *E* that take part in them.

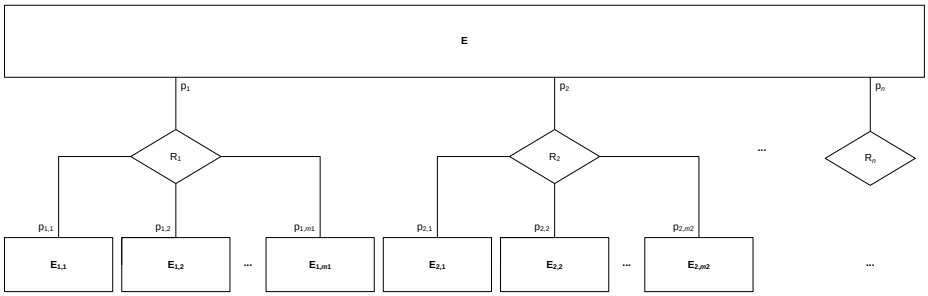


Figure 3. A general case of an entity type *E* and its generative set $\gamma(E)=\{R_1, R_2, ..., R_n\}$

Now we can get to general definitions. In the following definitions, these assumptions and notations are used (Figure 3):

- Let *E* be an entity type.
- Let $\gamma=\{R_1, R_2, ..., R_n\}$ be a finite set of relationship types such that *E* participates in each, where $R_i(p_i:E, p_{i,1}:E_{i,1}, p_{i,2}:E_{i,2}, ..., p_{i,mi}:E_{i,mi})$, $m_i \geq 1$, $1 \leq i \leq n$.

Definition (Projections). (See Figure 3 for assumptions and notation.)

- Let $r_i \in \varepsilon(R_i)$ be a relationship from the extent of R_i, $r_i=\{<p_i:e>, <p_{i,1}:e_{i,1}>, <p_{i,2}:e_{i,2}>, ..., <p_{i,mi}:e_{i,mi}>\}$, $e \in \varepsilon(E)$, $e_{i,j} \in \varepsilon(E_{i,j})$, $1 \leq i \leq n$, $j=1,...,m_i$. The *projection* of r_i for p_i, denoted with $\Pi(r_i, p_i)$, is the set:

  $\Pi(r_i, p_i)=\{<p_{i,1}:e_{i,1}>, <p_{i,2}:e_{i,2}>, ..., <p_{i,mi}:e_{i,mi}>\}$.

- Let $e \in \varepsilon(E)$ be an entity from the extent of *E* and p_i the role played by *E* in a relationship R_i, $1 \leq i \leq n$. The *projection* of *e* for p_i, denoted with $\Pi(e, p_i)$, is the set of projections of all relationships with existing entities of $E_{i,j}$ in which *e* plays the role p_i:

  $\Pi(e, p_i)=\{\Pi(r_i, p_i) | r_i=\{<p_i:e>, <p_{i,1}:e_{i,1}>, <p_{i,2}:e_{i,2}>, ..., <p_{i,mi}:e_{i,mi}>\},$
  $r_i \in \varepsilon(R_i), e_{i,j} \in \pi(E_{i,j}), j=1,...,m_i\}$.

- Let $e \in \varepsilon(E)$ be an entity from the extent of *E*. The *projection* of *e* for the roles $p_1, p_2, ..., p_n$ played by *E* is an *n*-tuple consisting of projections for each role:

  $\Pi(e, <p_1, p_2, ..., p_n>)=<\Pi(e, p_1), \Pi(e, p_2), ..., \Pi(e, p_n)>$.

Definition (Generative set). (See Figure 3 for assumptions and notation.) γ is a *generative set* of E, denoted with $\gamma(E)$, if and only if (by definition):
i) In any system's population, there are no two different entities of E with the same projections for the roles of E in the relationship types from γ:

$(\forall \pi(S))(\forall e^1, e^2 \in \pi(E))(\Pi(e^1, <p_1, p_2,..., p_n>)=\Pi(e^2, <p_1, p_2,..., p_n>) \Rightarrow e^1=e^2)$.

ii) For any system's population, and for any possible well-formed configuration of relationships for γ, that is, for any legal projection, there is an existing entity of E in that population with that projection:

$(\forall \pi(S))(\forall e^1 \in \varepsilon(E))$(relationships of e^1 satisfy all model constraints $\Rightarrow$
$(\exists e^2 \in \pi(E))(\Pi(e^1,<p_1, p_2, ..., p_n>)=\Pi(e^2,<p_1, p_2, ..., p_n>)))$.

These definitions imply a function Π_γ that maps $\pi(E)$ on all possible projections of relationships allowed by the model and its constraints, defined as $\Pi_\gamma(e)=\Pi(e, <p_1, p_2,..., p_n>)$. The clause *i*) of the latter definition means that Π_γ is injective, while the clause *ii*) means that Π_γ is surjective.

Being both injective and a surjective, Π_γ is a bijection. Being injective, Π_γ is an observation term. As it is also a surjection, its corresponding identification function f_{Π} is total and is equal to Π_γ^{-1}. It can be shown that the statement *ii*) of the given definition implies that there is not any potential true extension of the population of E and of the populations of some or all of the relationship types from $\gamma(E)$ that satisfies all model constraints and preserves injectivity of Π_γ.

It should be noted that surjectivity differentiates Π_γ from an observation term in general. For the example in Figure 2, if $\gamma(E)$ were a simple observation term that is not also a generative set, then instances of E would not have to exist for each combination given in Table 2; for a generative set, this is mandatory (i.e., f_{Π} is total for a generative set). In addition, the relationship types from a generative set do not have to be reference types as defined in [17], because the participation of E in reference relationship types has to be total, while this is not necessary for a generative set; an example is the participation of E in R_2 in Figure 2 with the lower multiplicity bound equal to 0 at the opposite end.

Now we can formulate the definitions of different kinds of data types and classes.

Definition (Primitive data type). A *primitive data type* is a constant entity type without a generative set, whose population definition is intensional and whose instances can be identified by literals.

Definition (Structured data type). A *structured data type* is an entity type *E* whose population definition is intensional and has a generative set $\gamma(E)$, where any $R \in \gamma(E)$ has all participating entity types other than *E* constant.

Definition (Class). A *class* is an entity type without a generative set, whose population definition is extensional.

Definition (Enumeration). An *enumeration* is a constant class whose instances can be identified by literals.

We will illustrate the meaning of the given definitions, especially of the one of a generative set, on two examples.

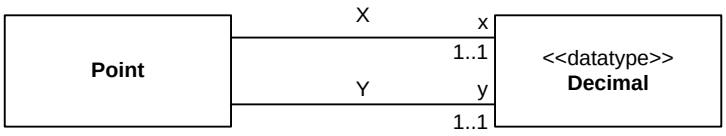
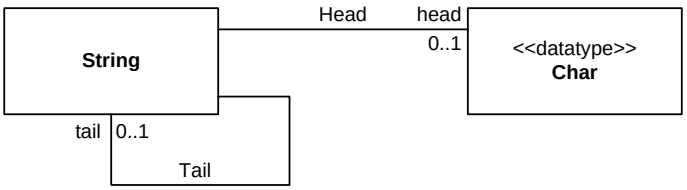


Figure 4. Example of entity type *Point*

Example (*Point*). The example in Figure 4 shows the entity type *Point* with two relationship types: *X(Point, x:Decimal)* and *Y(Point, y:Decimal)* with 1..1 cardinality constraints on *x* and *y*. If these two relationship types form the generative set for *Point*, then *Point* is a data type. In that case, the population of *Point* is defined intensionally and consists of one and only one instance of *Point* for each pair of existing instances of *Decimal* (it is basically isomorphic to the Cartesian product of *Decimal* with itself). On the other hand, if these two relationships do not have the semantics of a generative set, then *Point* is a class and its population is managed explicitly, by creating and deleting objects with actions executed in runtime. In practice, the modeler considers the case the opposite way: if she wants to have the semantics of an intensionally defined and constant population of *Point*, without having to manage its population in runtime by actions, she will declare *Point* as a data type, and the two relationship types will form the generative set of *Point*; otherwise, she will declare it as a class.




```
{context String inv:
    self.tail<>self and
    self.tail->size()+self.head->size()<>1}
```

Figure 5. Example of entity type *String*

Example (*String*). The example in Figure 5 illustrates another, more interesting case, where instances of entity type *String* can be recursively constructed from other instances of the same type. The constraint given below the diagram, along with the implicit constraints given in the multiplicities of the roles *head* and *tail*, specifies that only the following cases are allowed:

i) A *String* can have an empty head and an empty tail. This is the case of an empty string.

ii) A *String* can have one character as its head and another string as its tail. (Note that the tail can be an empty string.) This defines the recursive nature of constructing strings.

If the two relationship types are treated as a generative set and *String* as a data type consequently, there will be a logically infinite population of *String*, defined conceptually as follows:

- there is one and only one instance with no head and no tail, the "empty string;"
- for each existing instance of *Char* and each existing instance of *String*, there is another instance of *String* that is constructed by concatenating the character and the former string.

As before, in the opposite case when the two relationships do not have the semantics of a generative set, *String* is a class and its population has to be managed explicitly, by creating and destroying instances with actions. In that case, there can be two instances of *String* that are different identities, but which can be treated as equal strings with respect to their isomorphic structure, when they have the same character as their heads and (recursively) equal tails.

This example also suggests a general approach of defining other recursive structured data types, such as lists, trees, and others. It also shows the dual nature of those classical structures, since they can be treated as either classes or data types, depending on the intent of the modeler.

There are several interesting implications and properties that can be observed from the presented definitions:

- The extent and the population of a data type do not have to be finite. Since the definition of a primitive data type's population is intensional, it can be logically infinite.[7] The described example of the data type *String* is an example of a structured data type with an infinite population. The similar holds for a data type *BinaryLargeObject* (BLOB), which can be treated as primitive, and whose instances are finite but unlimited strings of bits.
- The population of a structured data type can be either finite or infinite, depending on the finiteness of the populations of other entity types that take part in its generative set, and on the way the generative set is defined.
- Due to the limitation that all entity types other than the considered structured entity type that take part in its generative set are constant, the extent and the population of the structured data type are equal, i.e., it is also constant.
- If all other entity types that take part in the generative set of a structured data type *E* are identifiable, then *E* is also identifiable through the observation term Π_{γ}. For example, in a typical (but not necessary) case when the relationship types in $\gamma(E)$ relate *E* only with primitive types, other identifiable structured data types, or enumerations, *E* is identifiable. *E* is not identifiable if, for example, it depends on a non-identifiable constant class.
- Due to its extensional definition, the population of a class is always finite, but generally unlimited.

Table 3. Summary of the main characteristics of different kinds of entity types

Entity type kind	*Population definition*	*Generative set*	*Population*	*Identifiable*	*Population finite*
Primitive data type	Intensional	No	Constant	Yes	Yes or No
Structured data type	Intensional	Yes	Constant	Yes or No	Yes or No
Constant class	Extensional	No	Constant	Yes or No	Yes
Enumeration	Extensional	No	Constant	Yes	Yes
Non-constant class	Extensional	No	Variable	Yes or No	Yes

[7] Typically, all such populations are physically finite, because they are limited by the amount of physically addressable computer memory or storage, but there is often no logical limit.

All these definitions and conclusions can be summarized in Table 3 and Figure 6.

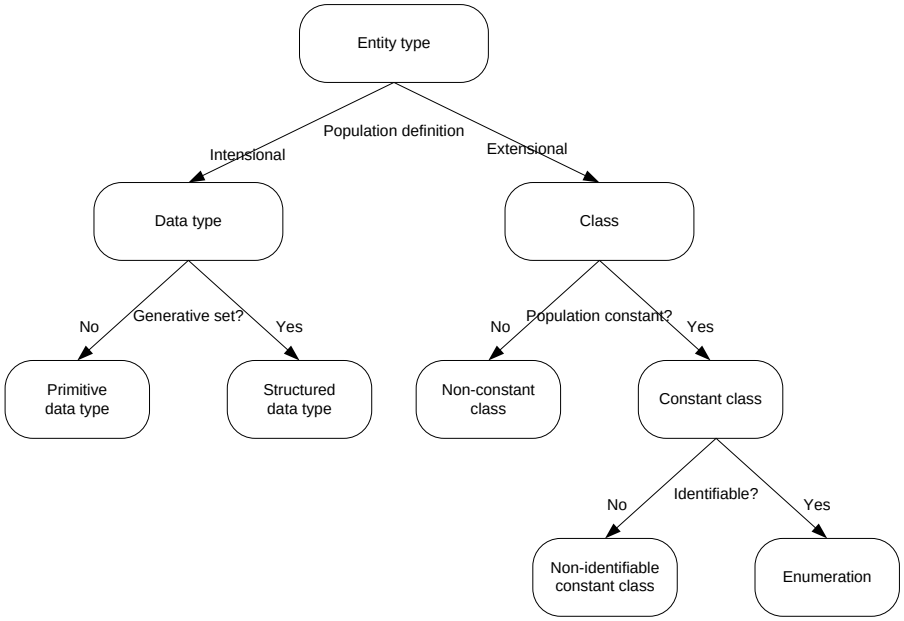


Figure 6. Classification of different kinds of entity types

6. Consequences

In this section, we analyze some consequences of the given semantic interpretations: how they relate to the definitions in UML, the value-based semantics, and the distinction between attributes and associations.

6.1. Classes vs. Data Types in UML

Populations of data types considered here are constant and defined intensionally. Therefore, in a certain sense, instances of data types (generally, of constant, intensionally defined entity types) can be considered as residing *outside* the scope of the running system, because the behavior of the system does not affect these populations and does not even define them explicitly. On the other hand, these populations are typically very large and sometimes unlimited. These are reasons why systems most often do not physically store instances of data types as individual physical entities referenced in relationships. Instead, a system internally uses *values* that are

actually *identifiers* or *references* to those external instances, in all places such as relationships. For example, a 32-bit two's complement binary representation with all digits equal to one is a value that refers to the conceptual notion of minus one. Such values are then stored and manipulated wherever needed. Of course, this interpretation does not prevent an implementation to store each physical instance of a data type and use compact references to it in all other places, if this is cost-effective. The decision, of course, depends on different engineering trade offs but is certainly an implementation, not a conceptual concern. For example, if the space required for storing a reference or a key to the instance in a map is comparable to the space of storing the actual instance, the system may opt not to store physical instances. If, on the other hand, the system opts to store particular instances of a data type, when the population of a data type is huge or unlimited, the system may also opt to store only instances that are referred to from relationships with other entities. For example, if an implementation estimates that only a tiny subset of the entire extent of 64-bit integers will be referred to in the system, it can store only the used instances in a central repository and provide short integer keys to those values in a map; it is then much more cost-effective to manipulate with those shorter keys instead of "original" 64-bit values throughout the system.

These observations bring us to the rationale behind defining data types in standard UML as well as in OOIS UML [14] as described in Section 3. There, instances of data types are actually only *typed references* to external entities, but not real entities. The remaining semantic implications follow directly from that observation:

- "Instances of data types in UML do not have their identity, they are pure values; equal values represent the same instance. Instances of data types are compared for equality on value base." Indeed, values are typed references to entities and do not have identity themselves; instead, they refer to external entities with identity. Obviously, equal references refer to the same entity, because references are inherently unique identifiers.
- "Instances of data types can be arbitrarily copied and passed by value (e.g., as parameters)," again because they are simple typed references.
- "Instances of data types are immutable, and operations of data types cannot change any instance; instead, they are pure functions that can create and return new values." Really, values in UML serve as references to and proxies of external entities, while data types are constant entity types. Therefore, operations on such values are operations called on proxies. They cannot create new entities of constant entity types, but can create and return references to external, already existing entities.
- "As opposed to objects of classes, instances of data types have implicit lifetime: they disappear as soon as they are not needed any more" (this is a rule specific for the OOIS UML profile). Indeed, a reference (as an implementation thing) is just a handle that is used to access an entity; it can be discarded implicitly, as soon as it is not needed in its scope or

when its temporal scope expires. On the other hand, in UML and OOIS UML, instances of data types are created by a Create Instance action. This is, formally, a minor semantic flaw. As it implies from this discussion, when a "data type instance is created" in UML, only a reference to an external entity is created and obtained; that external entity has to be identified then. More concretely, in a textual notation, instead of using a statement e.g.:

```
Integer i = new Integer(5);
```

a more appropriate notation would be:

```
Integer i = Integer(5);
```

because this is actually not an action that creates a new instance of an entity type, but a request for a reference to an external entity identified by the parameters of the request ("constructor").

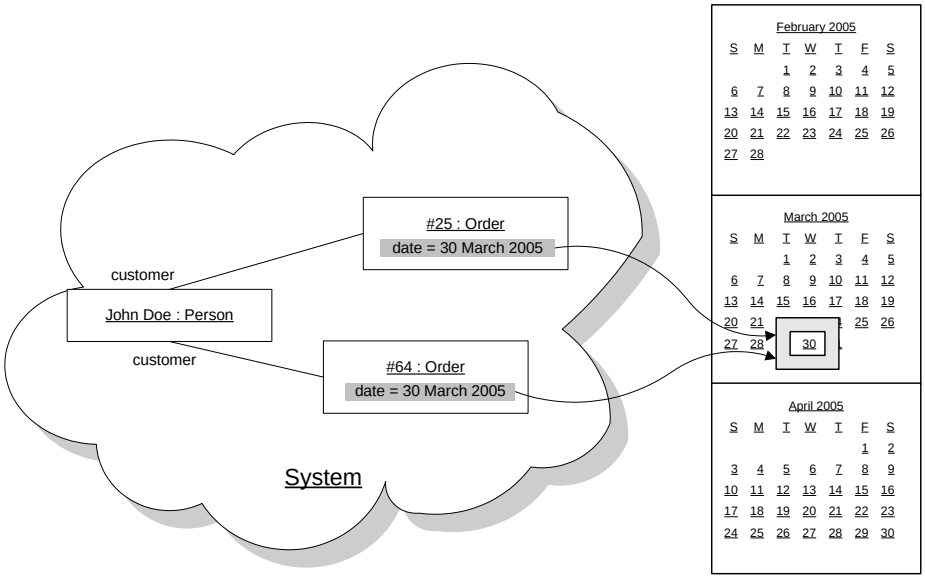


Figure 7: Instances of data types, such as *Date*, as defined in UML, represent values that refer to external entities that reside outside the system's boundaries

The decision whether an entity should be modeled with a class or with a data type is, thus, basically a matter of defining the boundaries of a system under consideration, as shown in Figure 7. If the system does not want to manage the population explicitly, but the population is defined outside the system's boundaries, then the entity type is a data type, while the values manipulated by the system refer to these external entities. For the example in Figure 7, objects of class *Order* are related with external entities of type *Date* that reside outside the system boundaries and abstract the common notion of

a day. If, however, the system wants to define the population explicitly, by enumerating them or by creating and deleting them with actions, then the entity type is a class. For example, in an event management system, *Day* can be a class that conceptualizes the explicitly defined limited set of days of an event, possibly related to the external notion of *Date* in order to be properly positioned in the calendar and described additionally for human readability.

6.2. Value-Based vs. Object-Based Semantics

Following the same deduction in a more general context, value-based systems, such as relational data models, found their semantics on the fundamental assumption that real entities and relationships between them reside outside the running system and its data space, while the system manipulates data values that refer to those entities and relationships. More precisely, by storing and manipulating values and tuples of values, the system actually manipulates *facts* that certain external entities exist, are instances of their types, and are related with certain relationships. The existence of many occurrences of the same value or the same tuple of values in the system has no particular meaning, as they all represent replicas of the same fact that holds in the external domain. In some sense, the system does not affect the external domain with its execution, nor does it create or delete entities and relationships, but just tracks the facts from the external system and changes its scope of concern: at some time, some facts are relevant for the system and some are simply not. An absence of a fact (a value or a tuple) in the system does not necessarily mean that the thing referred to by that fact does not exist in the domain: it may not exist, or it may exist, but it is simply not relevant for the system or not captured by it at that time. This is why values and tuples do not have identities in such systems, and why modeling languages based on such semantics require that all entities be identifiable, as those systems have to maintain references (identifiers) to external entities.

On the other hand, object-based systems maintain entities that have their inherent identity and lifetimes internal to the system. Although such entities (objects) often *model* external concrete or abstract entities from the problem domain, like persons, things, or logical concepts, they also often represent pure abstractions invented in the implementation domain. The system maintains such entities by creating and deleting them, and by changing their states and properties, i.e., by creating and deleting relationships between entities. In such systems, entities have their inherent identities and do not have to be identifiable as in value-based systems. More detailed explanations on this aspect with examples can be found in [14].

It seems that the recent understandings and trends in the design of languages and systems for conceptual modeling and model execution, as well as in designing data models in practice even with the relational paradigm take the second approach. Indeed, it is a common practice in designing

relational databases to introduce so called technical IDs as internally generated, outwardly invisible primary keys of records to ensure uniqueness of records in a certain table that cannot be identified by other means or whose "natural" (domain) identifiers are bulky. Such engineering technique is, actually, a reification ("objectization") of a tuple.

6.3. Attributes vs. Associations

The previous conclusions also justify the clear and precise discriminating characteristics of associations vs. attributes as defined in OOIS UML [14]:

- Associations are relationship types that can relate classes only; types of attributes can only be data types or enumerations.[8] This restriction clearly indicates that instances of attributes, i.e., attribute values, are references to external entities that reside outside the boundaries of the running system or to internal instances of enumerations whose populations are constant, as illustrated in Figure 4. Associations, on the other hand, model relationships (called links in UML) that are structural connections between objects, whereby the lifetime of both objects and links is managed by the system, i.e., by executed actions.
- Put the other way, (structured) data types can only have attributes, and cannot participate in associations. This rule clearly distinguishes the relationship types that form the generative set of a structured data type: they are modeled as its attributes, which fully fits into intuitive understanding of their purpose and meaning. On the other hand, the other relationship types with that same data type (that are not in its generative set) are modeled as attributes of other entity types: classes, when those attributes represent relationships that cross the system's object space boundaries, or attributes, when those attributes represent the generative set of another structured attribute whose definition is dependent on the former data type.
- The lifetime of an attribute value is bound to the lifetime of its owner object or another data type instance, while the lifetimes of objects are managed by the system and are generally independent (unless constrained by other specific model elements, such as propagated destruction [14]). Really, in this interpretation, attribute values are only references used within their owner objects to refer to external entities, i.e., relationships that cross the system's boundaries. This is why they are not needed when their scope ceases to exist. Attributes of data types actually model relationships that reside completely outside the system boundaries, so the system does not have to manage them through actions.

[8] In UML, as discussed previously, enumerations are classified as data types.

- Data types have implicit copy semantics; for an instance of a structured data type, a clone is a deep copy obtained by cloning the original's attribute values recursively. Again, this makes sense because attribute values of data types are just references to external relationships from the generative set and references can be copied.

All these rules explain that the support for classes and data types, as well as for associations and attributes in systems based on UML semantics, actually allows for mixtures of object-based and value-based semantics in the same system. The first given rule is the most important one. It seems to be a sounder rule for distinguishing associations vs. attributes than the rule that types of attributes should be entity types defined outside the model mentioned in Section 3. While in the latter rule it only matters whether an entity type is owned by the model of the considered system or is defined elsewhere, which is purely a matter of (static) model organization, packaging, and model compilation, it has nothing to do with the runtime semantics of the concepts. On the other hand, our rule is based completely on runtime semantics, which is, we believe, more sensible and appropriate for practical use, as it reduces confusion. Our experience shows that a modeling concept that has impact on the execution of the system is clear, easy to adopt, and unambiguous in use only if it has clear runtime semantics. We find this fact as the main advantage of programming languages over yet immature modeling techniques (without fully formal and executable semantics) and the main reason of a rather poor scale of adoption of model-driven development methods in industry.

7. Conclusions

We have provided a new interpretation of the semantic difference between classes and data types in conceptual modeling and UML. It is based on the difference between how populations are defined: extensionally for classes and intensionally for data types. Extensional definitions assume explicit statements that certain instances belong to the population, by enumerating instances or by executing actions in runtime. On the other hand, populations of data types are defined intensionally, by stating the conditions for belonging to the population, or describing the population implicitly. In the special case of structured data types, we have proposed the notion of a generative set, consisting of the relationship types that are used in defining the population of a structured data type: informally, the population is defined as a total and unique coverage of all allowed configurations of relationships from the generative set and for all existing entities of other entity types taking part in the generative set. We have also suggested that, according to this distinction, enumerations belong to classes rather than to data types. It has been suggested that in practice, the decision whether an entity should be modeled with a class or with a data type is, basically, a matter of defining the boundaries of a system under consideration, and whether the responsibility of

managing populations of entity types is inside or outside the system's boundaries.

The proposed interpretation allows for unambiguous discrimination of the related concepts, yet it fits into intuitive understanding and common practical usage of these concepts. In addition, our interpretation provides a clear explanation of apparent discrepancies between definitions taken in conceptual modeling and in UML.

We have also described some semantic implications of the given interpretation that clearly distinguish value-based vs. object-based semantics, associations vs. attributes, and identity vs. identification. We strongly believe that such precise definitions, based on formal runtime semantics of the concepts, are crucial for practical use and wider adoption of model-driven software development techniques in practice. In fact, the described interpretations emerged from and have been verified in our practical experience in applying the described interpretations and model-driven development with an executable UML profile and its implementation in the SOLoist framework (www.soloist4uml.com), in many industrial projects.

One topic of interest that has not been covered in this paper is the formal semantic alignment of the notion of generalization/specialization (i.e., inheritance, subsumption) relationship between entity types with the interpretations given in this paper, in particular, the coherence of intensional definitions of populations via generative sets in presence of specializations. Our internal analysis shows that the notion of specialization fully accords with the given interpretations. However, a deeper study is left for some future work and publications.

Acknowledgments. The author is grateful to the anonymous reviewers for their very thorough and qualified reviews that have inspired him to improve the paper in several aspects, and clarify or correct it in many places.

References

1. Barbier, F., Henderson-Sellers, B., Le Parc-Lacayrelle, A.,Bruel, J.-M.: Formalization of the Whole-Part Relationship in the Unified Modeling Language. IEEE Trans. Software Engineering, Vol. 29, No. 5, 459-470. (2003)
2. Beeri, C.: A Formal Approach to Object-Oriented Databases. Data and Knowledge Engineering, Vol. 5, No. 4, 353-382. (1990)
3. Bourdeau, R. H., Cheng, B. H. C.: A Formal Semantics for Object Model Diagrams. IEEE Trans. Software Engineering, Vol. 21, No. 10, 799-821. (1995)
4. Chen, P. P.: The Entity-Relationship Model. ACM Trans. on Database Systems, Vol. 1, No. 1, 9-36. (1976)
5. Diskin, Z., Dingel, J.: Mappings, Maps and Tables: Towards Formal Semantics for Associations in UML2. In O. Nierstrasz et al. (eds.): Proc. of MoDELS 2006, Lecture Notes in Computer Science 4199, Springer-Verlag, 230–244. (2006)
6. Diskin, Z., Kadish, B.: Variable Set Semantics for Keyed Generalized Sketches: Formal Semantics for Object Identity and Abstract Syntax for Conceptual Modeling. Data & Knowledge Engineering, Vol. 47, No. 1, 1-59. (2003)

7. France, R. B.: A Problem-Oriented Analysis of Basic UML Static Requirements Modeling Concepts. In Proc. 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '99), ACM SIGPLAN Notices, Vol. 34, No. 10, 57-69. (1999)
8. Génova, G., Llorens, J., Fuentes, J. M.: UML Associations: A Structural and Contextual View. Journal of Object Technology, Vol. 3, No. 7, 83-100. (2004)
9. Génova, G., Llorens, J., Martínez, P.: The Meaning of Multiplicity of N-ary Associations in UML. Software and Systems Modeling, Vol. 1, No. 2, 86-97. (2002)
10. Gogolla, M.: Identifying Objects by Declarative Queries. In Advances in Object-Oriented Data modeling, Papazoglou, M. P., Tari, Z., eds., MIT Press, 255-277. (2000)
11. Guttag, J.V.: The Specification and Application to Programming of Abstract Data Types. Ph.D. Thesis, Dept. of Computer Science, University of Toronto. (1975)
12. Guttag, J.V.: Algebraic Specification of Abstract Data Types. In Broy, M., Denert. E., (eds.): Software Pioneers, Springer (2002)
13. Mazur, B.: When is one thing equal to some other thing?. http://www.math.harvard.edu/~mazur/preprints/when_is_one.pdf, June 2007. (Retrieved December 2011)
14. Milicev, D. Model-Driven Development with Executable UML. John Wiley & Sons. (2009)
15. Milicev, D.: On the Semantics of Associations and Association Ends in UML. IEEE Trans. Software Engineering, Vol. 33, No. 4, 238-251 . (2007)
16. Object Management Group: UML Superstructure Specification, Version 2.4.1, http://www.omg.org. (2011)
17. Olivé, A.: Conceptual Modeling of Information Systems. Springer. (2007)
18. Övergaard, G.: A Formal Approach to Relationships in the Unified Modeling Language. In Proc. PSMT'98 Workshop on Precise Semantics for Modeling Techniques, Technische Universität München, TUM-I9803. (1998)
19. Övergaard, G.: Formal Specification of Object-Oriented Modelling Concepts. PhD Thesis, Department of Teleinformatics, Royal Institute of Technology, Stockholm, Sweden. (2000)
20. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W.: Object-Oriented Modeling and Design. Prentice-Hall International. (1991)
21. Rumbaugh, J., Jacobson, I., Booch, G.: The UML Reference Manual. Addison-Wesley. (2005)
22. Selic, B.: The Pragmatics of Model-Driven Development. IEEE Software, Vol. 20, No. 5, 19-25. (2003)
23. Selic, B., Ramackers, G., Kobryn, C.: Evolution, Not Revolution. Communications of the ACM, Vol. 45, No. 11, 70-72. (2002)
24. Stevens, P.: On the Interpretation of Binary Associations in the Unified Modeling Language. Software and Systems Modeling, Vol. 1, No. 1, 68-79. (2002)

Dr. Dragan Milićev is Associate Professor and Chairman of the Software Engineering Department at the University of Belgrade, Faculty of Electrical Engineering (www.etf.rs). He is specialized in software engineering, model-driven development, UML, object-oriented programming, software architecture and design, information systems, and real-time systems. He has published papers in some of the most prestigious scientific and professional

journals and magazines, contributing to the theory and practice of model-driven development and UML. He is also a member of the program committees of two premier international conferences on model-based engineering (MODELS and ECMFA). He is the author of three previous bestselling books on object-oriented programming and UML, published in Serbian, and a recent book in English, published by Wiley/Wrox, "Model-Driven Development with Executable UML" (also publish in Chinese by Tsinghua University Press, Beijing, China). With more than 25 years of extensive industrial experience in building complex commercial software systems, he has been serving as the chief software architect, project manager, or consultant in a few dozen international projects, some of them having national scope and importance. He is the founder of SOL Software (www.sol.rs), a software research and development company specialized in designing software development tools using model-driven approach, as well as in building custom applications and systems. He was chief software architect and project manager for most of SOL's projects and all its products.

Received: July 16, 2011; Accepted: February 27, 2012.