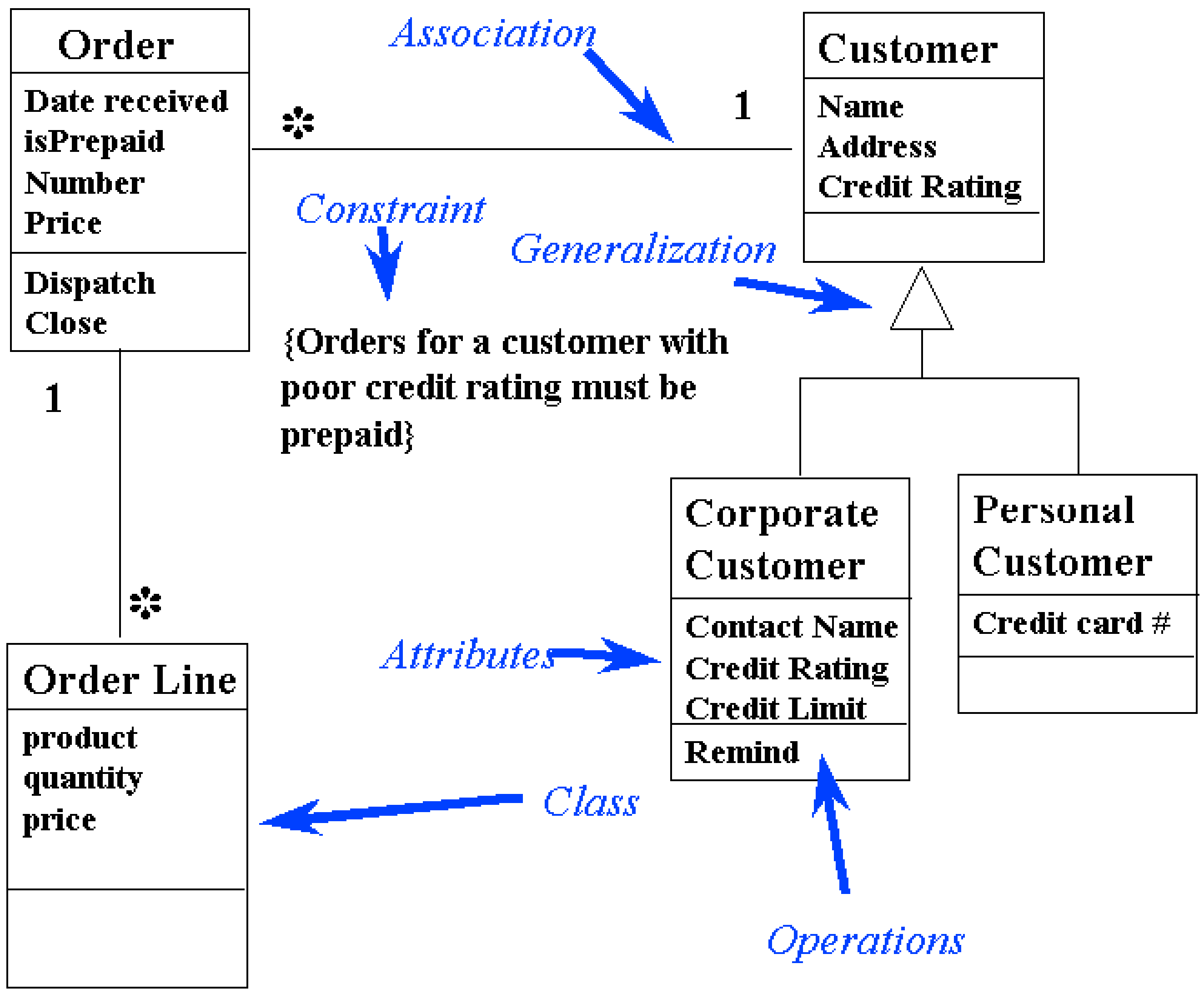
# Class Diagrams

The class diagram is a central modeling technique that runs through nearly all object-oriented methods. This diagram describes the types of objects in the system and various kinds of static relationships which exist between them. There are three principal kinds of relationships which are important: associations (a customer may rent a number of videos), subtypes (a nurse is a kind of person) and aggregation (an engine is part of an aircraft). The various OO methods all use different (and often conflicting) terminology for these concepts, this is extremely frustrating but inevitable: OO languages are just as inconsiderate. It is in this area that the UML will bring some of its greatest benefits in simplifying these different diagrams. In this section I will use the UML terms as my main terminology, and relate to other terms as I go along.



## Perspectives

Before I begin describing class diagrams I have to bring out an important subtlety in the way that people use class diagrams. This is a subtlety that is usually undocumented, but has an important impact on the way you should interpret a diagram, for it really concerns what it is you are describing with a model. Following the lead of [Cook and Daniels] I say that there are three perspectives you can use in drawing class diagrams (or indeed any model, but it is most noticeable in class diagrams).

- *Conceptual:* In this case you are drawing a diagram that represents the concepts in the domain under study. These concepts will naturally relate to the classes that implement them, but it is often not a direct mapping. Indeed the model is drawn with little or no regard for the software that might implement it, and is generally language independent. ([Cook and Daniels] call this the essential perspective, I use conceptual as the usage has been around for a long time)
- *Specification:* Now we are looking at software, but we are looking at the interfaces of the software, not the implementation. We are thus looking at types rather than classes. Object-oriented development puts a great emphasis on the difference between type and class, but this is often overlooked in practice. In my view it is important to separate interface (type) and implementation (class). Most OO languages do not do it and methods, influenced by that, have followed suit. This is changing (Java and CORBA will have some influence here) but not quickly enough. Types represent an interface which may have many implementations due to implementation environment, performance characteristics, or vendor. The distinction can be very important in a number of design techniques based on delegation, hence the discussion in [Gang of Four]
- *Implementation:* In this view we really do have classes and we are laying the implementation bare. This is probably the most often used perspective, but in many ways the specification perspective is often a better one to take.

Understanding the perspective is crucial to both drawing and reading class diagrams. As I talk about the technique further I will stress how each element of this technique depends heavily on the perspective. When you are drawing a diagram, draw it from a single clear perspective, when you read a diagram make sure you know which perspective the drawer drew it in. That knowledge is essential if you are to interpret the diagram properly. Unfortunately the lines between the perspectives are not sharp, and most modelers do not take care to get their perspective sorted out when they are drawing.

## Associations, attributes and aggregation

Associations represent relationships between instances of types (a person works for a company, a company has a number of offices…). The interpretation of them varies with the perspective. Conceptually they represent conceptual relationships between the types involved. In specification these are responsibilities for knowing, and will be made explicit by access and update operations. This may mean that a pointer exists between order and customer, but that is hidden by encapsulation. A more implementation interpretation implies the presence of a pointer. Thus it is essential to know what perspective is used to build a model in order to interpret it correctly.

Associations may be bi-directional (can be navigated in either direction) or uni-directional (can be navigated in one direction only). Conceptually all associations can be thought of as bi-directional, but uni-directional associations are important for specification and implementation models. For specification models bi-directional associations give more flexibility in navigation but incur greater coupling. In implementation models a bi-directional association implies coupled sets of pointers, which many designers find difficult to deal with.

One of the key aspects of associations is the cardinality of an association (sometimes called multiplicity). This specifies how many companies a person may work for, how many children a mother can have, etc. This corresponds to the notion of mandatory, optional, 1-many, many-many relationships in the Entity-Relationship approach. The UML has standard the mass of different approaches that existed here.

## Generalization

Subtyping is the most obvious addition to ER diagrams for use in OO. It has an immediate correspondence to inheritance in OO programming. However the object-oriented community should not forget that subtyping has been around in data modeling long before the object cavalry rode over the horizon.

A typical example of subtyping is to consider personal and corporate customers of a business. They have differences but also many similarities. The similarities can be placed in a general customer class with personal and corporate customer as subtypes.

Again this phenomenon has different interpretations at the different levels of modeling. Conceptually we can say that corporate customer is a subtype of customer if all instances of corporate customer are also, by definition, instances of customer. From a specification model we would say that the interface of corporate customer must conform to the interface of customer. That is an instance of corporate customer may be used in any situation where a customer is used, and the caller need not be aware that a subtype is actually present (the principle of substitutability). The corporate customer may respond to certain commands differently than another customer (polymorphism) but the caller should not need to worry about the difference.

Inheritance and subclassing in OO languages is an implementation approach. It says that the subclass inherits the data and operations of the superclass. It has a lot in common with subtyping, but there are important differences. Subclassing is only one way of implementing subtyping (see [Odell pragmatics] or [Fowler]). Subclassing may also be used without subtyping and most authors frown upon this practice. Newer languages and standards increasingly try to emphasize the difference between interface-inheritance (subtyping) and implementation-inheritance (subclassing).

## Constraint Rules

One area of OO analysis and design that has gained more attention in recent years is that of rules. The general notion is to apply the ideas of AI on rule based systems into OO modeling. Actually some of these have been around for a while in the object community. Eiffel has long had support for assertions as part of its principle of Design by Contract, which been sadly neglected in many OO methods. In the structural view the principal assertion is the constraint. This is a logical expression about a type which must always be true. Cardinality express some constraints, but not all. UML uses the brace {} notation to show constraints on the structural model.

## When to Use Them

Class diagrams are the backbone of nearly all OO methods so you will find yourself using them all the time. The trouble is that they are so rich that they can be overwhelming to use. Here are a few tips:

- Don't try to use all the various notations on offer. Start with the simple stuff: classes, associations, attributes, and generalization. Introduce other notations only when you need them.
- Sort out which perspective you are drawing the models from. If you are in analysis draw conceptual models. When working with software concentrate on specification models. Draw implementation models only when you are illustrating a particular implementation technique
- Don't draw models for everything, concentrate on the key areas. It is better to have a few diagrams that you use and keep up to date than many forgotten, out-of-date models.

The biggest danger with class diagrams is that you can get bogged down in implementation details far too early. To combat this use the conceptual or specification perspective. If you get these problems you may well find CRC cards to be extremely useful.

### Where to Find Out More

At the moment my advice depends on whether you prefer an implementation or a conceptual perspective. For an implementation perspective try [Booch], for a conceptual perspective try [Odell foundations]. Once you have read your choice read the other one. Both perspectives are important. After that any OO book will add some interesting insights. I particularly like [Cook and Daniels] for its treatment of perspectives and the formality that they introduce.

---

|previous|contents|next|