

Class Diagrams

A fact is conceivable, means
we can picture it.
Ludwig Wittgenstein

Class diagrams form the architectural backbone of many system modeling processes. Hence, this chapter introduces class diagrams defined in UML/P with the core elements *class*, *attribute*, *method*, *association*, and *composition*. The section about *views* and *representations* discusses forms of use for class diagrams. Furthermore, it is shown how modeling concepts are adapted for project-specific demands using *stereotypes* and *tags*.

2.1	Relevance of Class Diagrams	14
2.2	Classes and Inheritance	17
2.3	Associations	22
2.4	View and Representation	27
2.5	Stereotypes and Tags	30

Class diagrams still represent by far the most important and widely used modeling technique of UML. Historically, class diagrams originated from the ideas of entity/relationship modeling [Che76] and the graphical representation of modules, which themselves were influenced by data flow diagrams [DeM79]. Class diagrams describe the structure of a software system and thus form the first discussed core notation for object-oriented modeling.

Appendix C.2 additionally compares the kind of class diagrams introduced here with the UML standard and specifies the syntax of class diagrams.

2.1 Relevance of Class Diagrams

Object-oriented systems are highly dynamic. This makes the modeling of a system's structures a complex task in object-oriented software development. Class diagrams describe this structure, or architecture, of a system, forming the basis for nearly all other description techniques. However, class diagrams and the modeled classes fulfill various tasks.

Structure Modeling

In any object-oriented implementation, the code is organized into classes. Therefore, a class diagram constitutes an overview of the code structure and its internal relations. As programmers are familiar with the concept of *class* from programming, class diagrams used in modeling can be understood and communicated rather easily. Class diagrams are used for showing structural relations of a system and for that reason form the skeleton for almost all other notations and types of diagrams, as these rely on the classes and methods defined in class diagrams. Therefore, they also represent an essential—although not the only—form of description for modeling of software architectures and frameworks.

Classes During Analysis, Design, and Implementation

In analysis, class diagrams are used in order to structure real-world concepts. In contrast, in design and implementation documents, class diagrams are especially used to depict a structural view of the software system. The classes presented in the implementation view can actually be found in implemented systems too. But classes from analysis are often significantly modified, supplemented by technical aspects, or fully omitted when they only belong to the system context.

One of the deficits of UML arises from the less-than-ideal option to explicitly ascribe diagrams a purpose. Assuming that a class diagram reflects an implementation, the semantics of a class diagram can be explained relatively easily and understandably. A number of introductory textbooks about

class modeling or UML take this position [Mey97, Fow00]. Besides, this point of view is often implied by tools. Fusion [CAB⁺94], however, clearly distinguishes between classes belonging to the system and external classes and, thus, demonstrates that modeling of non-software-engineering concepts with class diagrams is feasible and reasonable.

The language profile UML/P is implementation oriented. This is why the following semantics of class diagrams based on the Java code modeled thereby is perfect for this purpose.

Variety of Tasks for a Class

In object-oriented programming and even more so in modeling, classes have numerous tasks. Primarily, they serve to *group* and *encapsulate* attributes and associated methods to create a conceptual unity. By assigning a *class name*, *instances* of the class can be created, saved, and passed on at arbitrary places in the code. Hence, class definitions at the same time act as *type system* and *implementation description*. They can (in general) be instantiated any number of times in the form of *objects*.

In modeling, a class is also understood as the *extension*, i.e., the number of all objects existing at a certain point in time. Due to the explicit availability of this extension in modeling, invariants for each existing object of a class can, for example, be described.

The potential unlimitedness of the number of objects in a system makes cataloging these objects into a finite number of classes necessary. Only this makes a finite definition of an object-oriented system possible. For this reason classes present a *characterization of all possible structures* of a system. This characterization at the same time also describes necessary structural constraints without determining a concrete object structure. As a result, there are usually an unlimited number of different object structures that conform to a class diagram. In fact, each correctly running system can be regarded as an evolving sequence of object structures where at each point in time the current object structure conforms to the class diagram.

In contrast to objects, classes, however, in many programming languages have no directly manipulable representation during the runtime of a system. One exception is, for example, Smalltalk, which represents classes as objects and therefore allows for unrestricted reflective programming.¹ Java is more restrictive, as it allows read-only access to the class code. Generally, reflective programming should be used only very reluctantly because maintenance of such a system gets far more complex due to reduced understandability. This is why reflective programming is ignored in the rest of the book.

¹ In Smalltalk, a class manifests as a normal object during runtime being manipulable like any other object. However, the content of such an object is a description of the structure and behavior of the instances assigned to this class object. See [Gol84].

Classes fulfill the following tasks:

- Encapsulation of attributes and methods in order to create a conceptual unity
- Manifestation of instances as objects
- Typing of objects
- Description of the implementation
- Class code (translated, executable form of the implementation)
- Extension (set of all objects existing at a certain time)
- Characterization of all possible structures of a system

Figure 2.1. Task variety of a class

Metamodeling

Due to the two-dimensional form of model representations, *metamodeling* [CEK⁺00, RA01, CEK01, Béz05, GPHS08, JJM09, AK03] has prevailed as a form of description of a diagrammatic language and thus replaced the grammars commonly used for text. A *metamodel* defines the abstract syntax of a graphical notation. At least since UML standardization, it is customary to use a simplified form of class diagrams as the metamodel language. This approach has the advantage that only one language needs to be learnt. We discuss metamodeling in Appendix A and use a variant of the class diagrams in order to represent the graphical parts of UML/P.

Further Concepts for Class Diagrams

UML offers further concepts that should be mentioned here for the sake of completeness. Association classes, for example, are classes that are attached to the associations that are subsequently introduced to store information that cannot be assigned to any of the classes participating in the association but only to the relation itself. But there are standard processes for modeling such data without association classes.

Modern programming languages such as C++ and Java [GJSB05] as well as UML since version 2.3 [OMG10a] now offer generic types first introduced by functional languages such as Haskell [Hut07]. In Java, this introduction has been integrated nicely [Bra04]. In UML, this has to be done carefully, because types appear in nearly all kinds of diagrams. As generics do not play such an important role in modeling but are applied for reuse of generic components especially in implementation, UML/P waives the full generality of generic classes with wildcards, bound typecasts, etc., and only the most important container classes are offered in a generically realized form, i.e., with type parameters. Thus, UML/P class diagrams do not provide mechanisms for defining generics. OCL/P, however, as well as the code generation allow us to use generics.

2.2 Classes and Inheritance

When introducing classes, attributes, methods, and inheritance, an implementation view—as already discussed—is taken as a basis in this section. Figure 2.2 contains a classification of the most important terms for class diagrams.

Class	A class consists of a collection of attributes and methods that determine the state and the behavior of its <i>instances (objects)</i> . Classes are connected to each other by associations and inheritance relations. The <i>class name</i> identifies the class.
Attribute	State components of a class are called attributes. An attribute is described by its <i>name</i> and <i>type</i> .
Method	The functionality of a class is stored in methods. A method consists of a <i>signature</i> and a <i>body</i> describing the implementation. In case of an <i>abstract</i> method, the body is missing.
Modifier	For the determination of visibility, instantiability, and changeability of the modified elements, the modifiers <code>public</code> , <code>protected</code> , <code>private</code> , <code>readonly</code> , <code>abstract</code> , <code>static</code> , and <code>final</code> can be applied to classes, methods, roles, and attributes. UML/P offers the following iconic variants for the first four modifiers of these: “+”, “#”, “-” and “?”.
Constants	are defined as special attributes with the modifiers <code>static</code> and <code>final</code> .
Inheritance	If two classes are in an inheritance relation to each other, the <i>subclass</i> inherits its attributes and methods from the <i>superclass</i> . The subclass can add further attributes and methods and <i>redefine</i> methods—as far as the modifiers allow. The subclass forms a <i>subtype</i> of the superclass that, according to the <i>substitution principle</i> , allows use of instances of the subclass where instances of the superclass are required.
Interface	An interface describes the signatures of a collection of methods. In contrast to a class, no attributes (only constants) and no method bodies are specified. Interfaces are related to abstract classes and can also be in an inheritance relation to each other.
Type	is a basis data type such as <code>int</code> , a class or an interface.
Interface implementation	is a relation between an interface and a class, similar to inheritance. A class can implement any number of interfaces.
Association	is a binary relation between classes that is used for the realization of structural information. An association is described by an <i>association name</i> , a <i>role name</i> for each end, a <i>cardinality</i> , and information about the <i>directions of navigation</i> .
Cardinality	The cardinality (also called multiplicity) is given for each end of the association. It has the form “0..1”, “1” or “*” and describes whether an association in this direction is optional or mandatory, or allows for multiple bindings.

Figure 2.2. Definitions for class diagrams

Figure 2.3 shows a simple class diagram that consists of a class and a comment attached. The explanation in italics and the curved arrows do not belong to the diagram itself. They serve to describe the elements of the diagram.

Usually, the representation of a class is divided into three compartments. In the first compartment, the class name is given.

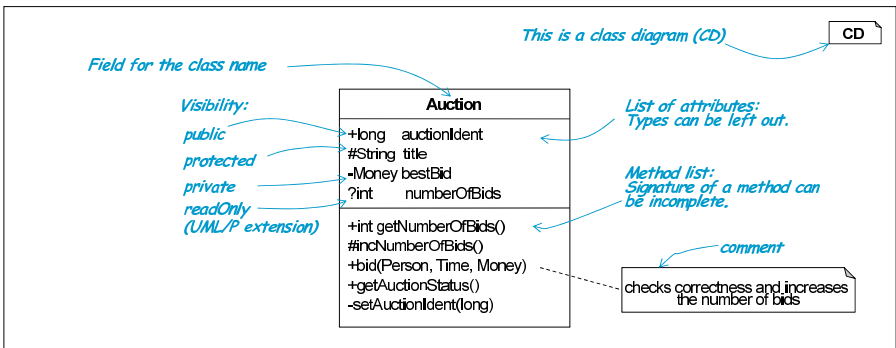


Figure 2.3. Class `Auction` in the class diagram

2.2.1 Attributes

The middle compartment of a class definition describes the list of attributes that are defined in this class. The information on attributes can be incomplete in many respects. First, an attribute can be indicated with or without its type. In the example in Fig. 2.3, the data types of all four attributes are shown. As we use Java as target language, the default notation for the UML “attribute: Type” has been replaced by the Java-compliant version “Type attribute”.

A number of *modifiers* are available for attributes, defining the attribute’s properties more precisely. As compact forms, UML provides “+” for `public`, “#” for `protected`, and “-” for `private` in order to describe the visibility of the attribute to foreign classes. “+” allows for general access, “#” for subclasses, and “-” allows access only within the defining class. The UML standard does not contain a fourth visibility declaration “?”, which is only offered by UML/P to mark an attribute as *readonly*. An attribute marked in this way is generally readable but can only be modified in subclasses and the class itself. This visibility thus has the same effect as `public` while reading and `protected` while modifying. It proves helpful in modeling in order to define access rights even more precisely.

Further modifiers offered by the programming language Java such as `static` and `final` for the description of static and nonmodifiable attributes can also be used in the class diagram. In combination, these modifiers serve for defining constants. However, constants are often omitted in class diagrams. An attribute marked `static` is also called a *class attribute* and can be marked alternatively by an underscore (see Fig. 2.4).

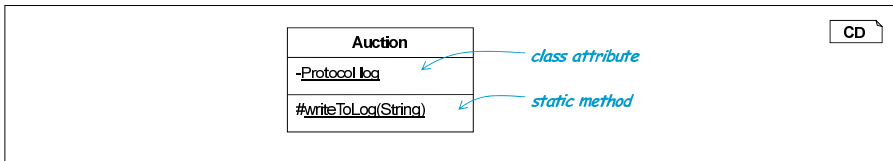


Figure 2.4. Class attribute and static method

UML provides *derived attributes* that are marked with “/” (see Fig. 2.5). In case of a derived attribute, its value can be calculated (“derived”) from other attributes of the same or other objects and associations. Usually, the calculation formula is defined in the form of a constraint `attr==...`. UML/P provides OCL that is introduced in Chap. 3 for this purpose.

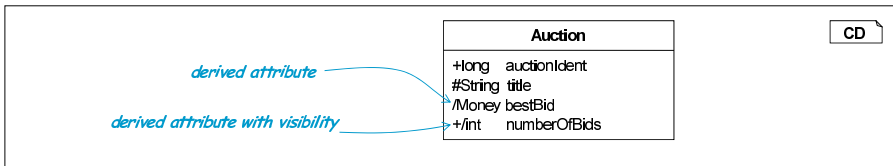


Figure 2.5. Derived attributes

2.2.2 Methods

In the third compartment of a class representation, methods are shown with names, signatures, and if any, modifiers for methods. Here, also the notation in line with Java `Type method (Parameter)` is used instead of the official UML notation `method (Parameter) : Type`. While attributes store the state of an object, methods serve for the execution of tasks and data calculation. For this purpose, they use data stored in attributes and call other methods of the same or other objects. Like Java, UML/P also provides methods with variable arity that, e.g., are given in the form `Type method (Type variable ...)`. The access rights for methods can be defined analogously to the visibilities for attributes with “+”, “#”, and “-”.

Further modifiers for methods are:

- `static` in order to make the method accessible even without an instantiated object
- `final` in order to make the method unchangeable for subclasses
- `abstract` in order to indicate that the method is not implemented in this class

Just like class attributes, UML prefers to outline static methods alternatively by underlining. Constructors are shown like static methods in the

form `class (arguments)` and are underlined. If a class contains an abstract method, the class itself is abstract. Then, the class cannot instantiate objects. In subclasses, however, abstract methods of a class can be implemented appropriately.

2.2.3 Inheritance

In order to structure classes into hierarchies, the inheritance relation can be used. If multiple classes with partly corresponding attributes or methods exist, these can be factorized in a common superclass. Figure 2.6 demonstrates this by means of similarities of several messages occurring in the auction system.

If two classes are in an inheritance relation, the *subclass* inherits its attributes and methods from the *superclass*. The subclass can extend the list of attributes and methods as well as *redefine* methods—as far as the modifiers of the superclass permit. At the same time, the subclass forms a *subtype* of the superclass that, according to the *substitution principle*, allows for the use of instances of the subclass where instances of the superclass are required.

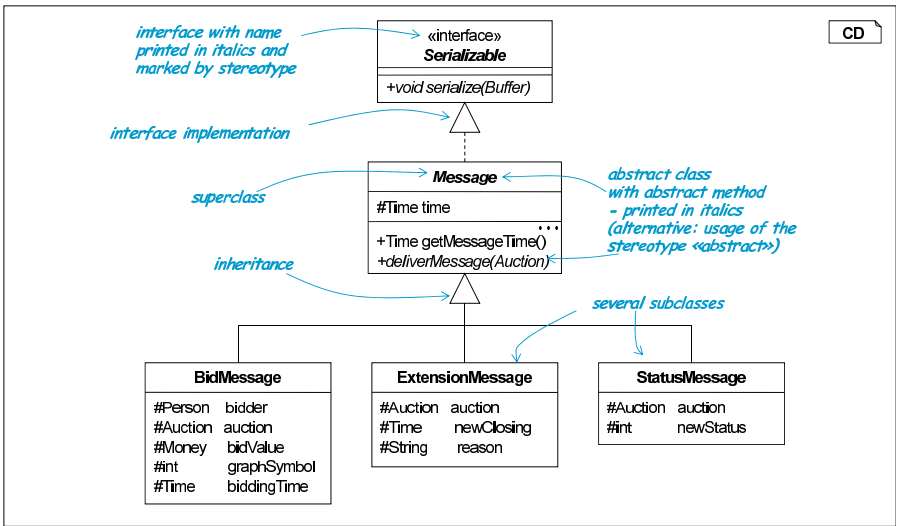


Figure 2.6. Inheritance and interface implementation

In Java, each class (except `Object`) inherits from exactly one superclass. However, a superclass can have many subclasses that in turn can have further subclasses. By using inheritance as a means of structuring, we define an *inheritance hierarchy*. A superclass can be regarded as a generalization of its subclasses, as its attributes and method signatures determine the similarities

of all subclasses. If in an inheritance hierarchy the code inheritance is less important than structuring, we speak of a *generalization hierarchy*. Especially for requirements elicitation and the architectural design, generalization plays a vital role to structure the system.

In object-oriented modeling, inheritance is an essential structuring mechanism, but deep inheritance hierarchies should be avoided as inheritance couples the classes and, thus, the code contained. To fully comprehend a subclass, its direct as well as all other superclasses have to be understood.

2.2.4 Interfaces

Java offers a special form of class, the *interface*. An interface consists of a collection of method signatures and constants and is applied especially for defining the usable interface between parts of the system, respectively its components. In Fig. 2.6, the interface `Serializable` is used in order to enforce a certain functionality from all classes that implement this interface.

An interface, like a class, is depicted by a rectangle but marked with the stereotype «interface». Objects can be instantiated neither directly from an interface nor from an abstract class. Instead, the given method signatures have to be realized in classes which implement the interface. Furthermore, interfaces can only contain constants but no attributes.

While in Java a class is only allowed to inherit from one superclass, it can implement any number of interfaces. An interface can extend other interfaces and, hence, have a *subtype relation* to these interfaces. In this case, the *subinterface* includes the method signatures defined by the *superinterface* in its own definition and extends these by additional methods. Figure 2.7 shows this by means of an extract of the Java class library.

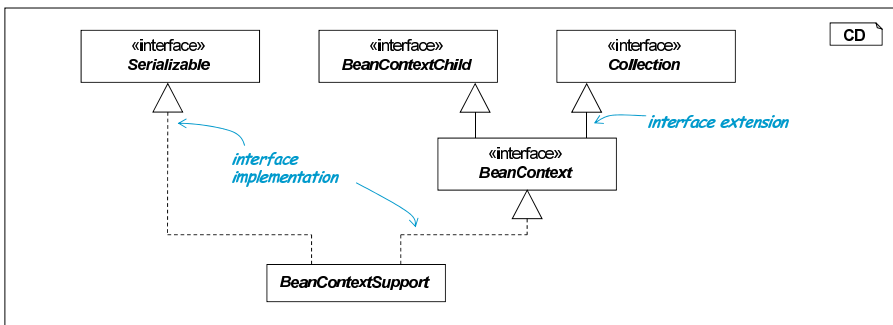


Figure 2.7. Interface implementation and extension

Technically, interfaces and classes as well as inheritance and interface implementation are similar concepts. That is why hereinafter, for the sake of simplification, the term *class* is often used as a generic term for classes and

interfaces as well as *inheritance* for the inheritance between classes, the implementation relation between interfaces and classes, and for the subtype relation between interfaces. This simplification is reasonable especially with regard to analysis and early design if the decision of whether a class can be instantiated, is abstract, or becomes an interface as not yet been made.

2.3 Associations

An association has the purpose of relating objects of two classes. Using associations, complex data structures can be formed and methods of neighboring objects can be called. Figure 2.8 describes a part of the auction system with three classes, two interfaces, and five associations in different forms.

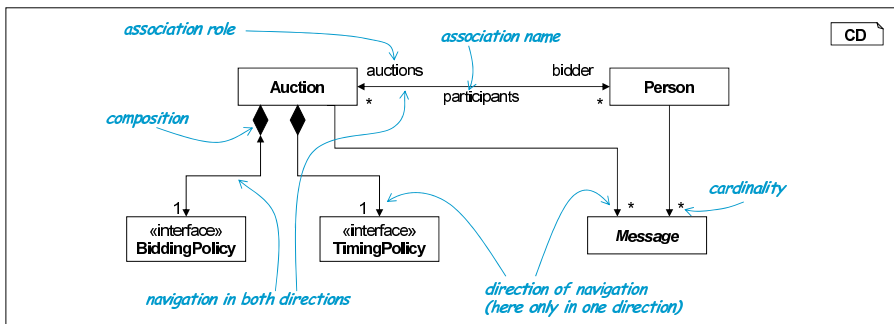


Figure 2.8. Class diagram with associations

Usually, an association has an *association name* and an *association role* for each of the two ends as well as information about the *cardinality* and a description of possible *navigation directions*. Details can also be omitted from the model if they are not important for the representation of the desired facts and if no ambiguity occurs. Association names, e.g., often serve only to distinguish associations, especially between the same classes. In case of a lack of the association or role name, there are standard rules for obtaining a surrogate name as outlined in the next section and further discussed in Sect. 3.3.8.

Just like a class, an association is a modeling concept in the class diagram. During the system's runtime, an association manifests through *links* between the connected objects. The number of links is limited by the association's cardinality. If an association is navigable in one direction, the implementation provides mechanisms to efficiently realize this navigability.

2.3.1 Roles

A role name is used to navigate to objects that are connected through an association or its links. In this way, one can access the auctions from an object of

the class `Person` by using the role name `auctions`. If no explicit role name is given, we use the name of the association or that of the target class as role name, provided that these unambiguously describe the intended navigation. In the example in Fig. 2.8, one can access the respective objects from an object of the class `Auction` with the names `biddingPolicy` and `messages`. According to the programming language used for the implementation and the cardinalities given in the class diagram, schematic conversions of the first letter in the name are made. Role names in UML/P always begin with a lower-case letter while class names begin with an upper-case letter.

If both ends of the association are connected to the same class, it is called a *reflexive association*. Reflexive associations enable the realization of a number of design patterns [GHJV94, BMR⁺96] such as a part-whole relationship. In a reflexive association, it is necessary to furnish at least one end with role names. By doing so, a distinction of the participating objects by their roles is possible.

Figure 2.10 shows a reflexive association *fellow* in which each observer is assigned to the bidder he can “observe.” Although that leads to a reflexive structure, the recursion depth is limited to 1 because bidders themselves are not observers and observers have direct connection to the bidders. This can be expressed through appropriate OCL conditions (see Chap. 3).

2.3.2 Navigation

During design and implementation activities, the navigation arrows of an association play an important role in a class diagram. The example in Fig. 2.8 describes the access of a `Person` object to its linked `Message` objects. Vice versa, (direct) access from a `Message` object to the `Persons` to which the object is linked is not possible. Therefore, the model allows the distribution of a message, e.g., using broadcasting, to several persons without duplication.

Basically, associations can be uni- or bidirectional. If no explicit arrow direction is given, a bidirectional association is assumed. Formally, the navigation possibilities are regarded as unspecified, and thus, no restriction is given in this situation.

If the fundamental navigability is modeled by the arrow, the role name determines how the association or the linked objects can be addressed. The modifiers `public`, `protected`, and `private` can be used for roles in order to correspondingly restrict the visibility of this navigation.

2.3.3 Cardinality

A cardinality can be indicated at each end of an association; For example, the association `participants` enables a person to participate in multiple auctions and allows various persons to place bids in the same auction, but only exactly one `TimingPolicy` is linked to each auction. The three cardinality specifications “*”, “1”, and “0..1” permit linking *any number of* objects,

exactly one, and *at most one* object respectively (see Fig. 2.8). More general cardinalities are of the form $m..n$ or $m..*$, and they could even be combined in the earlier UML 1.x versions (example 3..7, 9, 11..*). However, especially the three forms mentioned first can be directly implemented. Because of this, we abstain from discussing the general forms of cardinalities here. OCL invariants introduced in Chap. 3 allow for the description and methodical use of generalized cardinalities.

In the UML literature, a distinction is sometimes made between *cardinality* and *multiplicity*. In this case, cardinality designates the number of actual links of an association while multiplicity indicates the scope of potential cardinalities. The entity/relationship models do not make this distinction and consistently use the term cardinality.

2.3.4 Composition

Composition is a special form of association. It is indicated by a filled diamond at one end of the association. In a composition, the subobjects are strongly dependent on the *whole*. In the example in Fig. 2.8, *BiddingPolicy* and *TimingPolicy* are dependent on the *Auction* object in their lifecycle. This means that objects of these types are instantiated together with the *Auction* object and become obsolete at the end of the auction's lifecycle. As *BiddingPolicy* and *TimingPolicy* are interfaces, suitable objects which implement these interfaces are used instead.

An alternative form of representation expresses the nature of the composition of an association better by using graphic containedness instead of a diamond. Figure 2.9 shows two alternatives differing only in details. In class diagram (a), the association character of the composition is highlighted. It also describes navigation possibilities. In class diagram (b), navigation directions are not directly shown but both classes have a role name that describes how to access the components from the containing *Auction* object. The cardinality is indicated in the upper-right corner of the class. Representation (b) seems on the one hand more intuitive but is on the other hand less expressive. It is possible neither to clarify the backwards direction of the navigation nor to add further tags to the composition associations. The cardinality on the composition side is "1" by default but can be adjusted to "0..1", i.e., one object is assigned to at most one composite.

There are a number of interpretation variants regarding the possibility to exchange objects and for the lifecycle of dependent objects in a composite.² Thus, a precise definition of a composite's semantics should always be determined project specifically. This can, for instance, be done by stereotypes introduced in Sect. 2.5, which accompany supplementary project- or company-specific, informal explanations or by self-defined stereotypes.

² A detailed discussion on this topic is, e.g., provided by [HSB99] and [Bre01].

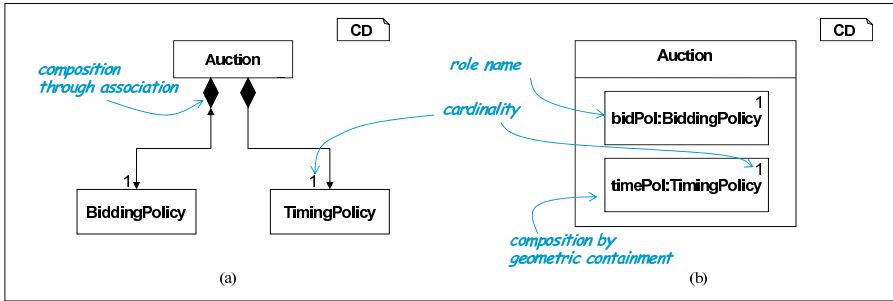


Figure 2.9. Alternative representations of composition

2.3.5 Derived Associations

Besides derived attributes, there are also derived associations in UML/P. They are also marked by a “/” in front of the association’s name. An association is considered derived if the set of its links can be calculated (“derived”) from other state elements. Other attributes and associations can be used for the calculation. In the example in Fig. 2.10, two associations are given, describing which persons are allowed to place bids in an auction and which persons can observe the behavior of a bidding colleague (“fellow”). The derived association `/observers` is calculated from these two associations. For this purpose, e.g., the OCL characterization given in Fig. 2.10 (see Chap. 3) can be used.

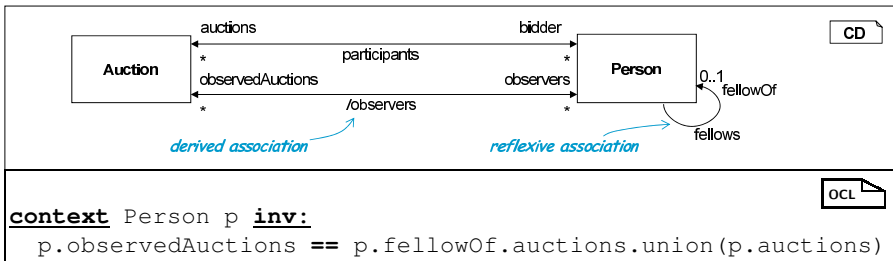


Figure 2.10. Derived association

2.3.6 Tags for Associations

The UML standard offers a variety of additional tags for associations that concretize the properties of associations more precisely. Figure 2.11 contains three tags that are interesting in this context. `{ordered}` indicates that an association with cardinality “*” allows ordered access. In the case shown, the order of the messages of an auction is relevant. `{frozen}` indicates that

the two associations to the policy objects can no longer be changed after the initialization of an `Auction` object is completed. In this way, the same policies are available throughout the whole lifetime of an `Auction` object. `{addOnly}` models that objects can only be added to the association and that removing them is prohibited. Thus, the model in Fig. 2.11 expresses that messages that have been sent in an auction cannot be withdrawn.

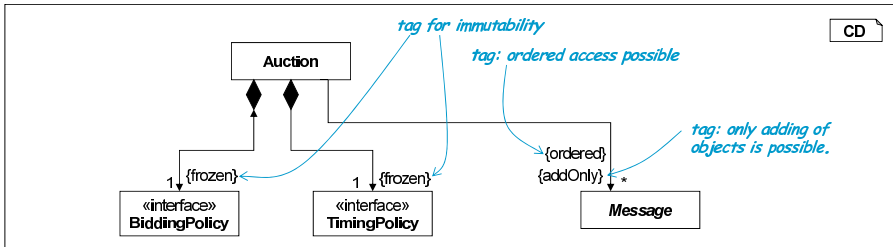


Figure 2.11. Tags for associations

An association that is tagged with `{ordered}` on one end certainly has to provide a mechanism that allows access according to this order. Associations with the tag `{ordered}` present a special case of qualified associations.

2.3.7 Qualified Associations

In their most general form, qualified associations provide an opportunity to select a single object from a set of assigned objects by means of a *qualifier*. Figure 2.12 shows several associations qualified in different ways. In addition, it is possible to qualify a composition or to use qualification at both ends of an association.

In the auction system, an object of the class `AllData` is used in order to store all currently loaded auctions and their participants. The qualified access via the auction identifier `auctionIdent` is understood as a mapping, and thus, the functionality of the interface `Map<long, Auction>` is provided, but this does not determine the form in which the qualified association is implemented. During code generation, a transformation into an alternative data structure can be applied or further functionality, e.g., from the `NavigableMap`, can be added.

As the key for the mapping, the auction identifiers already existing in the auction are used in the example. Analogously, persons can be selected via their name, which is of the type `String`. However, there is a significant difference between the cases where the qualifier is a type (in the example: `String`) or an attribute name of the associated class. While in the first case any object or value of the given type can be used as a key, only the actual

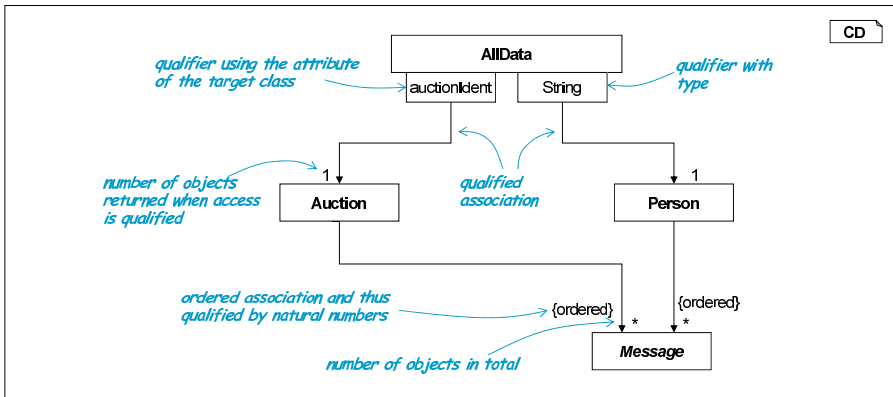


Figure 2.12. Qualified associations

attribute content is legitimate as a qualifier in the second case. This property can be formulated with OCL, which we will introduce in Chapter 3, as follows:

context AllData ad **inv:**
forall k **in** long:
 auction.containsKey(k) **implies**
 auction.get(k).auctionIdent == k

OCL

While in explicitly qualified associations the type of qualifier can be freely chosen, ordered associations use integer intervals starting with 0.³

Wherever an explicit qualifier is used, only one object is reached by the qualified access even if another cardinality is given. Usually, the target object is uniquely identified only in terms of the source object and the qualifier; For example, each auction can store a different message at index 0. The qualifier **allIdent** is only unique system-wide as it presents at the same time an unambiguous key of the target object. In case the qualifier is not set, access requests have to react appropriately, e.g., with a **null** pointer as in Java maps or an exception. In an ordered association (tag **{ordered}**), the qualifier remains implicit and is not indicated in a respective box at the end of the association.

2.4 View and Representation

A class diagram is mainly used to describe the structure and the relations necessary for a certain task. A complete list of all methods and attributes is often obstructive in this case. Instead, only those methods and attributes that are helpful for presenting a “story” should be illustrated. “Story” is a

³ As common in Java, indexing starts with 0.

metaphor that is deliberately used to indicate that a diagram has a focus highlighting significant and omitting unimportant information. Diagrams can, e.g., model different parts of the system or particular system functions.

Hence, a class diagram often represents an incomplete *view* of the whole system. Some classes or associations can be missing. Within classes, attributes and methods can be omitted or presented incompletely. For example, the argument list and the return type of a method can be omitted.

Unfortunately, in general it is not evident from a UML diagram whether the information contained therein is complete or not. Therefore, “©” has been taken over from [FPR01] for the display of complete information. It supplements the representation indicator “...”already offered by UML to mark incomplete information.

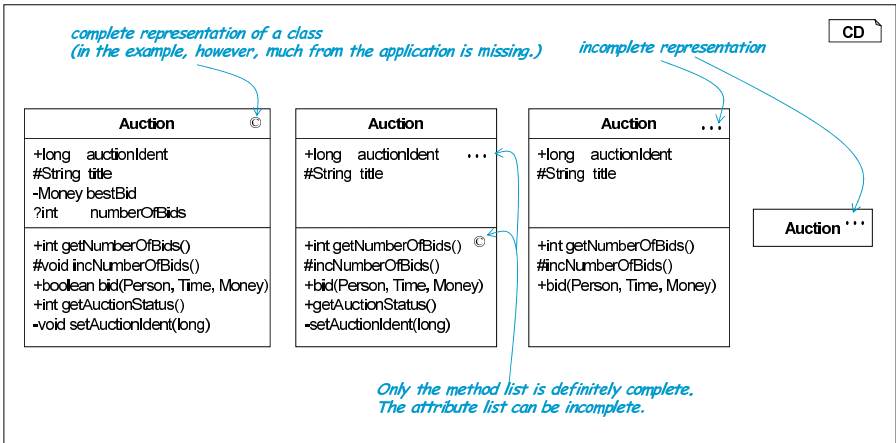


Figure 2.13. Complete class representation

Figure 2.13 shows how the two indicators “...” and “©” can be used. The indicators “©” and “...” do not have any effect on the class itself but on their representation within the class diagram. A “©” in the class name shows that the attribute as well as the method list is complete. In contrast, the incompleteness indicator “...” means that the presentation *can* be incomplete. Due to the dualism between associations and attributes later discussed, it is implied that all associations that can be navigated from this class are also modeled when the attribute list is marked as incomplete.

Both indicators can also be applied to the list of attributes and methods individually. The incompleteness indicator “...” acts as default when no indicators are given. This corresponds to the usual assumption that a diagram presents an abstraction of the system.

To explain the use of these indicators precisely, the three model levels illustrated in Fig. 2.14 are to be distinguished: the system itself, the complete

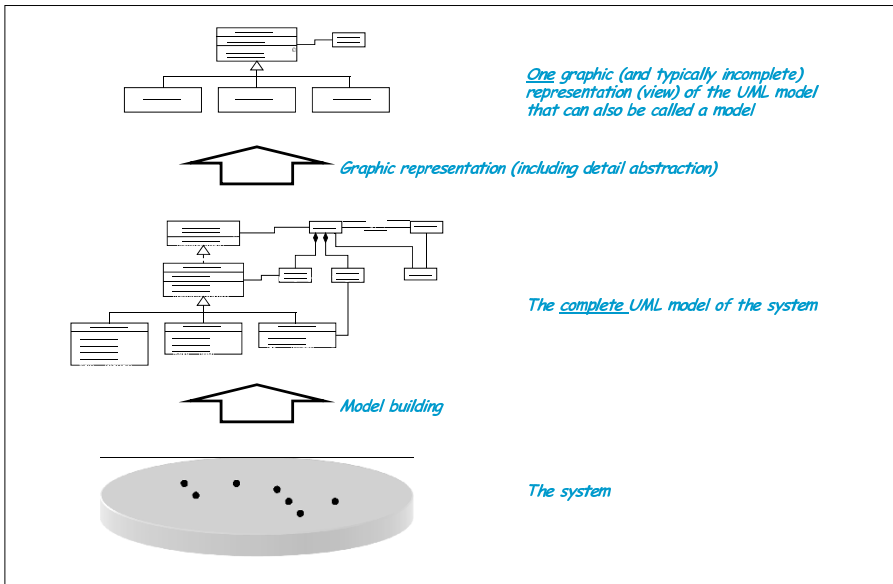


Figure 2.14. Illustration of the three model levels

UML model of the system, and a graphic, incomplete *representation* of the model, e.g., on screen. In most cases, the representation is also called a *view* but is itself a model. Both indicators “@” and “...” describe a relation between the view and the incomplete UML model on which the view is based. As Fig. 2.13 shows, there are a number of different views for the same class. The two marks are thus called *representation indicators*.

If a software system reaches a certain size, a complete class diagram can seem quite overloaded. Due to the many details, the story is concealed rather than presented. Because of this, in software development it is reasonable to work with multiple smaller class diagrams. Necessarily, class diagrams that each show an extract of the system have overlaps. Through these overlaps, the correlation between the individual models is established. Figure 2.15 shows two extracts of the auction system in which the class Auction is represented in different forms. Here, the indicator “...” allows one to explicitly describe that only the information necessary for the respective story is illustrated.

Through *fusion*, a complete class diagram can be obtained from a collection of single class diagrams. As discussed in Sect. 1.4.4, some tools always use a complete class diagram internally as their model and the user gets extracts in the form of views. A fusion of class diagrams is basically a unification of all the class, association, and inheritance relations, whereas in the case of a repeatedly occurring class, attribute and method lists are also consolidated. There are of course a number of consistency conditions to be fol-

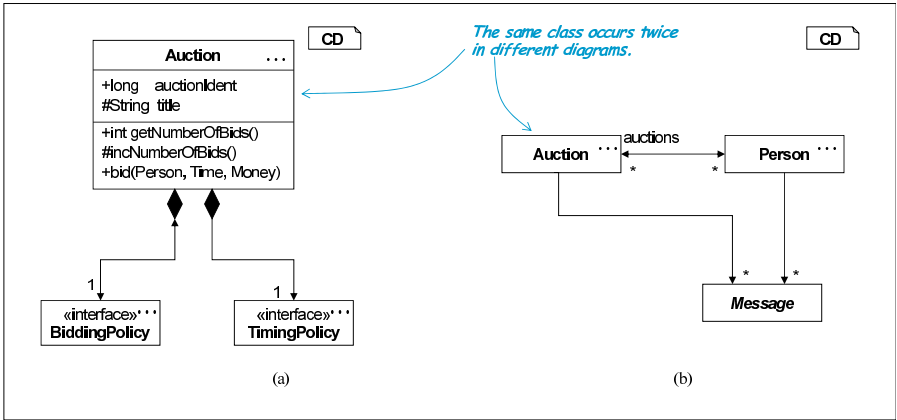


Figure 2.15. Overlapping of class diagrams

lowed. These include conforming types for attributes and methods, compatible navigation roles and multiplicity specifications for associations, as well as prevention of cyclical inheritance relations.

2.5 Stereotypes and Tags

Although UML is designed as a graphic language for the implementation, it has a lot in common with natural languages. UML has a core structure that corresponds to a grammar, and its sentences are built in the form of diagrams. Similar to natural languages, there are mechanisms to extend and adjust the language vocabulary according to the respective necessary requirements. These mechanisms form the basis for project- and company-specific dialects or profiles of UML.

However, the introduction of a new class already extends the available vocabulary, as we can then use this class in other places. From this perspective, programming is a steady expansion of the vocabulary at hand in the system. But while in a programming language the introduction of a new control structure is not possible, UML allows, in a restricted form, the introduction of new kinds of model elements by offering *stereotypes* and *tags* by means of which existing model elements can be specialized and adjusted (see Fig. 2.16).

Without providing an explicit mechanism, UML allows on the one hand for modification of the syntactic appearance through restriction or expansion, but on the other hand also to change the semantics of the language. As UML is designed as a “universal” language, its forms of use can impose a certain bandwidth for its semantics. This so-called *semantic variability* [Grö10] enables project-specific adjustment of the semantics and tools. “Semantic variation points” cannot be described in standard UML itself. This is

why, in [GRR10, CGR09], an independent mechanism on the basis of feature diagrams is defined and can be used for profile creation. The generally possible adaptations go far beyond the concept of stereotypes and tags introduced here.

<p>Stereotype. A stereotype classifies model elements such as classes or attributes. Through a stereotype, the meaning of the model element is specialized and can thus, e.g., be treated more specifically in code generation. A stereotype can have a set of tags.</p> <p>Tag. A tag describes a property of a model element. A tag is denoted as a pair consisting of a <i>keyword</i> and <i>value</i>. Several such pairs can be combined in a comma-separated list.</p> <p>Model elements are the (fundamental) parts of UML diagrams. For instance, the class diagram has classes, interfaces, attributes, methods, inheritance relations, and associations as model elements. Tags and stereotypes can be applied to model elements, but they themselves are not model elements.</p>

Figure 2.16. Definition: tag and stereotype

In the previous examples in this chapter, *stereotypes*, *tags*,⁴ and related mechanisms have already been used occasionally. In the class shown in Fig. 2.3, the visibility markers “+”, “#”, “?”, and “-” were introduced. Figure 2.13 shows the two representation indicators “@” and “...” referring to the representation of a view of the model. Figure 2.6 shows the stereotype «interface» that marks a “special” class, namely an interface. The tags {ordered}, {frozen}, and {addOnly} exclusively serve to mark the ends of associations, as shown in the example in Fig. 2.11.

2.5.1 Stereotypes

Figure 2.17 exhibits three kinds of stereotypes. While the stereotype «interface» is provided by default by UML, the two stereotypes on the right «JavaBean» and «Message» are to be defined in the project, tool or framework itself.

Stereotypes are normally indicated in French quotation marks (guillemots) with reversed tips. In principle, each UML model element can be equipped with one or several stereotypes. However, stereotypes are often used in order to assign special properties to classes.

The stereotype «interface» tags an interface, which is regarded as a special form of class. The stereotype «JavaBean» acts as an indicator for the fact that the class tagged provides the functionality required by JavaBeans. The stereotype «Message» is used in the auction project in order to record in

⁴ In the UML definition, the terms *tagged values* and *properties* are used, being summarized, among others, as *tags* in [Bal99].

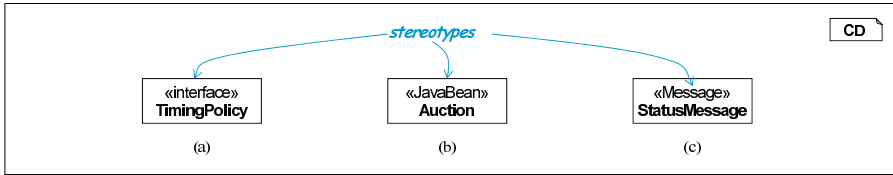


Figure 2.17. Types of stereotypes for classes

compact form that the class tagged is a subclass of `Message` and thus serves for the transfer of information.

Hence, there are a multitude of application possibilities for stereotypes. They can classify model elements, e.g., in order to specify additional properties or functionalities or to impose restrictions. The UML standard offers a metamodel-based and a tabular approach for informal definition of stereotypes. Depending on the stereotype's intention, restrictive conditions can also be formulated more precisely, or mechanisms for a specific code generation can be indicated. The following list shows some application possibilities for stereotypes:

- A stereotype describes syntactic properties of a model element by demanding additional properties or specializing already existing properties.
- A stereotype can describe the representation of a model provided to the user. The indicator “©” can be considered such a special form.
- A stereotype can describe application-specific requirements. The «persistent» stereotype, for example, can specify that objects of this class are persistently stored, although it is not explained how this storage is to happen.
- A stereotype can describe a methodical relation between model elements. For instance, the stereotype «refine» in the UML standard is designed for this purpose.
- A stereotype can reflect the modeler's intention describing how a programmer should use a certain model element. A class can, e.g., be tagged as «adaptive» in order to imply that this class is well suited to extension. Such stereotypes are especially suited for frameworks (see [FPR01]). With stereotypes of the form «Wrapper», the role of a class in a design pattern can be documented.

Of course, there are a number of overlaps between the mentioned and further application options for stereotypes.

To allow the developer to specify properties in a more detailed form, a stereotype can be equipped with a variety of tags. Then, the application of a stereotype in a model element implies that its assigned tags are also defined on the model element.

2.5.2 Tags

Figure 2.18 shows a test class of the auction system marked by a corresponding stereotype. Information on the indicated test and its execution is stored in the form of tags.

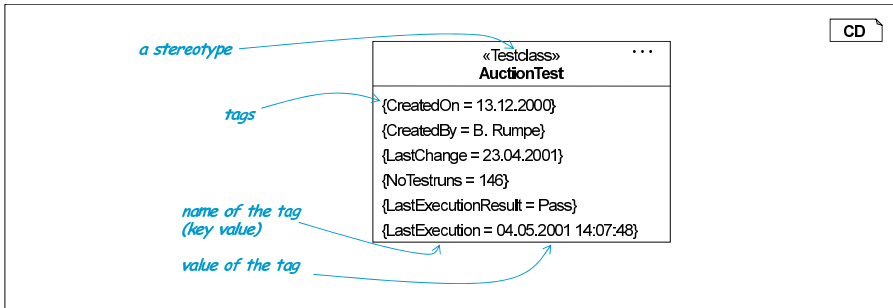


Figure 2.18. Tags applied on a test class

Tags can be attached to basically each model element. In addition, tags, as shown in Fig. 2.18, can be bound to a stereotype and thus applied together with the stereotype to model elements. In the example, the existence of the last three tags is demanded by the use of the stereotype **«Testclass»**, as in the auction project these three tags are assigned to the stereotype.

A tag is usually denoted in the form **{name = value}**. Principally, strings and numbers are accepted as values. An explicit typing of the tag values would be desirable, but up to now it is supported by neither the UML language standard nor the tools. If the value is not relevant or it is the Boolean value **true**, it can also be omitted; For example, **{are_Tests_OK = true}** and **{are_Tests_OK}** are alternative representations. The UML standard [OMG10a] by default offers tags for associations such as **{ordered}** but also allows one to define new, situation-specific tags.

Even if a tag is added to a class, it considerably differs from an attribute. A tag assigns a property to the model element, while an attribute has an independent value in each instance of the class. Attributes appear at the runtime of a system while tags do not exist there. Tags, however, can have effects on the system if they influence the properties of the model element with regard to its implementation. Among others, tags qualify for the presentation of the following properties:

- The initial value of an attribute can be specified.
- Figure 2.18 shows how project information is presented in the form of tags. This includes the name of the author of a class, the date of the most recent change, the current version number, and suchlike.
- Informal comments can be indicated by tags.

- Designated techniques for data storage or transfer via a network can be filed, e.g., in the model.
- For graphic representation in a tool, a suitable graphic symbol can be assigned to a model element whose file name is stored in the tag.

2.5.3 Introduction of New Elements

The variety of application options for stereotypes and tags makes it nearly impossible to describe the meaning of such elements directly within UML. Hence, when defining a tag or stereotype, usually an informal description is given. By doing so, not only the concrete appearance but especially the intention and the application domains are described.

The most important reason for the introduction of a stereotype is the methodical, tool-supported handling during software development. As there are many different application possibilities for stereotypes, ranging from controlling code generation to documentation of unfinished pieces in the model, one can in general say little about the meaning of stereotypes. Therefore, Table 2.19 only gives a general notation proposal for the definition of stereotypes that can be adjusted and extended by appropriate tools for concrete tasks.

Stereotype «Name»	
Model element	To which element is the stereotype applied? If applicable, pictures can illustrate the concrete form of presentation.
Motivation	What is the purpose of the stereotype? Why is it necessary? How does it support the developer?
Glossary	Concept formation—as far as necessary.
Usage condition	When can the stereotype be applied?
Effect	What is the effect?
Example(s)	Illustrations through application examples, mostly underpinned with diagrams.
Pitfalls	Which special problems can occur?
See also	What other model elements are similar or supplement the stereotype defined here?
Tags	Which tags (name and type) are associated with the stereotype? Which ones are optional? Are there default values? What is the meaning of a tag (unless defined in a separate table)?

(continued on the next page)

(continues Table 2.19.: Stereotype «Name»)

Extendable to	The stereotype can often be applied to a superior model element or a whole diagram in order to be applied element-wise on all subelements.
---------------	--

Table 2.19. Stereotype «Name»

The definition of a stereotype follows the general form of design patterns [GHJV94], recipes [FPR01], and process patterns [Amb98] by discussing motivation, requirements, application form, and effects on an informal basis. However, the template should not be regarded as rigid but, as needed, should be extended as appropriate or shortened by removing unnecessary sections. In principle, the same template can be used for tags.

Tags, however, are basically more easily structured and easier to understand, so that such a detailed template often seems unnecessary.

UML offers a third form of adaptations for model elements. *Constraints* are an instrument for the detailed specification of properties. As constraint languages, OCL introduced in Chap. 3 or informal text is suggested. A constraint is generally given in the form {*constraint*}. The UML standard by default provides some constraints. These include the already known constraint {*ordered*} for associations that, however, can also be defined as a tag with Boolean type. This example illustrates in particular that the differences between constraints, tags, and stereotypes cannot always be clarified precisely. It also makes little sense to introduce a stereotype consisting of exactly one tag, as this tag could also be attached directly to a model element. When introducing new stereotypes, tags or constraints, certain creative freedom is given that can be used by the modeler in order to design a suitable model.

Modeling with UML

Language, Concepts, Methods

Rumpe, B.

2016, XIV, 281 p. 175 illus., 3 illus. in color., Hardcover

ISBN: 978-3-319-33932-0