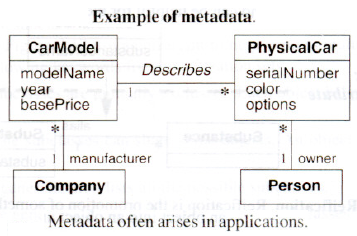
**Metadata:**

* *Metadata* is data that describes other data.

E.g. A class definition is metadata. Models are inherently metadata, since they describe the things being modeled (rather than *being* the things). Many real-world applications have metadata, such as parts catalogs, blueprints, and dictionaries. Computer-language implementations also use metadata heavily.

Figure shows an example of metadata and data.

A car model has a model name, year, base price, and a manufacturer. Some examples of car models are a 1969 Ford Mustang and a 1975 Volkswagen Rabbit.

A physical car has a serial number, color, options, and an owner. As an example of physical cars, John Doe may own a blue Ford with serial number 1*FABP*and a red Volkswagen with serial number 7E81 *F.*

A car model describes many physical cars and holds common data. A car model is metadata relative to a physical car, which is data.

* Classes can also be considered as objects, but classes are meta-objects and not real-world objects. Class descriptor objects have features, and they in turn have their own classes, which are called *meta-classes.* Treating everything as an object provides a more *uniform implementation and greater functionality* for solving complex problems.
* *Languages vary in their accessibility for metadata.* Some languages, like Lisp and Smalltalk, let metadata be inspected and altered by programs at run time. In contrast, languages like C++ and Java dea1with metadata at compile time but do not make the metadata explicitly available at run time.

**Reification:**

*Reification* is the *promotion* of something that is not an object into an object. Reification is a helpful technique for meta applications because it lets *shifting the level of abstraction*. On occasion it is useful to *promote attributes, methods, constraints and control information into objects* so they can be described and manipulated as data.

E.g.1. Consider a *database manager*. A developer could write code for each application so that it can read and write from files. Instead, for many applications, it is a better idea to reify the notion of data services and use a database manager. A database manager has abstract functionality that provides a general-purpose solution to accessing data reliably and quickly for multiple users.

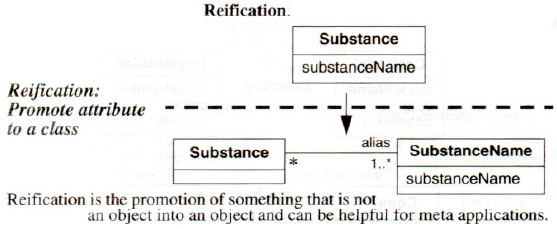
E.g.2. Consider *state-transition diagrams* (Next two chapters). A state-transition diagram can be used to specify control and then implement it by writing the corresponding code. Alternatively, a metamodel can be prepared and a state-transition model can be stored as data. A general-purpose interpreter reads the contents of the metamodel and executes the intent.

Figure promotes the *substanceName* attribute to a class to capture the many-to-many relationship between *Substance* and *SubstanceName.* A chemical substance may have multiple aliases. E.g. propylene may be referred to as *propylene* and C3*H*6*.* Also, an alias may pertain to multiple chemical substances. Various mixtures of ethylene glycol and automotive additives may have the alias of *antifreeze.*

**Constraints:**

A *constraint* is a boolean condition involving model elements, such as objects, classes, attributes, links, associations, and generalization sets. A constraint restricts the values that elements can assume. Constraints can be expressed with natural language or a formal language such as the Object Constraint Language (OCL) [Warmer-99].

***Constraints on Objects:***

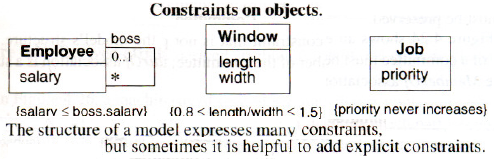
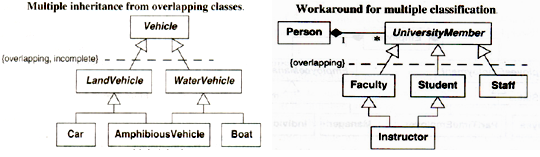


Figure above shows several examples of constraints. No employee's salary can exceed the salary of the employee's boss (a constraint between two things at the same time). No window can have an aspect ratio (length/width) of less than 0.8 or greater than 1.5 (a constraint between attributes of a single object). The priority of a job may not increase (constraint on the same object over time). Simple constraints may be placed in class models.

***Constraints on Generalization Sets:***

Class models capture many constraints through their very structure.

E.g. The semantics of generalization imply certain structural constraints. With single inheritance the subclasses are mutually exclusive. Furthermore, each instance of an abstract superclass corresponds to exactly one subclass instance. Each instance of a concrete superclass corresponds to at most one subclass instance.



Above figures use a constraint to help express multiple inheritance.

*The UML defines the following* ***keywords*** *for generalization sets*.

**Disjoint**: The subclasses are mutually exclusive. Each object belongs to exactly one of the subclasses.

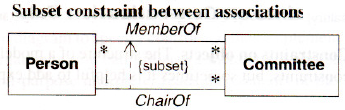
**Overlapping:** The subclasses can share some objects. An object may belong to more than one subclass.

**Complete:** The generalization lists all the possible subclasses.

**Incomplete:** The generalization may be missing some subclasses.

***Constraints on Links:***

* **Multiplicity** is a constraint on the cardinality of a set. Multiplicity for an association restricts the number of objects related to a given object. Multiplicity for an attribute specifies the number of values that are possible for each instantiation of an attribute.
* **Qualification** also constrains an association. A qualifier attribute does not merely describe
* the links of an association but is also significant in resolving the "many" objects at an
* association end.
* An **association class** implies a constraint. An association class is a class in every right; for example, it can have attributes and operations, participate in associations, and participate in generalizations. But an association class has a constraint that an ordinary class does not; it derives identity from instances of the related classes.
* An **ordinary association** presl'mes no particular order on the objects of a "many" end.
* The **constraint *{ordered}***indicates that the elements of a "many" association end have an explicit order that must be preserved.

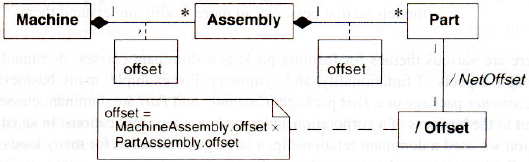
Figure shows an explicit constraint that is not part of the model's structure. The chair of a committee must be a member of the committee; the *ChairOf* association is a subset of the *MemberOf* association.

***Use of Constraints:***

It is better to express constraints in a declarative manner. Declaration expresses a constraint's *intent*, without supposing an implementation. Constraints are converted to procedural form before implementing them in a programming language. This conversion is usually straightforward.

Constraints provide *one criterion for measuring the quality of a class model*; a "good" class model captures many constraints through its structure. It often requires several iterations to get the structure of a model right from the perspective of constraints. Every constraint need not be enforced with the model’s structure but only the important ones.

The *UML has two alternative notations* for constraints.

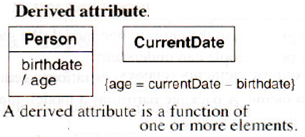
A constraint can be *delimited with braces* or it can be *placed in a "dog-eared" comment box* [Figure].

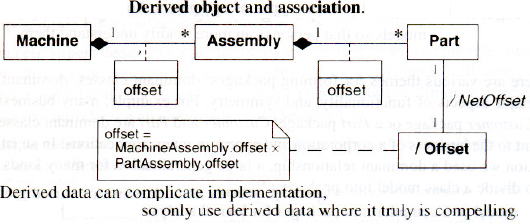
Either way better to position the constraints near the affected elements. Dashed lines can be used to connect constrained elements. A dashed arrow can connect a constrained element to the element on which it depends.

**Derived Data:**

A *derived element* is a function of one or more elements, which in turn may be derived. A derived element is redundant, because the other elements completely determine it. Ultimately, the derivation tree terminates with base elements. Classes, associations, and attributes may be derived. The notation for a derived element is a *slash* in front of the element name.

Constraint that determines the derivation must also be shown.

Figure shows a derived attribute. Age can be derived from *birthdate* and the *currentdate.*

In figure, a machine consists of several assemblies that in turn consist of parts. An assembly has a geometrical offset with respect to machine coordinates; each part has an offset with respect to assembly coordinates. We can define a coordinate system for each part that is derived from machine coordinates, assembly offset, and part offset. This coordinate system can be represented as a derived class called *Offset* related to each part by a derived association called *NetOffset.*

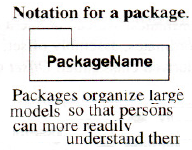
It is useful to distinguish operations with side effects from those that merely compute a functional value without modifying any objects. The latter kind of operation is called a *query.*

Queries can be regarded with no arguments except the target object as derived attributes. E.g. Width of a box can be computed from the positions of its sides. In many cases, an object has a set of attributes with interrelated values, of which only a fixed number of values can be chosen independently. A class model should generally distinguish independent *base attributes* from dependent *derived attributes.* The choice of base attributes is arbitrary but should be made to avoid over specifying the state of the object. Some developers tend to include many derived elements. Generally, this is not helpful and clutters a model. Derived elements should be included only when they are important application concepts or substantially ease implementation. It can be quite difficult to keep derived elements consistent with the base data, so only use derived elements for implementation where they are clearly compelling.

**Packages:**

A class model can fit on a single page for many small and medium-sized problems. However, it is often difficult to grasp the entirety of a large model. It is recommended to partition large models so that people can understand them.

A *package* is a group of elements (classes, associations, generalizations, and lesser packages) with a *common theme*. A package partitions a model, making it easier to understand and manage. Large applications may require *several tiers of packages*. Packages *form a tree* with *increasing abstraction* toward the root, which is the application, the top-level package.

Figure shows the notation for a package i.e. a *box with a tab*. The purpose of the tab is to suggest the enclosed contents, like a *tabbed folder*.

There are *various themes* for forming packages:

dominant classes, dominant relationships, major aspects of functionality, and symmetry. E.g. many business systems have a *Customer* package or a *Part* package; *Customer* and *Part* are dominant classes that are important to the business of a corporation and appear in many applications. In an engineering application we used a dominant relationship, a large generalization for many kinds of equipment, to divide a class model into packages. Equipment was the focus of the model, and the attributes and relationships varied greatly across types of equipment. The class model of a compiler can be divided into packages for lexical analysis, parsing, semantic analysis, code generation, and optimization. Once some packages have been established, *symmetry may suggest additional packages*.

***Tips for devising packages***:

* **Carefully delineate each package's scope**: The precise boundaries of a package are a matter of judgment. Like other aspects of modeling, defining the scope of a package requires planning and organization. Make sure that class and association names are unique within each package, and use consistent names across packages as much as possible.
* **Define each class in a single package:** The defining package should show the class name, attributes, and operations. Other packages that refer to a class can use a class icon, a box that contains only the class name. This convention makes it easier to read class models, because a class is prominent in its defining package. Readers are not distracted by definitions that may be inconsistent or misled by forgetting a prior class definition. This convention also makes it easier to develop packages concurrently.
* **Make packages cohesive**: Associations and generalizations should normally appear in a single package, but classes can appear in multiple packages, helping to bind them. Try to limit appearances of classes in multiple packages. Typically no more than 20-30% of classes should appear in multiple packages.

**Practical Tips:**

Here are additional tips for constructing class models.

**Enumerations**. When constructing a model, enumerations and their values should be declared because they often occur and are important to users. Do not create unnecessary generalizations for attributes that are enumerations. Only specialize a class when the subclasses have distinct attributes, operations, or associations.

**Class-scoped(static) attributes:** It is acceptable to use an attribute with class scope to hold the extent of a class. Otherwise, attributes with class scope should be avoided because they can lead to an inferior model. Model can be improved by explicitly modeling groups and assigning attributes to them.

**N-ary associations:** Try to avoid n-ary associations. Most n-ary associations can be decomposed into binary associations.

**Concrete superclasses**: As a matter of style, it is best to avoid concrete superclasses. Then, abstract and concrete classes are readily apparent at a glance - all superclasses are abstract and all leaf subclasses are concrete. Concrete superclasses can always be eliminated by introducingan ***Other*** subclass. **Multipleinheritance**: Use of multiple inheritance must be limited to the essentiality for a model.

**Constraints:** A class model may be restructured to improve clarity and capture additional constraints.

**Derived elements**: When an element is derived, it must always be indicated. Use derived elements sparingly. **Large models**: Use packages to organize large models so that the reader can understand portions of the model at a time, rather than having to deal with the whole model at once.

**Defining classes**. Define each class in a single package and show its features there. Other packages that refer to the class should use a class icon, a box that contains only the class name. This convention makes it easier to read class models and facilitates concurrent development.