**What Is the UML?**

The Unified Modeling Language (UML) is a family of graphical notations,

backed by single meta-model, that help in describing and designing software

systems, particularly software systems built using the object-oriented (OO)

style.

Graphical modeling languages have been around in the software industry for

a long time. The fundamental driver behind them all is that programming languages

are not at a high enough level of abstraction to facilitate discussions

about design.

**Ways of Using the UML**

**UML as sketch**: emphasis is on selective communication rather than complete specification.

**UML as blueprint**: intend to be comprehensive, often with the aim of reducing programming to a simple and fairly mechanical activity.

**UML as programming language**: developers draw UML diagrams that are compiled directly to executable code, and the UML becomes the source code.

. Forward Engineering

. Reverse Engineering

. Conceptual Perspective

. Software Perspective

**Notations and Meta-Models**

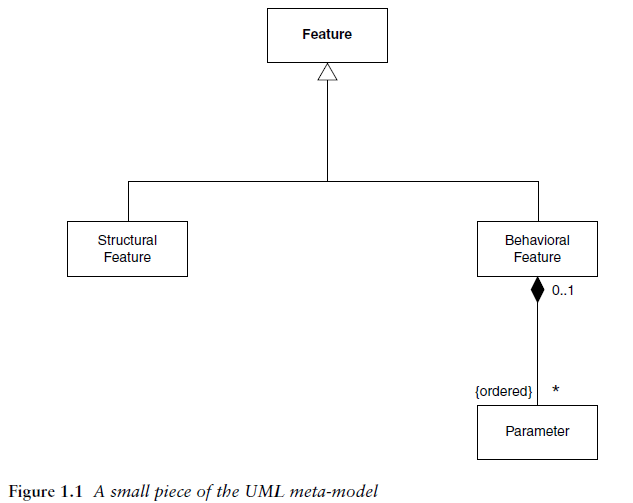
The **notation** is the graphical stuff you see in models;

it is the graphical syntax of themodeling language.

**meta-model:** a diagram, usually a class diagram, that defines the concepts of

the language.

Intuition rather than to formal definition.

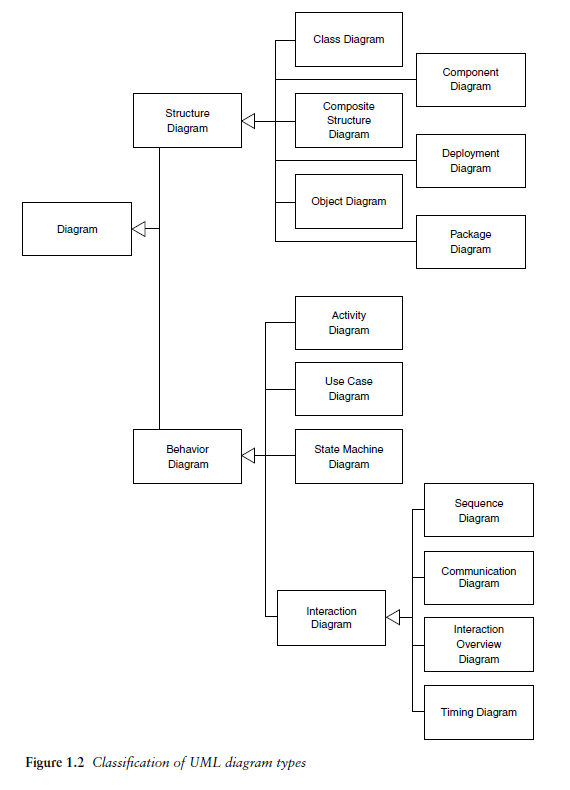


**TIP** - As you get deeper into the more detailed usage of the UML, you realize that

you need much more than the graphical notation. This is why UML tools are so

complex.

**UML Diagrams**



**What Is Legal UML?**

**prescriptive rules**

**descriptive rules**

**TIP** - You cannot look at a UML diagram and say *exactly* what

the equivalent code would look like. However, you can get a *rough idea* of what

the code would look like.

**UML Is Not Enough**

You shouldn’t hesitate to use a non-UML diagram if no UML diagram

suits your purpose.

. Screen flow diagram

. Decision table

**Class Diagrams: The Essentials**

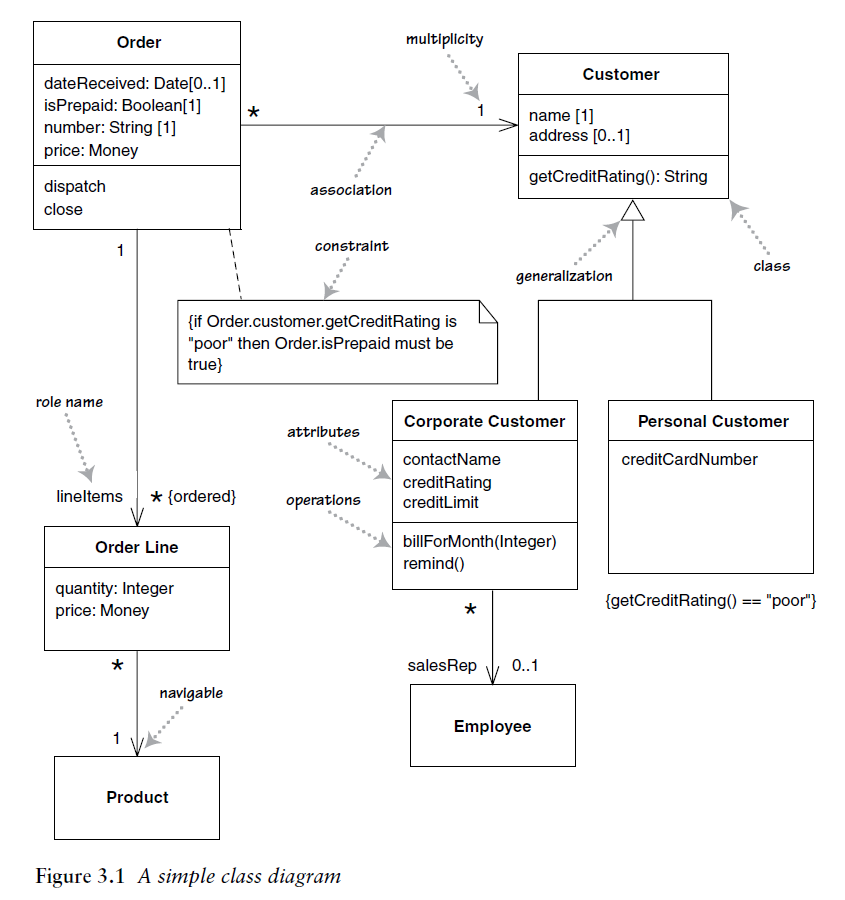
It is a **structural** diagram. A **class diagram** describes the types of objects in the system and the various

kinds of static relationships that exist among them. Class diagrams also show

the properties and operations of a class and the constraints that apply to the

way objects are connected. The UML uses the term **feature** as a general term

that covers properties and operations of a class.



**TIP - Properties** represent structural features of a class.

**TIP** - **Properties** are a single concept, but they appear in two quite distinct notations:

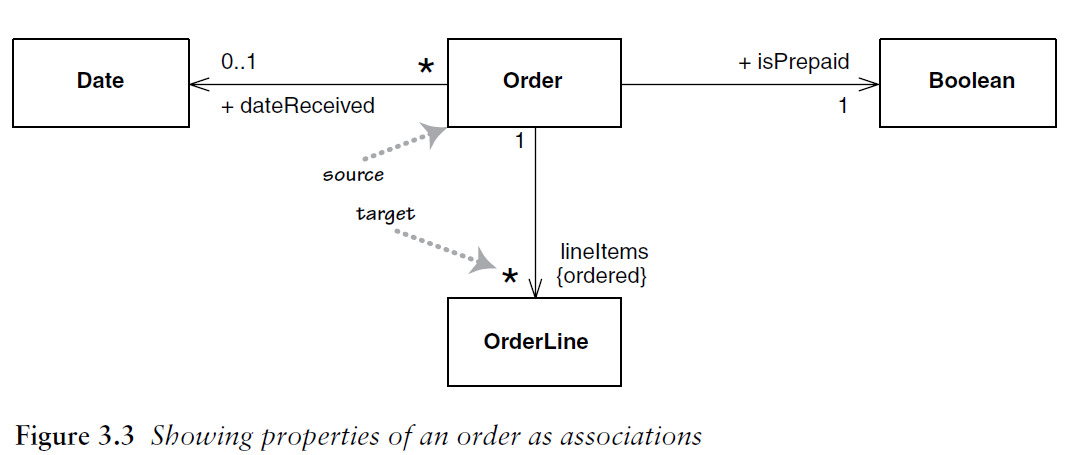
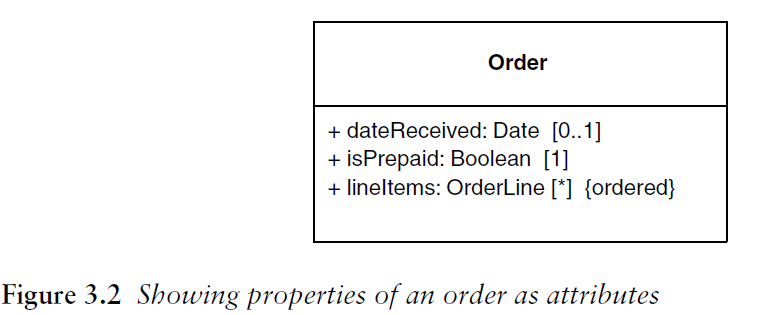
attributes and associations. Although they look quite different on a diagram,

they are really the same thing.

**Associations**

An **association** is a solid line between two classes, directed from the source

class to the target class. The name of the property goes at the target end of the association, together with its multiplicity. The target end of the association links to the class that is the type of the property.



In general, I tend to use attributes for small things,

such as dates or Booleans—in general, value types—and associations

for more significant classes, such as customers and orders.

**Navigability arrows**

It shows which class knows about the other side. For example, in Figure 3.2 only Order knows about OrderLine which means only order holds reference from OrderLine.

**Multiplicity**

The **multiplicity** of a property is an indication of how many objects may fill the

property. The most common multiplicities you will see are

• **1** (An order must have exactly one customer.)

• **0..1** (A corporate customer may or may not have a single sales rep.)

• **\*** (A customer need not place an Order and there is no upper limit to the number of Orders a Customer may place—zero or more orders.)

Association constraints example:

. {ordered}

. {unordered}

. {nonunique}

. {unique}

. {bag} - unordered, nonunique.

**TIP** - If the lower and upper bounds are the same, you can use one number;

hence, 1 is equivalent to 1..1. Because it’s a common case, \* is short for 0..\*.

**TIP** - The default multiplicity of an attribute is [1].

**TIP** - If the ordering of the items in association has meaning, you need to add {**ordered**} to the association

end. If you want to allow duplicates, add {**nonunique**}. (If you want to explicitly show the default, you can use {**unordered**} and {**unique**}.) You may also see collection-oriented names, such as {**bag**} for unordered, nonunique.

**TIP** - You should be very afraid of classes that are nothing but a collection of fields

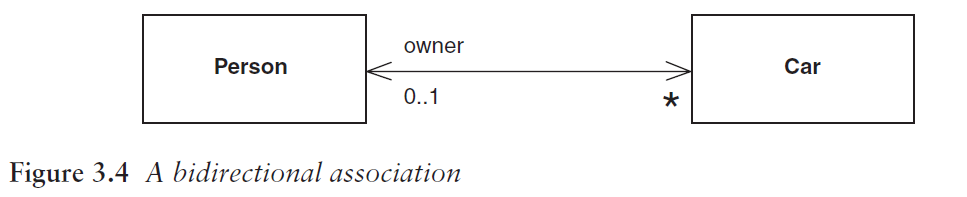
and their accessors. Object-oriented design is about providing objects that are

able to do rich behavior, so they shouldn’t be simply providing data to other

objects. If you are making repeated calls for data by using accessors, that’s a

sign that some **behavior should be moved to the object that has the data**.

**Bidirectional Associations**



A bidirectional association is a pair of properties that are linked together as

inverses. The Car class has property owner:Person [*0*..1] , and the Person class has a property cars:Car[\*]. (Note how I named the cars property in the plural form of the property’s type, a common but non-normative convention.)

The inverse link between them implies that if you follow both properties,

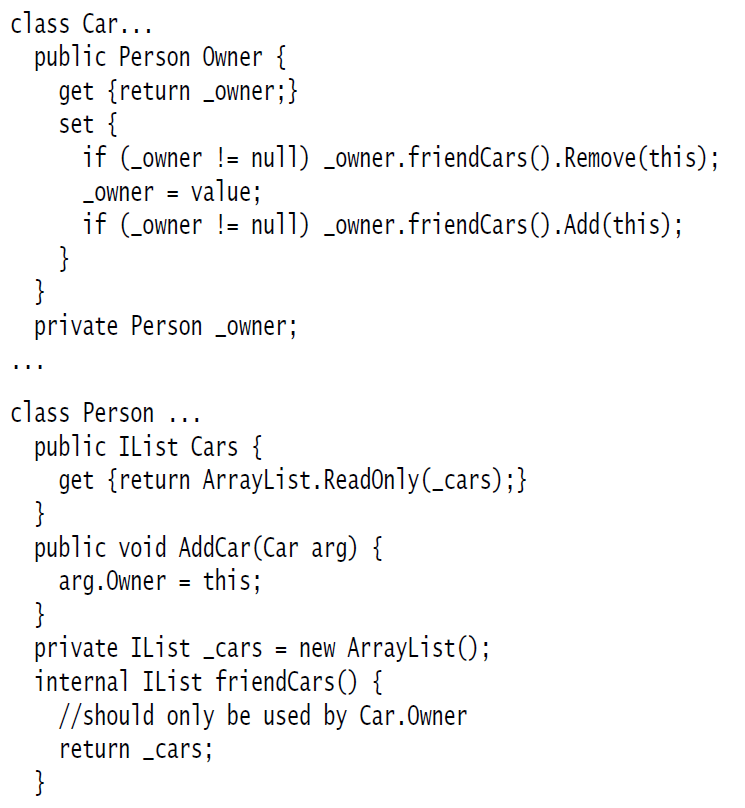
you should get back to a set that contains your starting point. For example, if I

begin with a particular MG Midget, find its owner, and then look at its owner’s cars, that set should contain the Midget that I started from.

The primary thing is to let one side of the association—a single-valued side,

if possible—control the relationship. For this to work, the slave end (Person)

needs to leak the encapsulation of its data to the master end. This adds to the slave class an awkward method, which shouldn’t really be there, unless the language has fine-grained access control.



**Operations**

visibility name (parameter-list) : return-type {property-string}

E.g. -> + balanceOn (date: Date) : Money

**TIP** - Another distinction is between operation and method. An **operation** is something

the procedure declaration—whereas a **method** is the body of a procedure. The two are different when you have polymorphism.

If you have a supertype with three subtypes, each of which overrides

the supertype’s getPrice operation, you have one operation and four methods

that implement it.

**Generalization**

With a software perspective, the obvious interpretation is **inheritance**.

With perspectives of modeling, idea is that everything we say about a Supertype—associations,

attributes, operations—is true also for a Subtype.

**Subtyping vs. Subclassing**

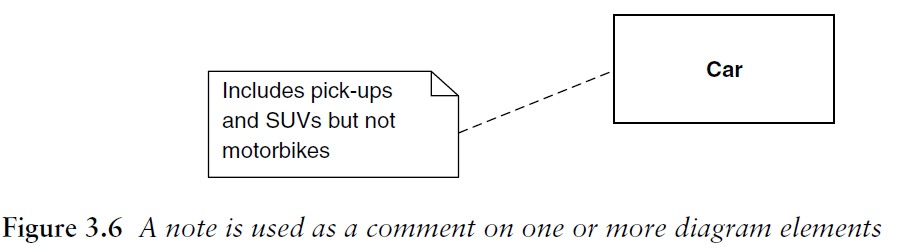
Subtyping: is about interface inheritance

Subclassing: is about implementation inheritance

**Notes and Comments**

Notes are comments in the diagrams. Notes can stand on their own, or they can

be linked with a dashed line to the elements they are commenting.



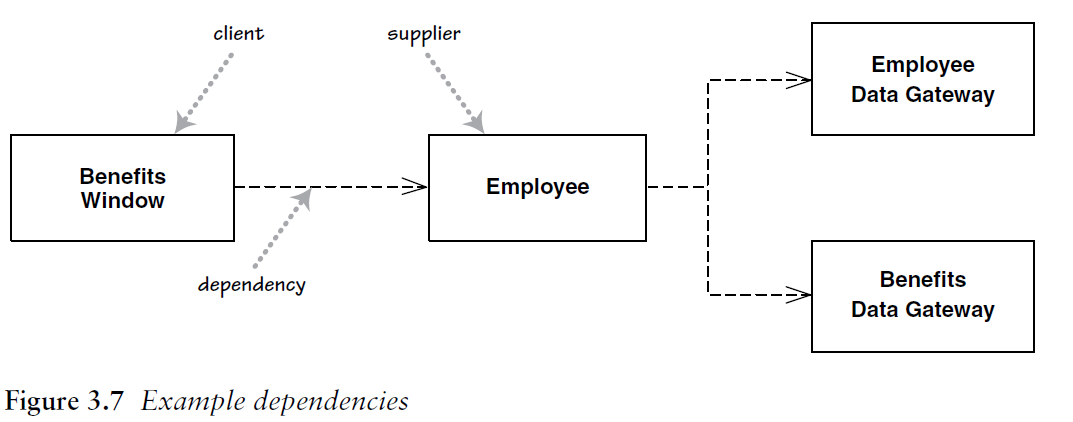
**Dependency**

A **dependency** exists between two elements if changes to the definition of one

element (the **supplier** or target) may cause changes to the other (the **client** or source).

. One direction dependency

. Direct dependency



**Class Diagrams: Advanced Concepts**

**Keywords**

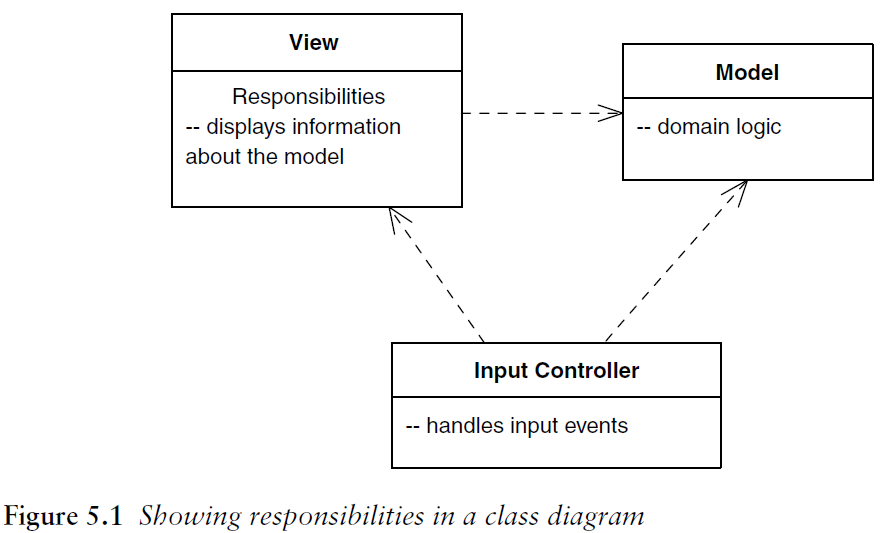
UML often tries to reduce the number of symbols and use keywords instead.

Some keywords, such as <<interface>> or {abstract}.

**TIP** - A UML **interface** is a class that has only public operations, with no method bodies.

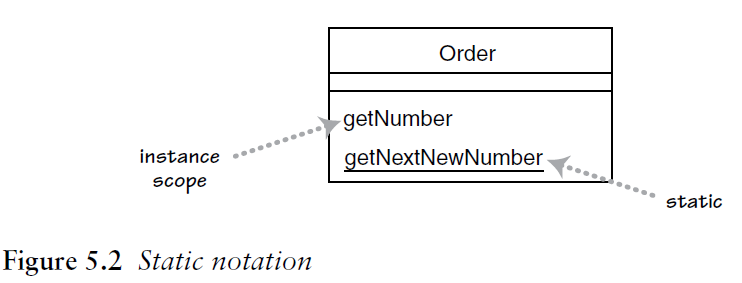
**Responsibilities**

The best way to show them is as **comment strings** in their own compartment in the class (Figure 5.1).



**Static Operations and Attributes**

Static features are **underlined** on a class diagram (see Figure 5.2).



**Aggregation and Composition**

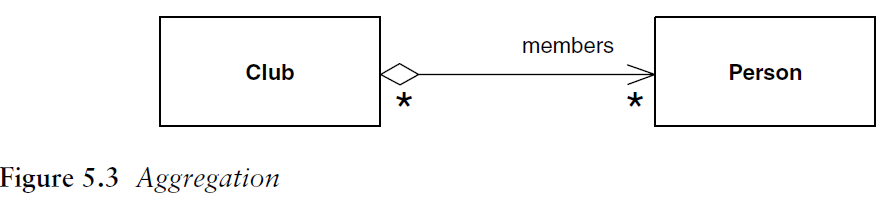
**TIP –** Understanding whether a relation of two objects is Aggregation or Composition, this understanding is a **domain centric**.

**TIP –** The story of Aggregation and Composition is all about **life-time** and **tracking**.

**TIP** – Whenever we want to track an object outside of the software we are writing, the story is about Aggregation. Imaging two objects Car and Trier. The relation between these two objects is a completely domain centric. If we have to track the Tier with an identity out of the software, the relation of these two is composition. Otherwise, it’s Composition.

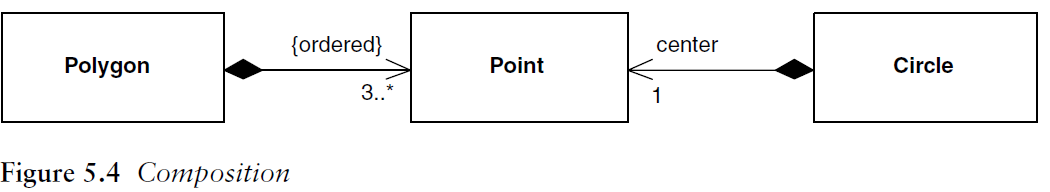
**TIP –** the story of life-time is about whether an object can live a lone when the source object gets **removed**.

**Aggregation** is the part-of relationship. The difficult thing is considering what the difference is between aggregation and association.



**Composition** is **the relationship between the whole and the part, but the whole and the part cannot be separated**.

The **“no sharing”** rule is the key to composition. Another assumption is that if you delete the polygon, it should **automatically** ensure that any owned Points also are **deleted**.



**TIP** - In Figure 5.4, an instance of Point may be part of a polygon or may be the

center of a circle, but it cannot be both. The general rule is that, although a

class may be a component of many other classes, any **instance** must be a component

of only **one owner**. The class diagram may show multiple classes of

potential owners, but any instance has only a single object as its owner.

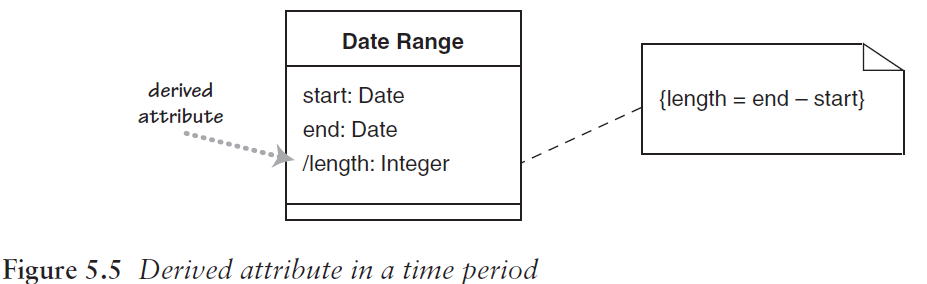
**TIP** - Composition is a good way of showing properties that own by value, properties

to **value objects**, or properties that have a strong and somewhat

exclusive ownership of particular other components.

**Derived Properties**

**Derived properties** can be **calculated** based on other values. You can use derivation to indicate the difference between a calculated value and a stored value.



**TIP** - Derivation can also be applied to properties using association notation. In

this case, you simply mark the name with a /.

**Interfaces and Abstract Classes**

. Abstract class + abstract operation (pure declaration)

. Classes have two kinds of relationships with interfaces:

. Providing (when a class implements the interface)

. Requiring (when a class needs an instance of the interface)

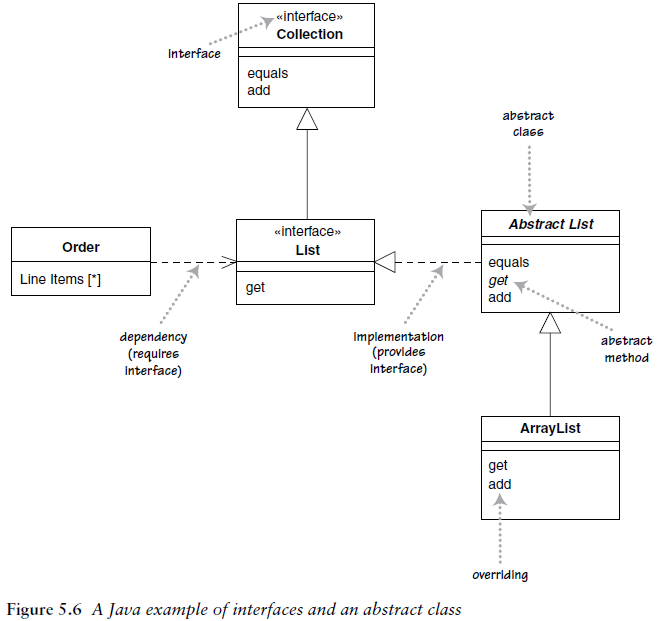
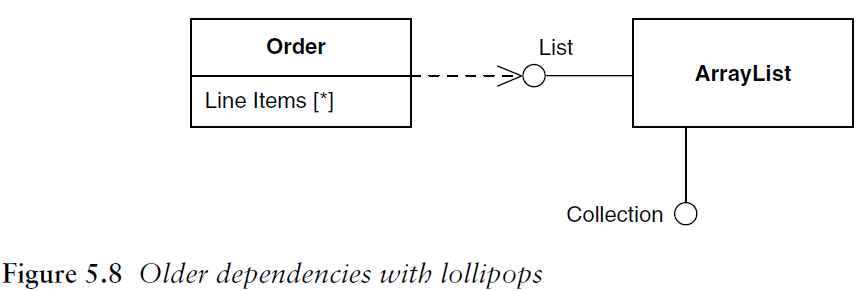


Figure 5.8 shows a more compact notation.



**Reference Objects and Value Objects**

**Reference objects** are such things as Customer. Here, **identity** is very important because you usually want only one software object to designate a customer in the real world. Any object that references a Customer object will do so through a reference, or pointer; all objects that reference this Customer will reference

the same software object. That way, changes to a Customer are available to all users of the Customer. If you have two references to a Customer and wish to see whether they are the same, you usually compare their identities. Copies may be disallowed; if they are allowed, they tend to be made rarely, perhaps for archive purposes or for replication across a network. If copies are made, you need to sort out how

to synchronize changes.

**Value objects** are such things as Date. You often have multiple value objects representing the same object in the real world. For example, it is normal to have hundreds of objects that designate 1-Jan-04. These are all interchangeable copies. Value objects should be **immutable;** in other words, you should not be able to

take a date object of 1-Jan-04 and change the same date object to be 2-Jan-04.

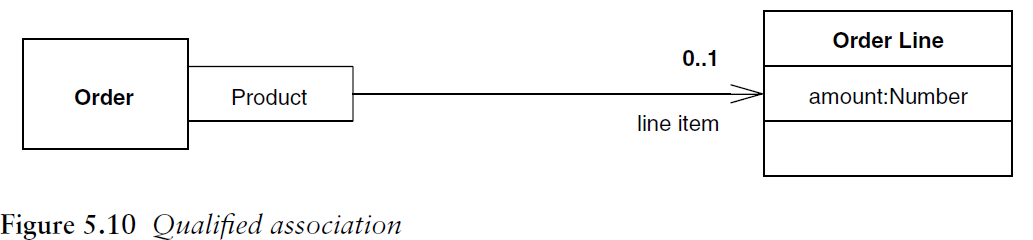
Instead, you should create a new 2-Jan-04 object and use that instead. The reason

is that if the date were shared, you would update another object’s date in an

unpredictable way, a problem referred to as **aliasing**.

In days gone by, the difference between reference

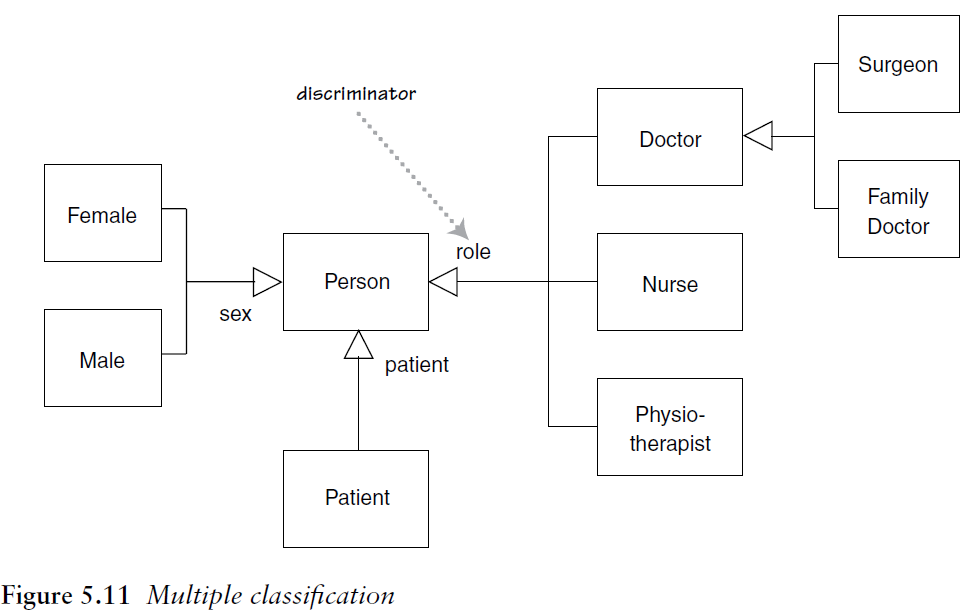
**Qualified Associations**



The qualifier says that in connection with an Order, there may be one Order Line for each instance of Product. So the diagram says that an Order has 0..1 Line Items per Product. A multiplicity of 1 would indicate that Order would have to have a Line Item for every instance of Product. A \* would indicate that you would have multiple Line Items per Product but that access to the Line Items is indexed by Product.

**Classification and Generalization**

Whenever we want to analyses an entity from **different point of views** and whenever we do analysis in combination with **decomposition** in order to better understanding and visualization, in fact we are doing classification. As far as, the purpose of classification is to visualization, we need a way to determine our point of view from which we are analyzing the entity and that’s why we use a discriminator to indicate our point of view. See figure 5.11:



**TIP** – usually we don’t use classification in programming because it leads to use multiple inheritance. For example, in Figure 5.11 imagine we need a Female Surgeon Person, what should we do? It seems there is no way except using multiple inheritance. This is not good practice because in now a day most of the programming languages don’t support multiple inheritance. So, in programming we prefer to convert multiple classification relationship into **Enumeration**.

**Association Class**

**Association classes** allow you to add attributes, operations, and other features

to associations, as shown in Figure 5.12. We can see from the diagram that a

person may attend many meetings. We need to keep information about how

awake that person was; we can do this by adding the attribute attentiveness to

the association.

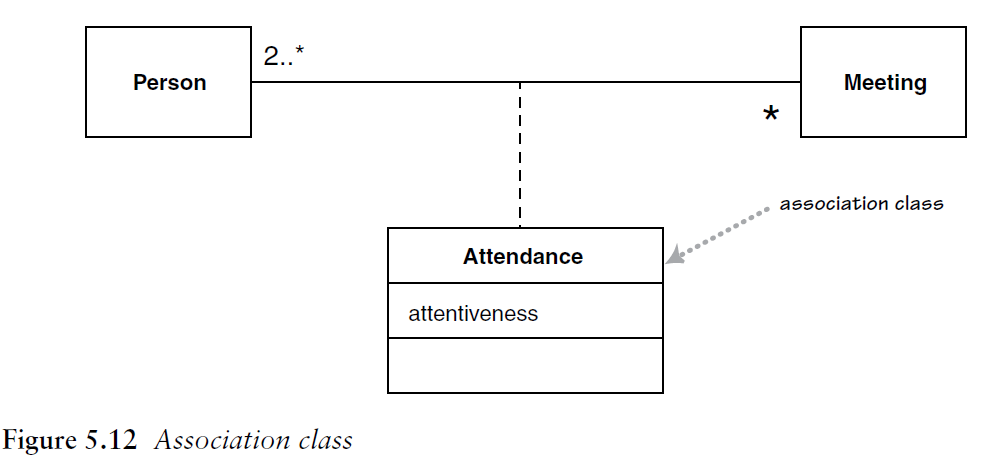
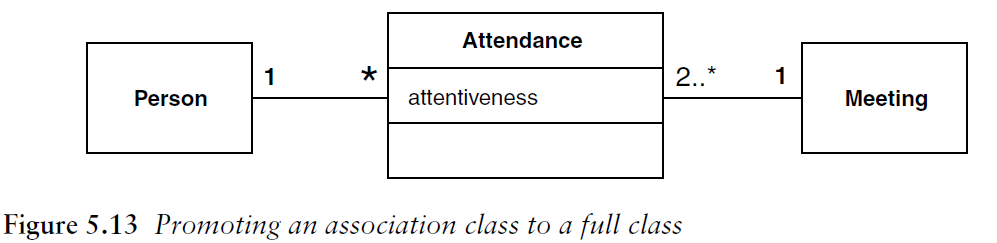


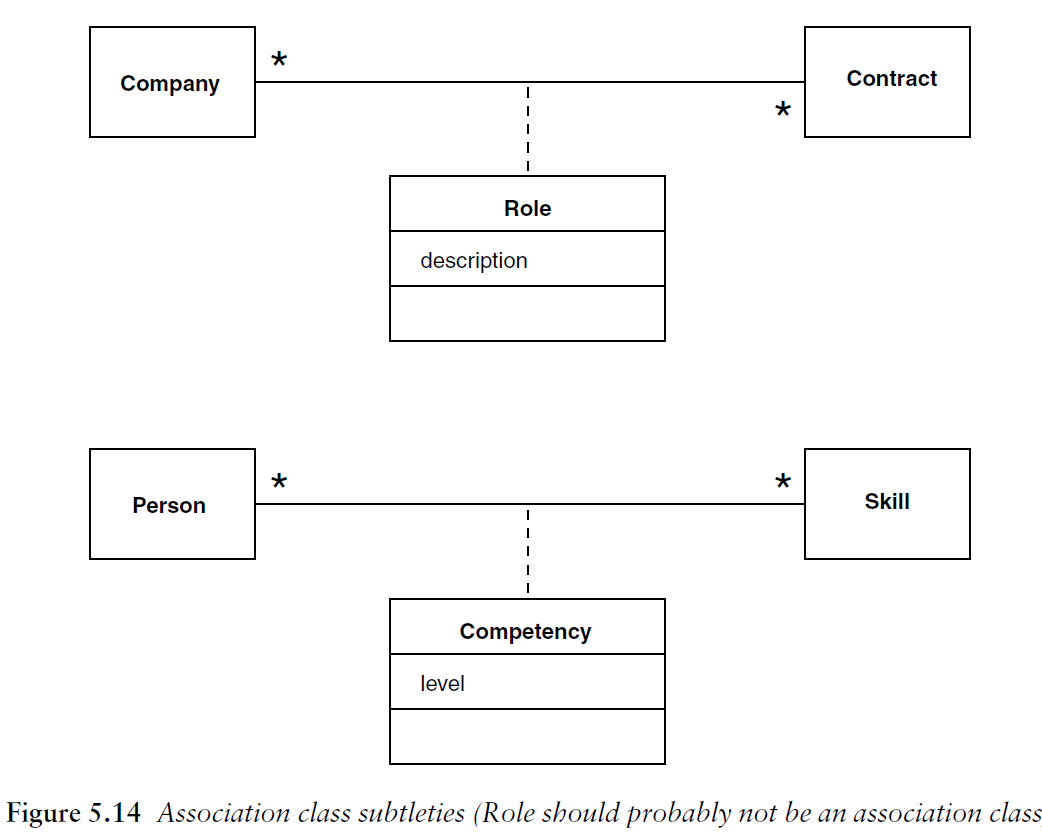
Figure 5.13 shows another way to represent this information.



**TIP** - The association class adds an **extra constraint**, in that there can be only one instance of the association class between any two participating objects.

**Another Example:**

The following two diagrams, are identical with same message to the reader. Both emphasis the constraints which says both objects in the relationship can not have more than one Competency (Person-Skill) or Role (Company-Contract). Be alerted that this constraint cannot be meet structurally and should be handled via pre-condition or post-condition in programming.



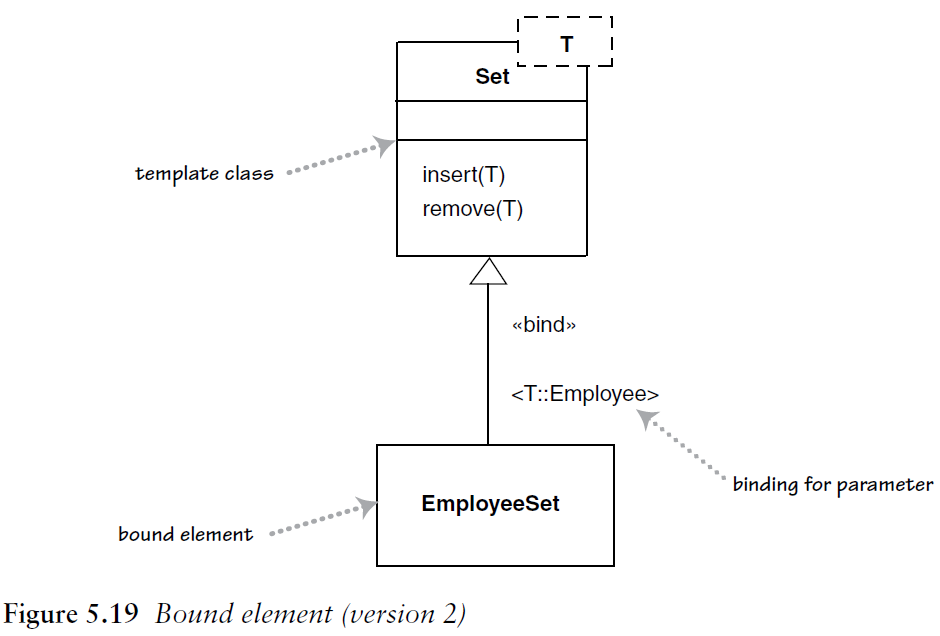
**TIP** - You can have only one (or want to have only one) competency for each combination of Person and Skill. The top diagram in Figure 5.14 would not allow a Company to have more than one Role on a single contract. If you need to allow this, you need to make Role a full class, in the style of Figure 5.13

**Template (Parameterized) Class**

class Set <T> {

void insert (T newElement);

void remove (T anElement);



A use of a parameterized class, such as Set<Employee>, is called a **derivation**. (see Figure 5.19)

**TIP** - Using a derivation is *not* the same as subtyping, however. You are not

allowed to add features to the bound element, which is completely specified by

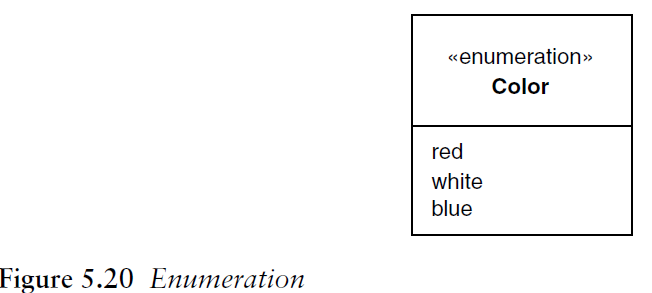
its template; you are adding only restricting type information. If you want to

add features, you must create a subtype.

**Enumerations**

**Enumerations** (Figure 5.20) are used to show a fixed set of values that don’t

have any properties other than their symbolic value.



**Active Class**

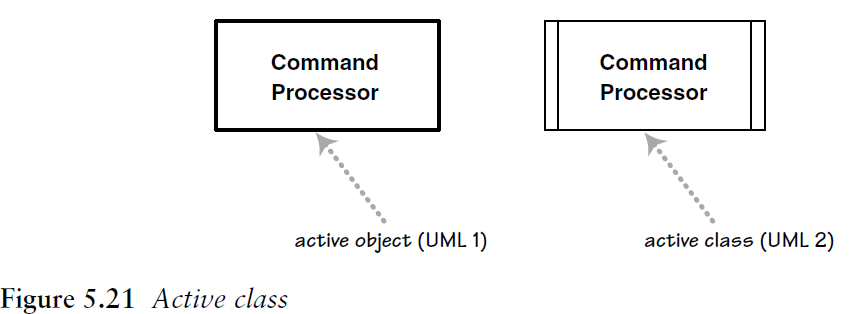
An **active class** has instances, each of which executes and controls its own

thread of control. Method invocations may execute in a client’s thread or in the

active object’s thread. A good example of this is a command processor that

accepts command objects from the outside and then executes the commands

within its own thread of control.

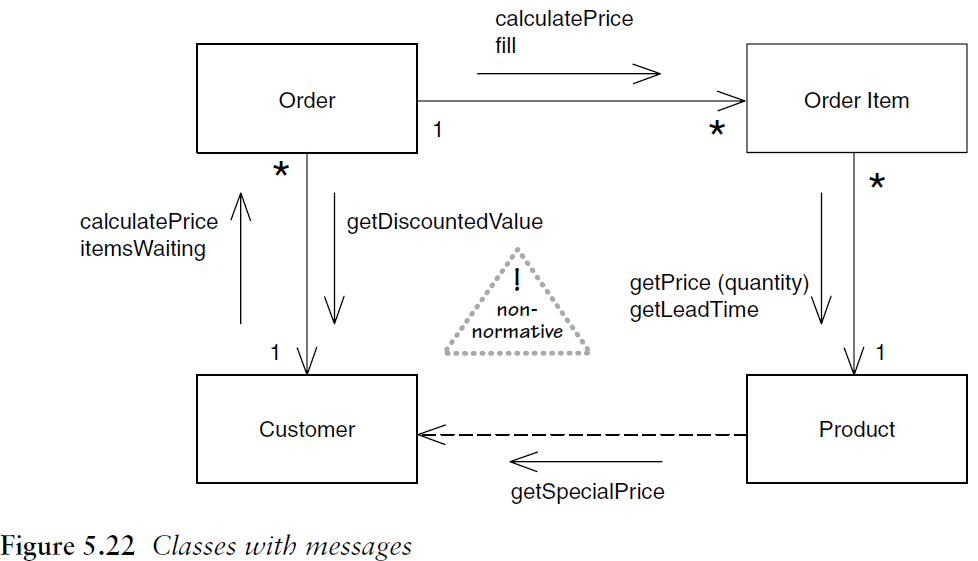


**Visibility**

UML provides four abbreviations for visibility: **+** (public), **–** (private), ~ (package), and **#** (protected).

**Messages**

Standard UML does not show any information about message calls on class diagrams. However, I’ve sometimes seen conventional diagrams like Figure 5.22.



**State Machine Diagrams**

It is a behavioral diagram. You draw a state machine diagram for **a single class** to show the **lifetime behavior** of a single object.

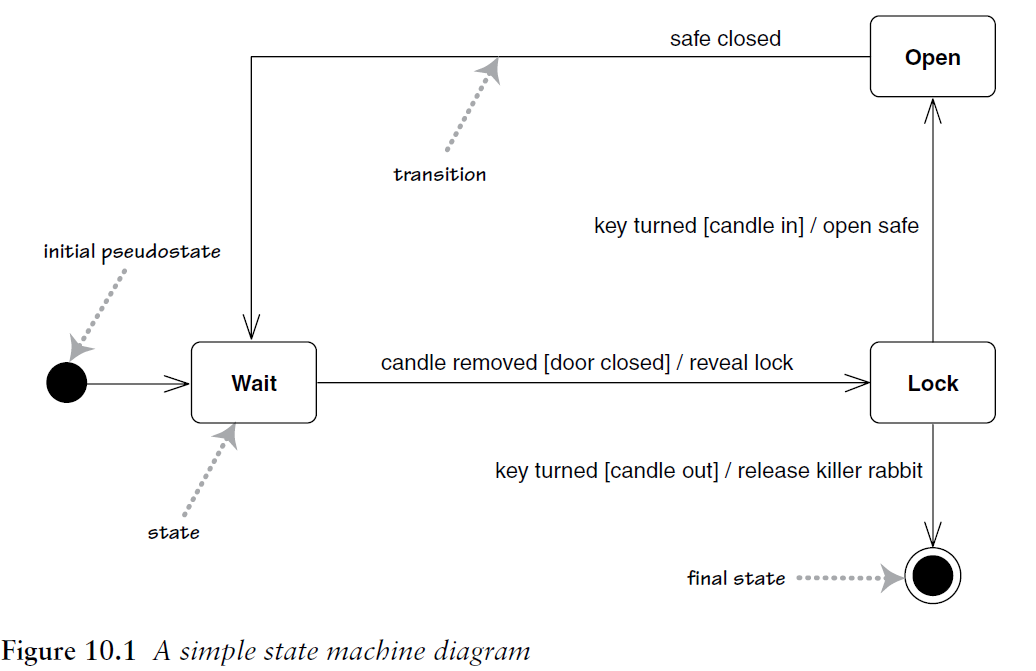


Figure 10.1 shows a state machine diagram of the controller class that directs an unusual security system. The state diagram starts with the state of the controller object when it’s created: in Figure 10.1, the Wait state. The diagram indicates this with **initial pseudostate**, which is not a state but has an arrow that points to the initial state.

The diagram shows that the controller can be in three states: Wait, Lock, and Open. The diagram also gives the **rules** by which the controller changes from state to state. These rules are in the form of **transitions**: the lines that connect the states.

The **transition** indicates a movement from one state to another. Each transition has a label that comes in three parts: **trigger-signature [guard]/activity**. All the parts are optional.

trigger-signature: is usually **a single event** that triggers a potential change of state.

guard: if present, is a **Boolean condition** that must be true for the transition to be taken.

activity: is **some** behavior that’s executed during the transition. It may be any behavioral expression.

All three parts to a transition are **optional**. A missing activity indicates that you don’t do anything during the transition. A missing guard indicates that you always take the transition if the event occurs. A missing trigger-signature is rare but does occur. It indicates that you take the transition immediately, which you see mostly with **activity states**, which I’ll come to in a moment.

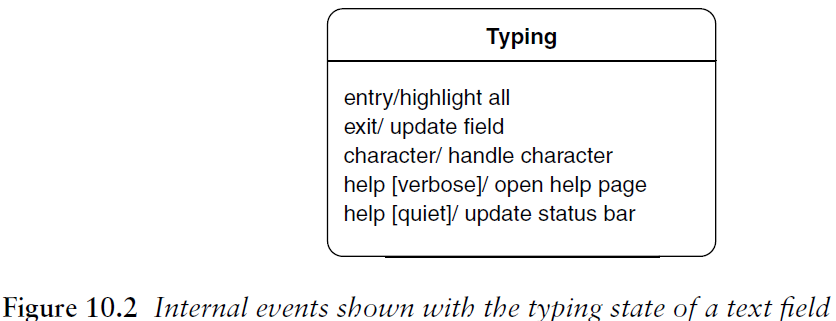
Figure 10.1, you read the outward transition from the Wait state as “In the Wait state if the candle is removed providing the door is closed, you reveal the lock and move to the Lock state.”

The final state indicates that the state machine is completed, implying the deletion of the controller object. Thus, if someone should be so careless as to fall for my trap, the controller object terminates, so I would need to put the rabbit in its cage, mop the floor, and reboot the system.

**Internal Activities**

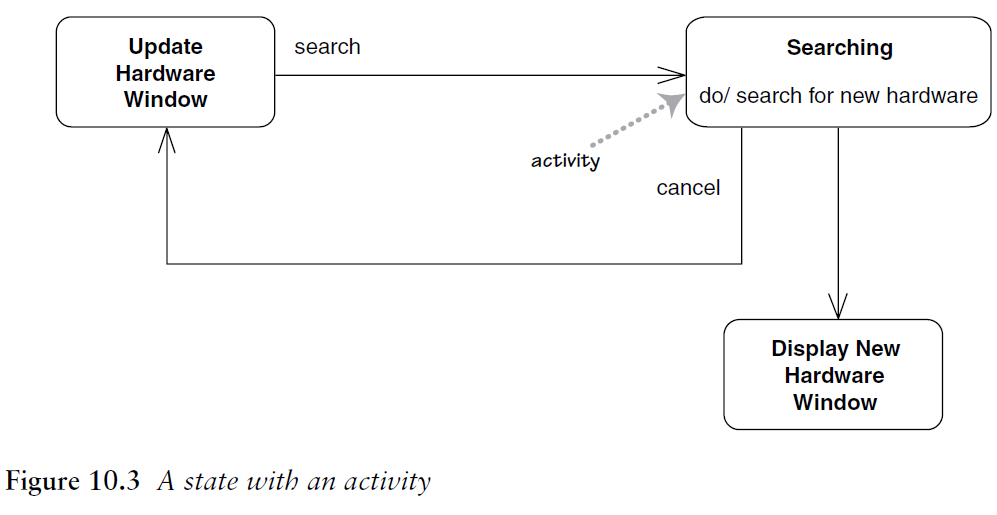
States can react to events without transition, using **internal activities:** putting the event, guard, and activity inside the state box itself.

Figure 10.2 shows a state with internal activities of the character and help events, as you might find on a UI text field. An internal activity is similar to a **self-transition: a transition that loops back to the same state**. The syntax for internal activities follows the same logic for event, guard, and procedure.



**Activity States**

In the states we’ve described so far, the object is quiet and waiting for the next event before it does something. However, you can have states in which the object is **doing some ongoing work**.



The Searching state in Figure 10.3 is such an **activity state:** The ongoing activity is marked with the **do/**; hence the term **do-activity**. Once the search is completed, **any transitions without an event**, such as the one to display new hardware, are taken. If the cancel event occurs during the activity, the do-activity is unceremoniously halted, and we go back to the Update Hardware Window state.

**TIP** - Both do-activities and regular activities represent carrying out some behavior.

The critical difference between the two is that regular activities occur “**instantaneously**”

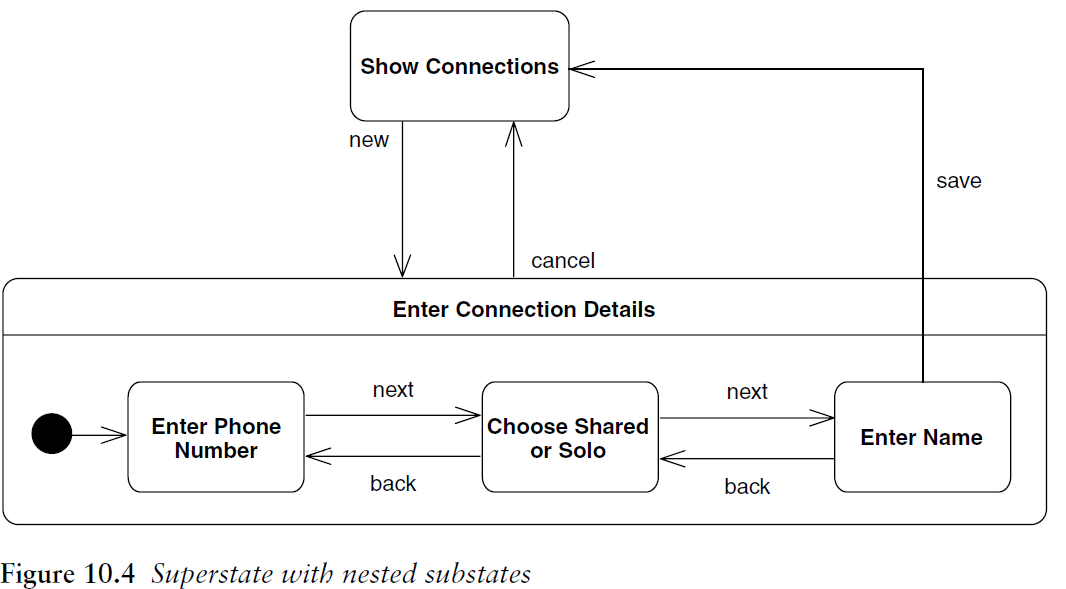
and **cannot be interrupted** by regular events, while do-activities can

take finite time and **can be interrupted**, as in Figure 10.3.

**TIP** - UML 1 used the term **action** for regular **activities** and used activity only for do-activities.

**Superstates**

Often, you’ll find that several states share common transitions and internal activities. In these cases, you can make them substates and **move the shared** **behavior into a superstate**, as in Figure 10.4. Without the superstate, you would have to draw a cancel transition for all three states within the Enter Connection Details state.



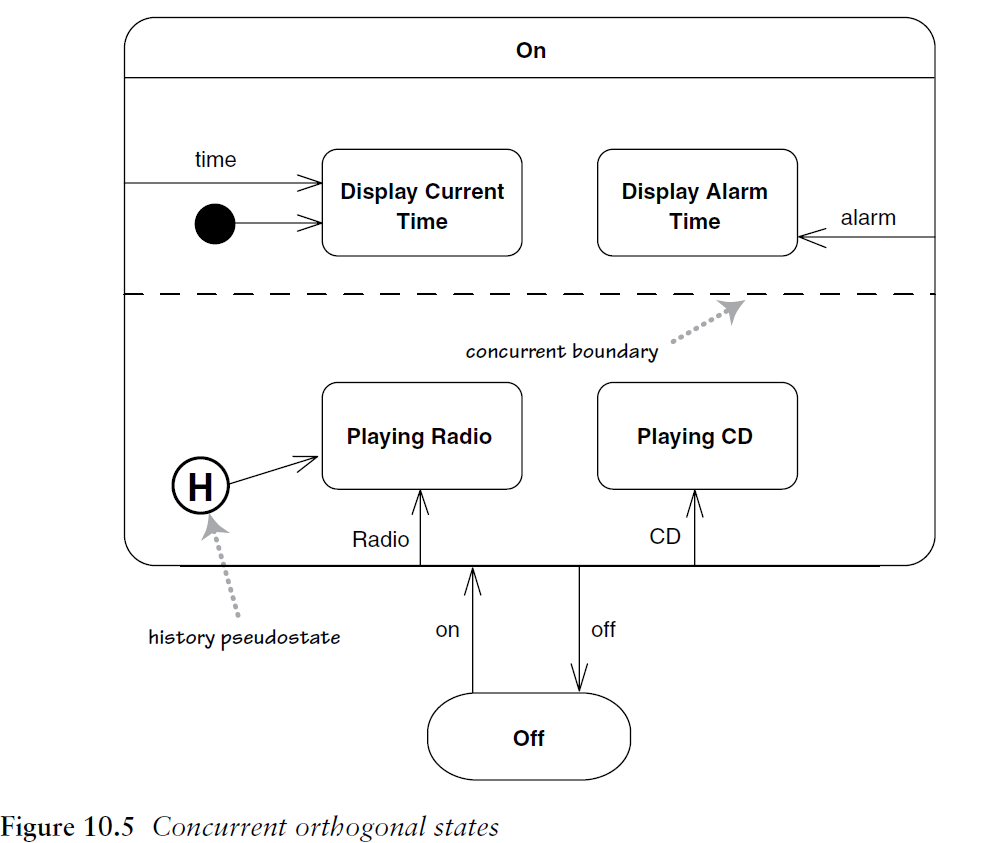
**Concurrent States**

States can be broken into several orthogonal state diagrams that run concurrently.

Figure 10.5 shows a pathetically simple alarm clock that can play either CDs or the radio and show either the current time or the alarm time.

The choices CD/radio and current/alarm time are orthogonal choices. If you wanted to represent this with a nonorthogonal state diagram, you would need a messy diagram that would get very much out of hand should you want more states. Separating out the two areas of behavior into separate state diagrams makes it much clearer.

Figure 10.5 also includes a **history pseudostate**. This indicates that when the clock is switched on, the radio/CD choice goes back to the state the clock was in when it was turned off. The arrow from the history pseudostate indicates what state to be in on the first time when there is no history.



**Sequence Diagrams**

**Interaction diagrams** describe how groups of objects **collaborate** in some behavior.

The UML defines several forms of interaction diagram, of which the most

common is the sequence diagram.

Typically, a sequence diagram captures the behavior of a **single scenario**. The

diagram shows a number of example objects and the **messages** that are passed

between these objects within the use case.