UML Distilled

Chapter 3

Class Diagrams: The Essentials

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.

Properties

Properties represent structural features of a class. They appear in two quite distinct notations. Attributes and associations. Although they look quite different on a diagram, they are really the same thing

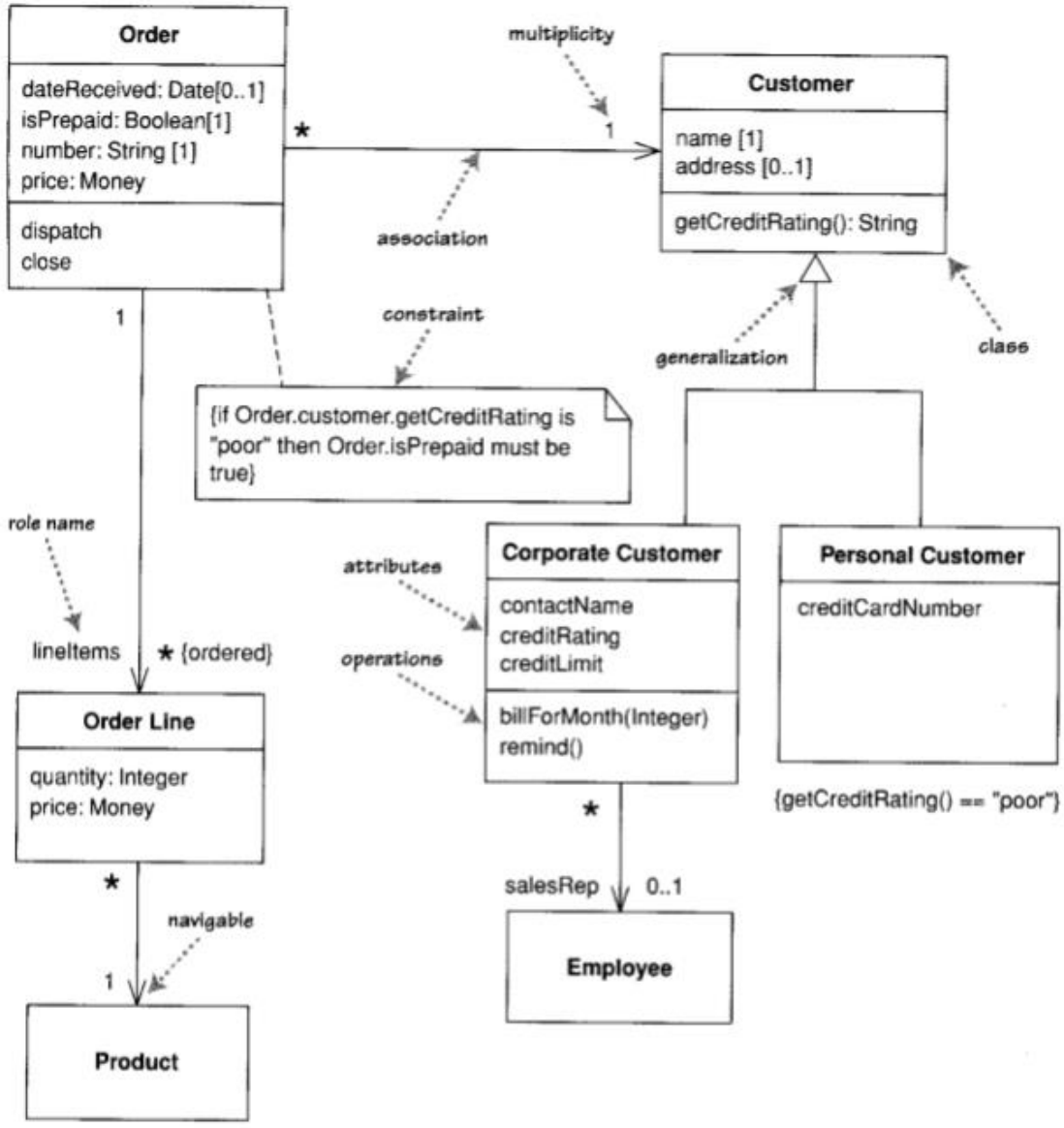

Figure 3.1 *A simple class diagram*

Attributes

The arrtibute notation describes a property as a line of text within the class box itself.

```
visibility name: type multiplicity = default {property-string}
```

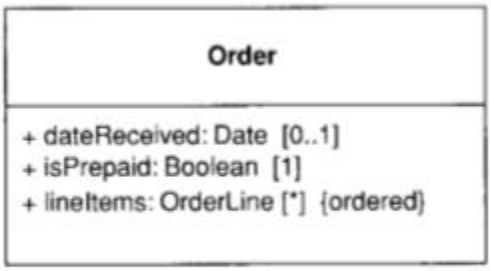

Figure 3.2 *Showing properties of an order as attributes*

Associations

An association is a solid line between two lasses, directed from the source class to the target class.

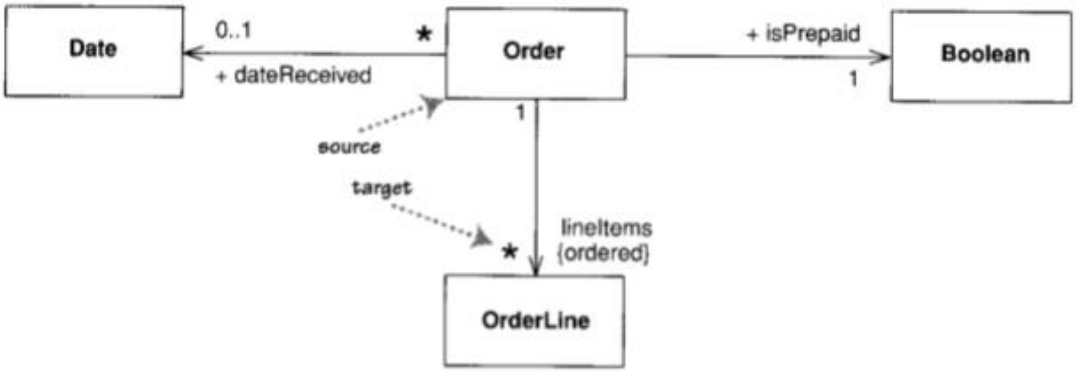

Figure 3.3 *Showing properties of an order as associations*

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.)

In attributes, you come across various terms that refer to the multiplicity

- **Optional** implies a lower bound of 0.
- **Mandatory** implies a lower bound of 1 or possibly more.
- **Single-valued** implies an upper bound of 1.
- **Multivalued** implies an upper bound of more than 1: usually *.

Programming Interpretation of Properties

There’s no way to interpret properties in code. The most common software representation is that of a field or property of your programming language.

Bidirectional Associations

Figure 3.4 *A bidirectional association*

A bidirectional association is a pair of properties that are linked together as inverses. The Car class has property owner:Person[1], and the Person class has a property cars:Car[*].

Operations

Operations are the actions that a class knows to carry out. Operations most obviously correspond to the methods on a class.

The full UML syntax for operations is:

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

- This visibility marker is public (+) or private (-); others on page 83.
- The name is a string.
- The parameter-list is the list of parameters for the operation.
- The return-type is the type of the returned value, if there is one.
- The property-string indicates property values that apply to the given operation.

The parameters in the parameter list are notated in a similar way to attributes. The form is:

```
direction name: type = default value
```

- The name, type, and default value are the same as for attributes.
- The direction indicates whether the parameter is input (in), output (out) or both (inout). If no direction is shown, it's assumed to be in.

An example operation on account might be:

```
+ balanceOn (date: Date) : Money
```

The parameters in the parameter list are notated in a similar way to attributes. The form is:

```
direction name: type = default value
```

- The name, type, and default value are the same as for attributes.
- The direction indicates whether the parameter is input (in), output (out) or both (inout). If no direction is shown, it's assumed to be in.

An example operation on account might be:

```
+ balanceOn (date: Date) : Money
```

Generalization

A typical example of generalization involves the personal and corporate customers of a business. They have differences but also similarities. The similarities can be placed in a general Customer class (the supertype), with Personal Customer and Corporate Customer as subtypes.

Notes and Comments

Notes are comments in the diagrams. They can appear in any kind of diagram.

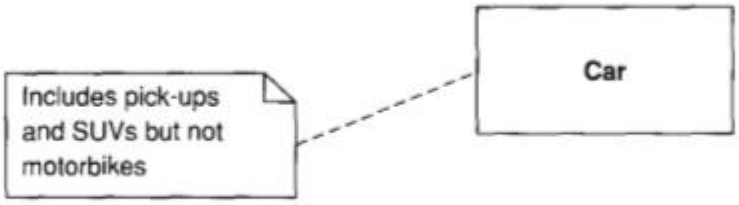

Figure 3.6 *A note is used as a comment on one or more diagram elements*

Dependency

A **dependency** exists between two elements if changes to the definition of the element (the **supplier**) may cause changes to the other (the client).

Dependency is in only one direction and goes from the presentation class to the domain class.

Dependencies can be determined by looking at code, so tools are ideal for doing dependency analysis.

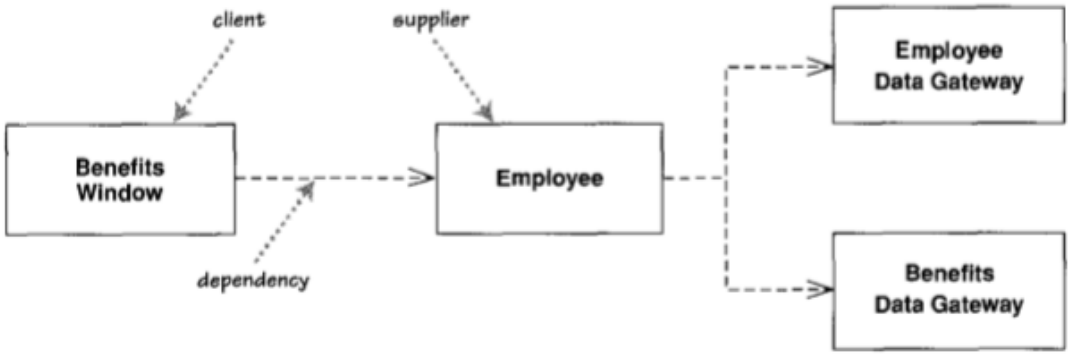

Figure 3.7 *Example dependencies*

Constraint Rules

The basic constructs of association, attribute, and generalization do much to specify important constraints, but they cannot indicate every constraint. These constraints still need to be captured; the class diagram is a good place to do that.

The UML allows you to use anything to describe constraints. The only rule is that you put them inside braces ({})

Design by Contract

At the heart of Design by Contract is the assertion. An **assertion** is a Boolean statement that should never be false and, therefore, will be false only because of a bug. Typically, assertions are checked only during debug and are not checked during production execution.

Design by Contract uses three particular kinds of assertions: post-conditions, pre-conditions, and invariants. Pre-conditions and post-conditions apply to operations. A **post-condition** is a statement of what the world should look like after execution of an operation.

A **pre-condition** is a statement of how we expect the world to be before we execute an operation. An **exception** occurs when an operation is invoked with its pre-condition satisfied yet cannot return with its post-condition satisfied.

An **invariant** is an assertion about a class. The invariant is "always" true for all instances of the class.

When to Use Class Diagrams

- Don't try to use all the notations available to you. Start with the simple stuff in this chapter: classes, associations, attributes, generalization, and constraints.
- Work hard on keeping software out of the discussion and keeping the notation very simple.
- Don't draw models for everything; instead, concentrate on the key areas

Chapter 4

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.

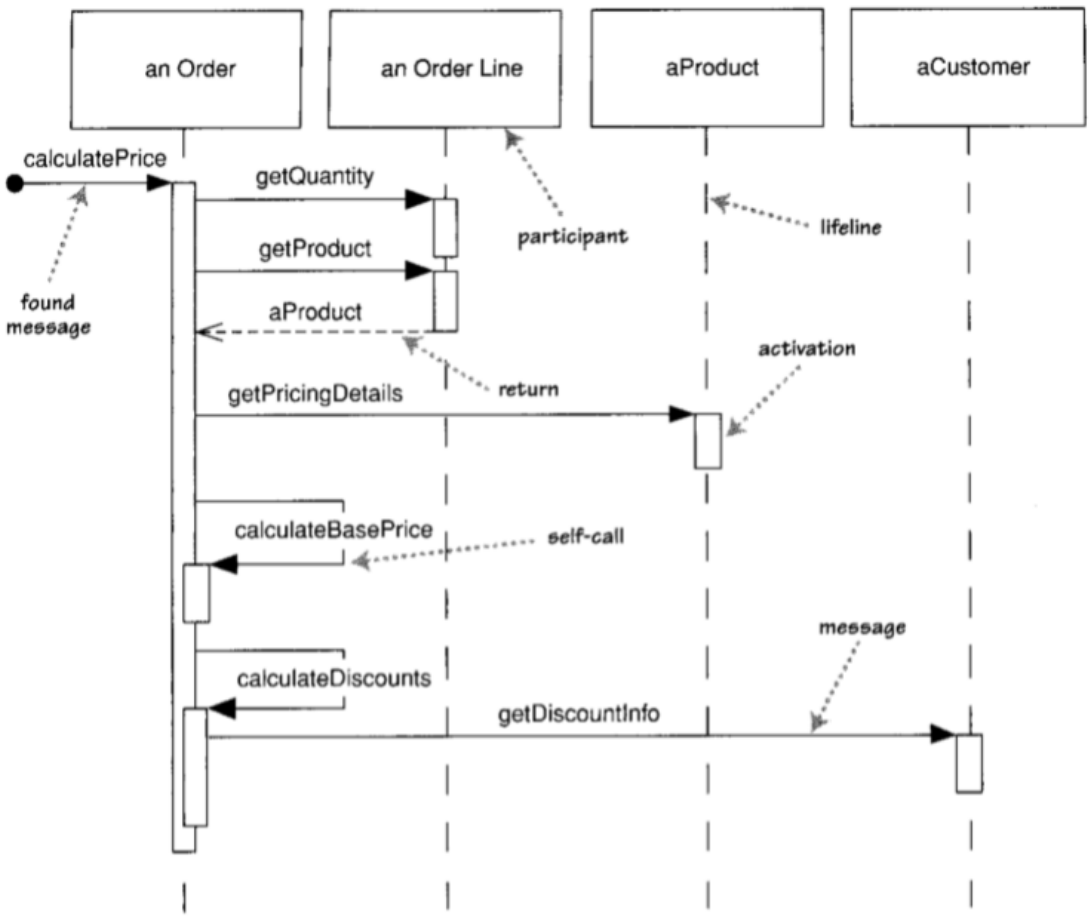

Figure 4.1 *A sequence diagram for centralized control*

One of the nice things about a sequence diagram is that 1 almost don't have to explain the notation. You can see that an instance of order sends getQuantity and getProduct messages to the order line. You can also see how we show the order invoking a method an itself and how that method sends getDiscountInfo to an instance of customer.

For another approach to this scenario, take a look at Figure 4 .2. The basic problem is still the Same, but the way in which the participants collaborate to implement it is very different.

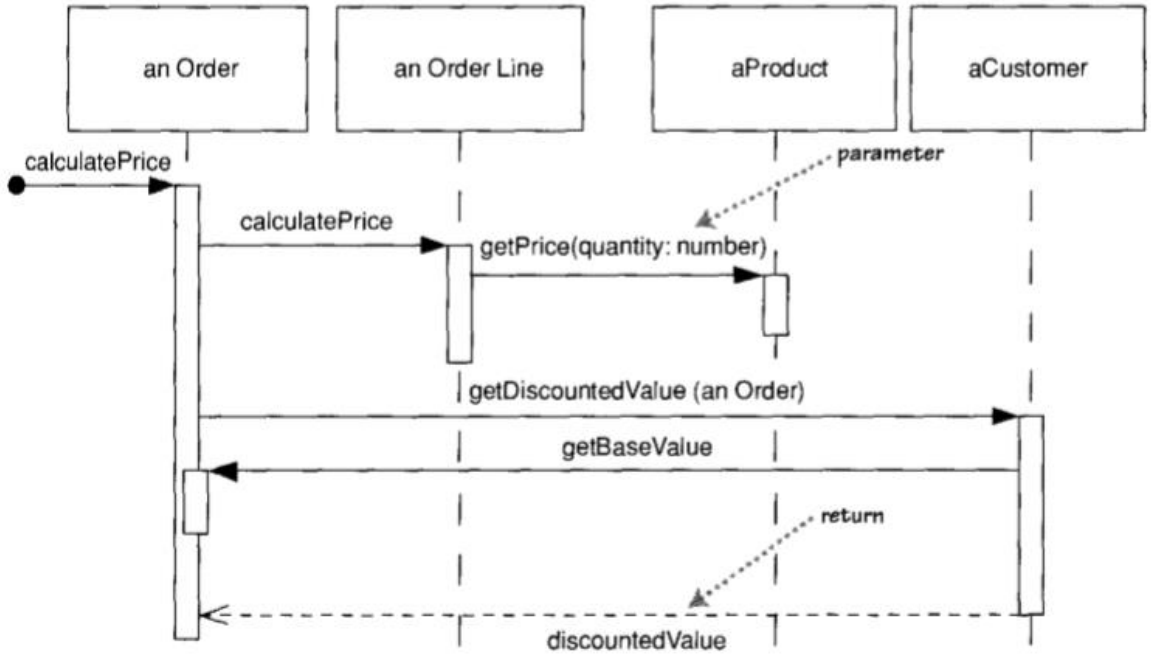

Figure 4.2 *A sequence diagram for distributed control*

Figure 4.1 is centralized control, with one participant pretty much doing all the processing and other participants there to supply data . Figure 4.2 uses distributed control, in which the processing is split among many participants, each one doing a little bit of the algorithm.

Furthermore, by distributing control, you create more opportunities for using polymorphism rather than using conditional logic.

Creating and Deleting Participants

Deletion of a participant is indicated by big X. A message arrow going into the X indicates one participant explicitly deleting another; an X at the end of a lifeline shows a participant deleting itself.

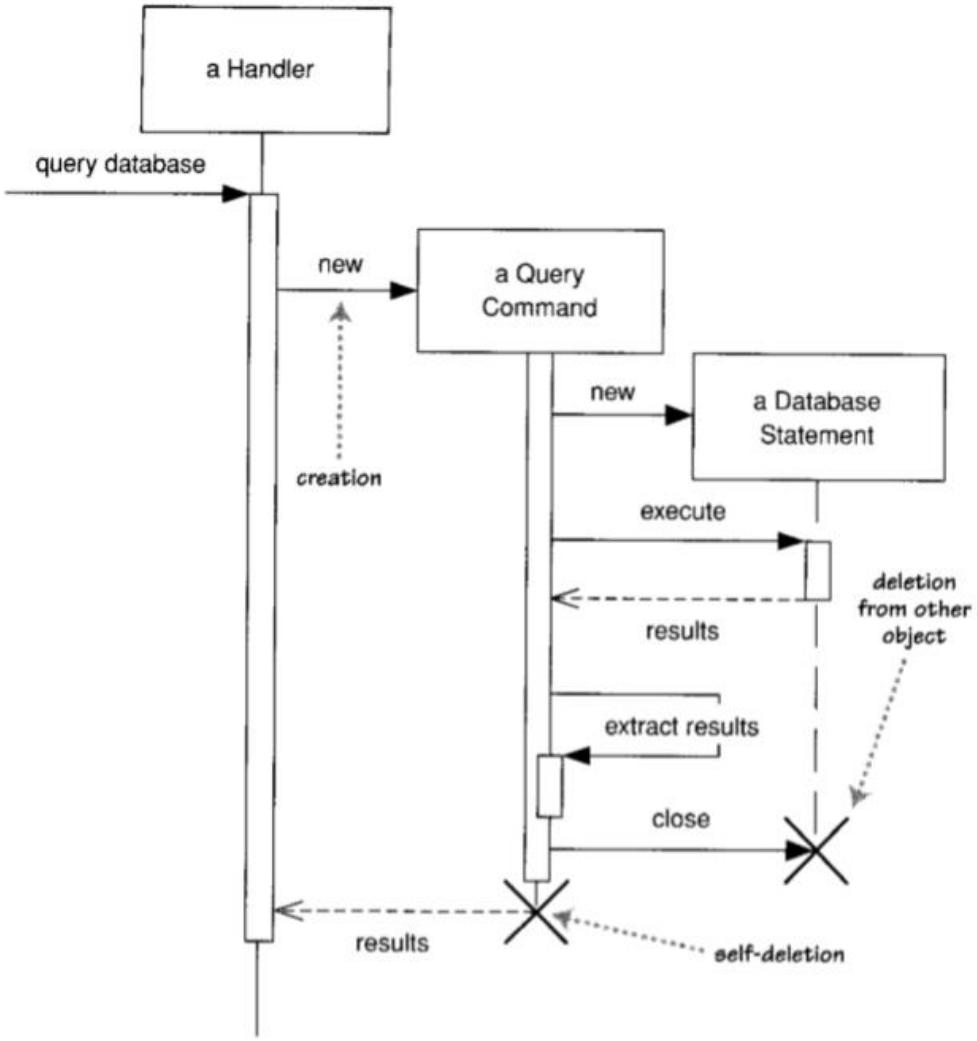

Figure 4.3 *Creation and deletion of participants*

Loops, Conditionals, and the Like

Treat sequence diagrams as a visualization of how objects interact rather than as a way of modeling control logic.

```
procedure dispatch
  foreach (lineitem)
    if (product.value > $10K)
      careful.dispatch
    else
      regular.dispatch
    end if
  end for
  if (needsConfirmation) messenger.confirm
end procedure
```

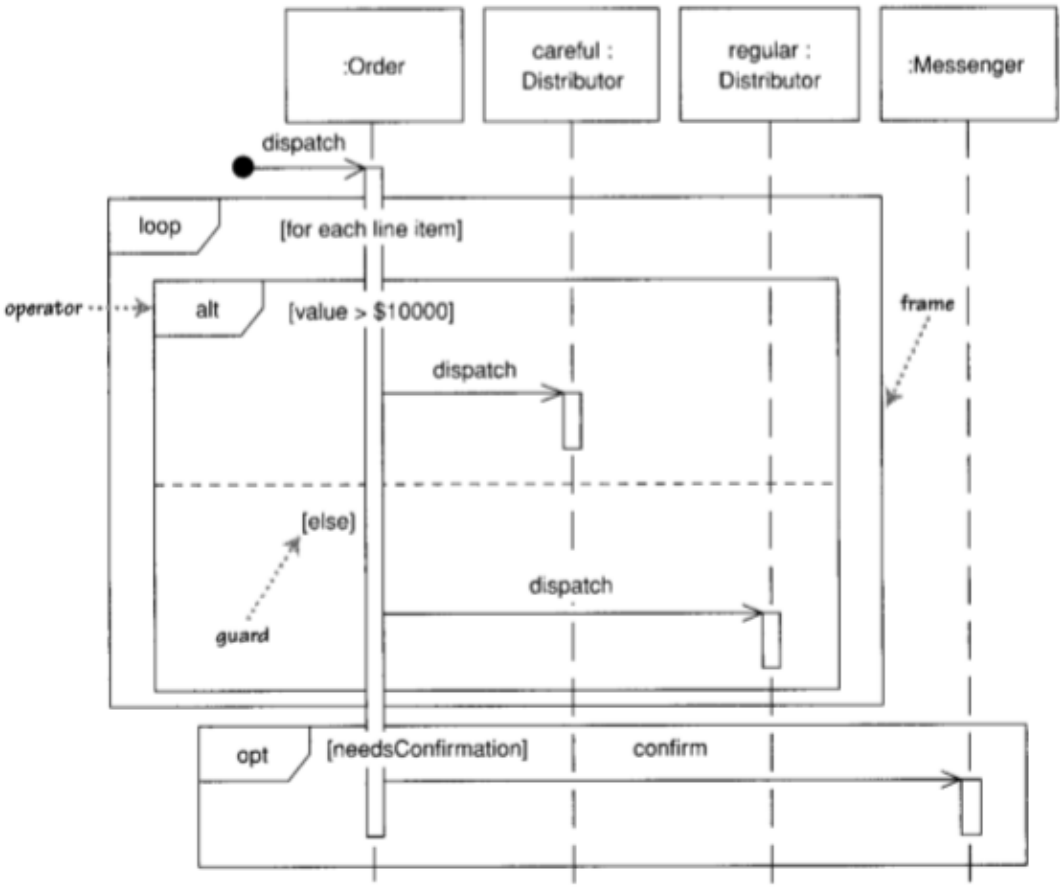

Figure 4.4 *Interaction frames*

Table 4.1 *Common Operators for Interaction Frames*

Operator	Meaning
`alt`	Alternative multiple fragments; only the one whose condition is true will execute (Figure 4.4).
`opt`	Optional; the fragment executes only if the supplied condition is true. Equivalent to an `alt` with only one trace (Figure 4.4).
`par`	Parallel; each fragment is run in parallel.
`loop`	Loop; the fragment may execute multiple times, and the guard indicates the basis of iteration (Figure 4.4).
`region`	Critical region; the fragment can have only one thread executing it at once.
`neg`	Negative; the fragment shows an invalid interaction.
`ref`	Reference; refers to an interaction defined on another diagram. The frame is drawn to cover the lifelines involved in the interaction. You can define parameters and a return value.
`sd`	Sequence diagram; used to surround an entire sequence diagram, if you wish.

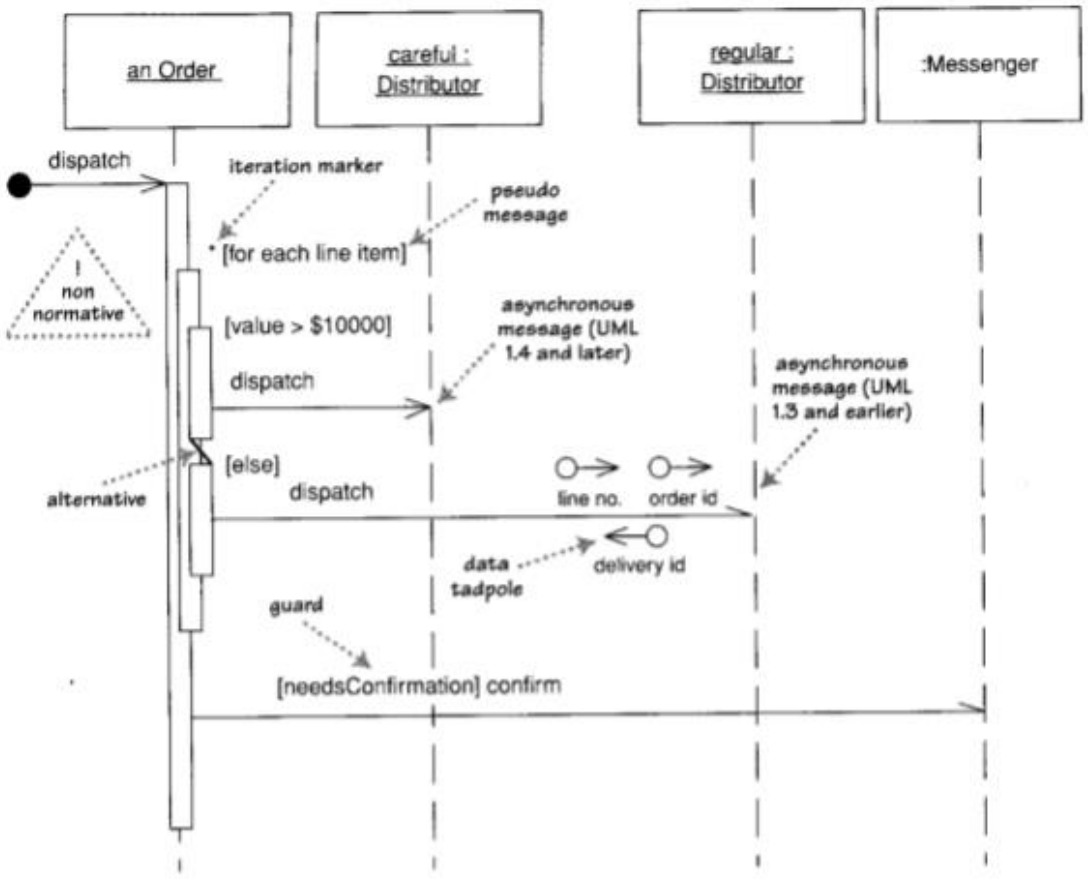

Figure 4.5 *Older conventions for control logic*

To get around this last problem, an unofficial convention that's become popular is to use a pseudomessage, with the loop condition or the guard on a variation of the self-call notation

Synchronous and Asynchronous Calls

If a caller sends a synchronous message, it must wait until the message is done, such as invoking a subroutine. If a caller sends an asynchronous message, it can continue processing and doesn't have to wait for a response

When to Use Sequence Diagrams

You should use sequence diagrams when you want to look at the behavior of several objects within a single use case. Sequence diagrams are good at showing collaborations among the objects; they are not so good at precise definition of the behavior.

Chapter 5

Class diagrams: Advanced Concepts

Responsibilities

Often, it's handy to show responsibilities (page 63) an a dass in a dass diagram. The best way to show them is as comment strings in their own compartment in the dass (Figure 5 .1).

Static Operations and Attributes

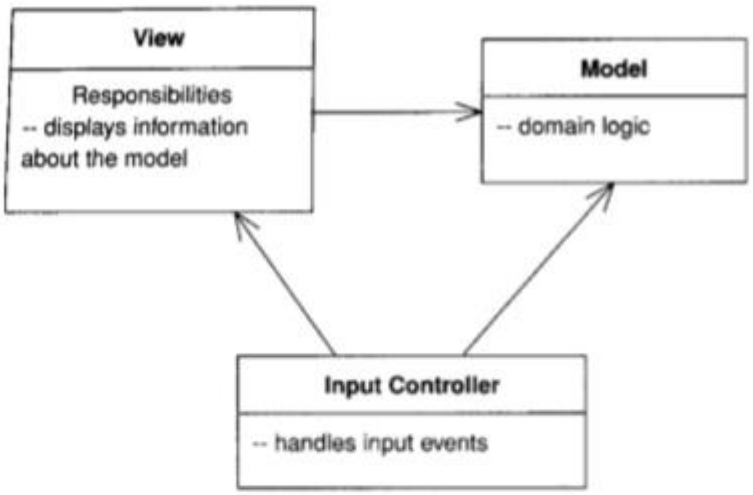

Figure 5.1 *Showing responsibilities in a class diagram*

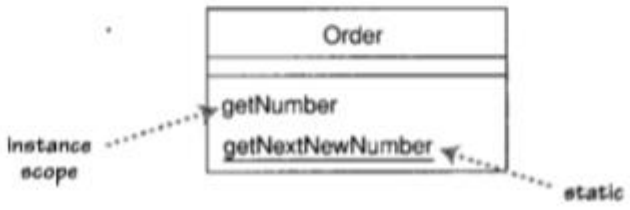

Figure 5.2 *Static notation*

Aggregation and Composition

Figure 5.3 *Aggregation*

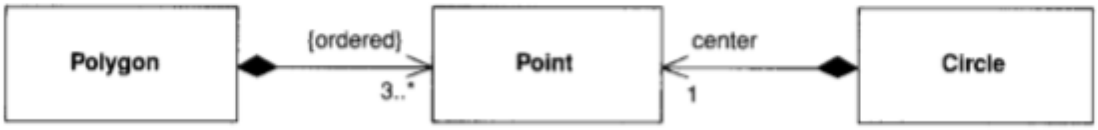

Figure 5.4 *Composition*

Composition is a good way of showing properties that own by value, properties to ;alue objects (page 73), or properties that have a strong and somewhat exclusive ownership of particular other components . Aggregation is strictly meaningless; as a result, I recommend that you ignore lt in your own diagrams. If you see lt in other people's diagrams, you'11 need to dig deeper to find out what they mean by it. Different authors and teams use lt for very different purposes.

Derived Properties

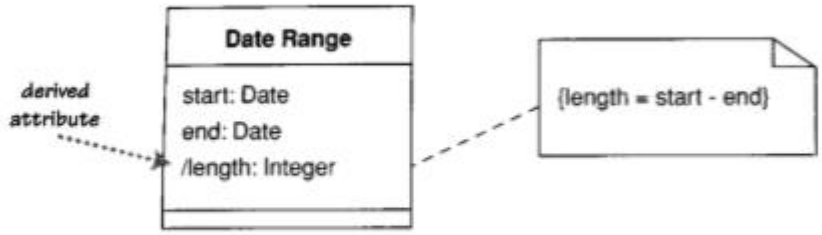

Figure 5.5 *Derived attribute in a time period*

Derivation in software perspectives can be interpreted in a couple of different ways. You can use derivation to indicate the difference between a calculated value and a stored value. In this case, we would interpret Figure 5.5 as indicating that the starr and end are stored but that the length is computed.

Interfaces and Abstract Classes

An abstract class is a class that cannot be directly instantiated. Instead, you instantiate an instance of a subclass. Typically, an abstract class has one or more operations that are abstract. An abstract operation has no implementation; it is pure declaration so that clients can bind to the abstract class.

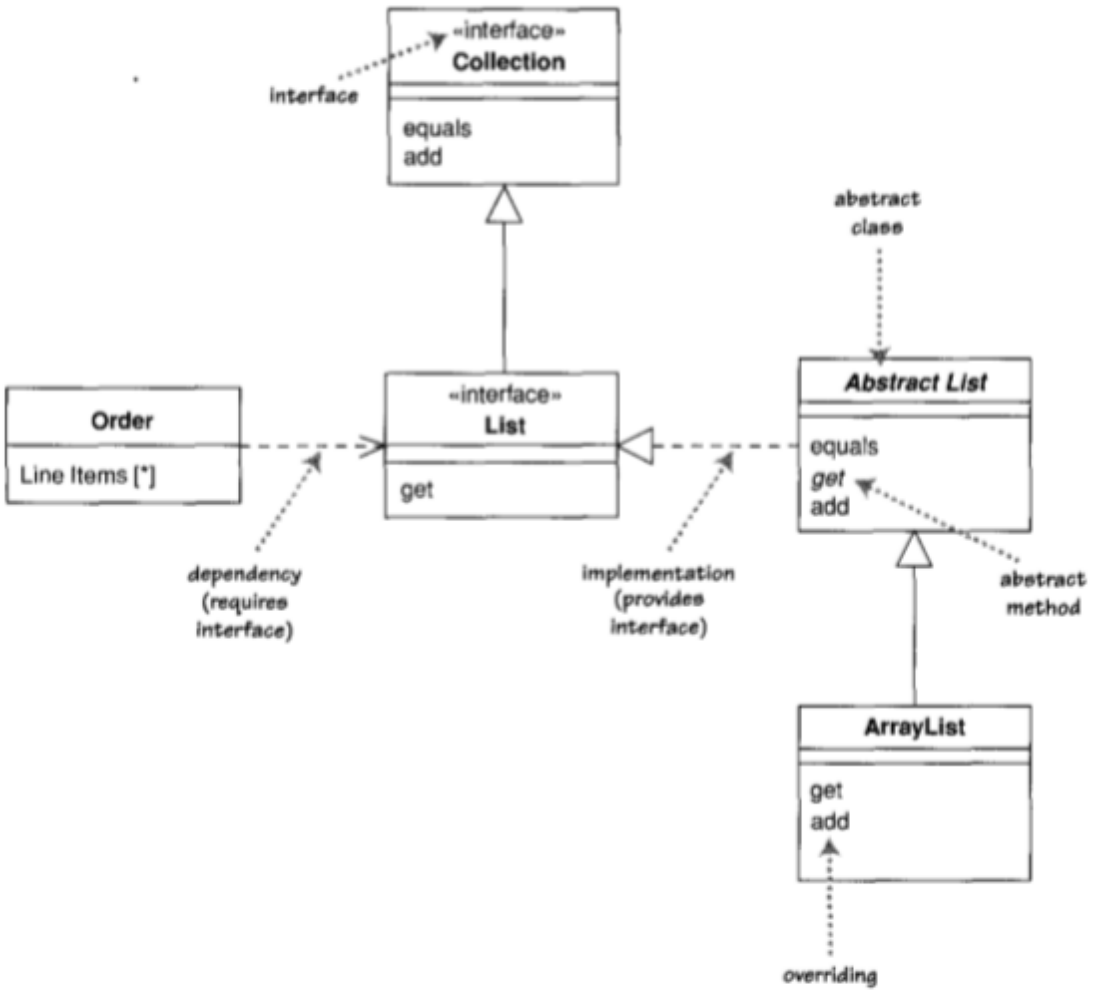

Figure 5.6 *A Java example of interfaces and an abstract class*

Read-Only and Frozen

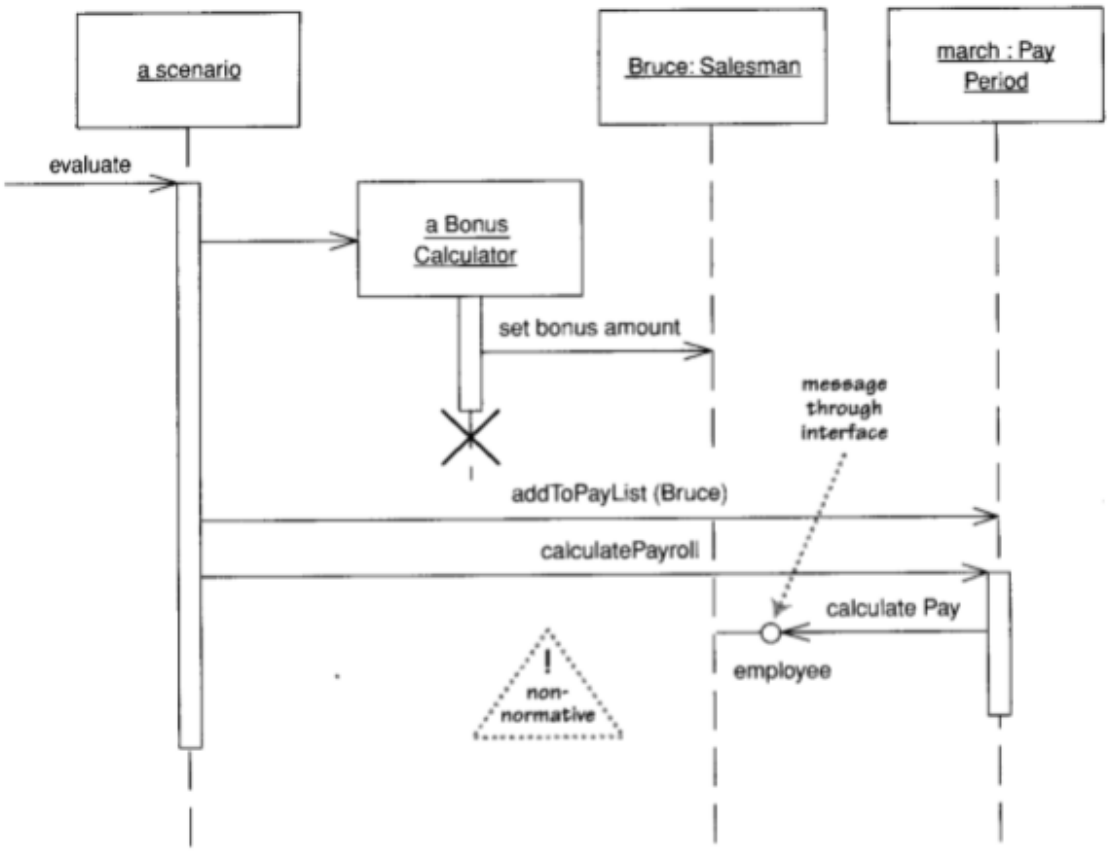

Figure 5.9 *Using a lollipop to show polymorphism in a sequence diagram*

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.

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. New Bates are created and destroyed frequently.

Qualified Associations

A qualified association is the UML equivalent of a programming concept variously known as associative arrays, maps, hashes, and dictionaries.

Figure 5.10 *Qualified association*

Classification and Generalization

Consider the following phrases.

1. Shep is a Border Collie.
2. A Border Collie is a Dog.
3. Dogs are Animals.
4. A Border Collie is a Breed.
5. Dog is a Species.

Why can I combine some of these phrases and not others? The reason is that some are classification-the object Shep is an instance of the type Border Collie-and some are generalization-the type Border Collie is a Subtype of the type Dog. Generalization is transitive; classification is not. I can combine a classification followed by a generalization but not vice versa.

Multiple and Dynamic Classification

In single classification, an object belongs to a single type, which may inherit from supertypes. In multiple classification, an object may be described by several types that are not necessarily connected by inheritance.

Multiple classification is different from multiple inheritance . Multiple inheritance says that a type may haue mang supertypes but that a single type muss be defined for each object. Multiple classification allows multiple types for an object without defining a specific type for the purpose.

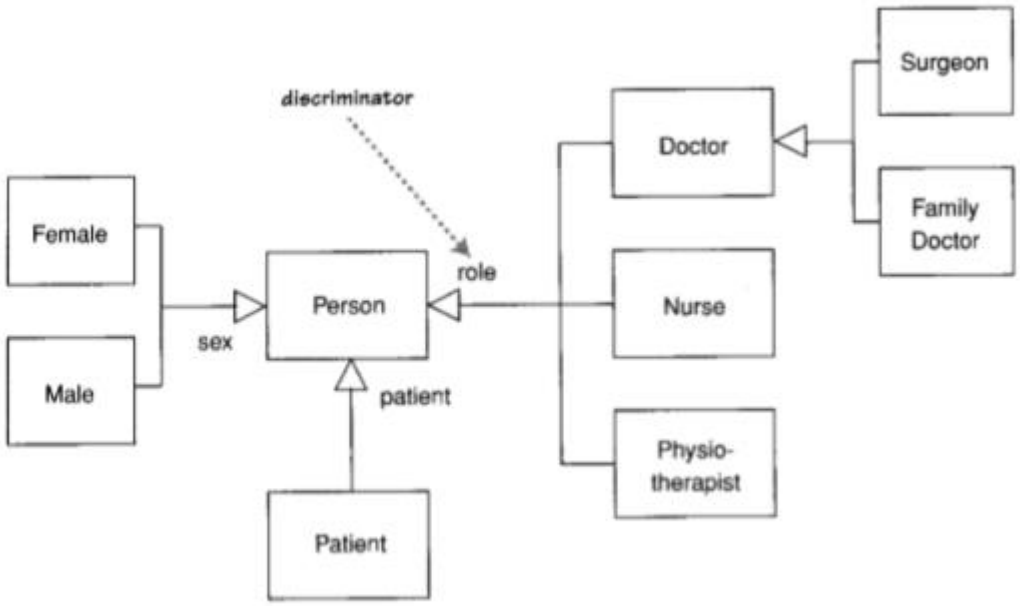

Figure 5.11 *Multiple classification*

Dynamic classification allows objects to change class within the subtyping structure; static classification does not. With static classification, a separation is made between types and states; dynamic classification combines these notions.

Association Class

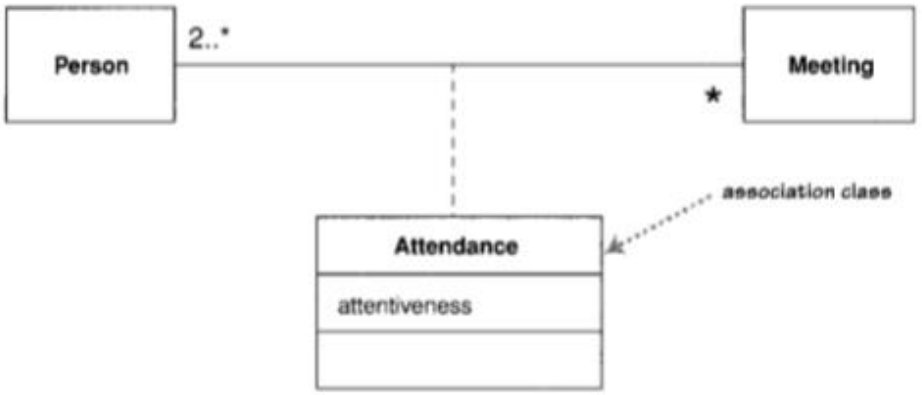

Figure 5.12 *Association class*

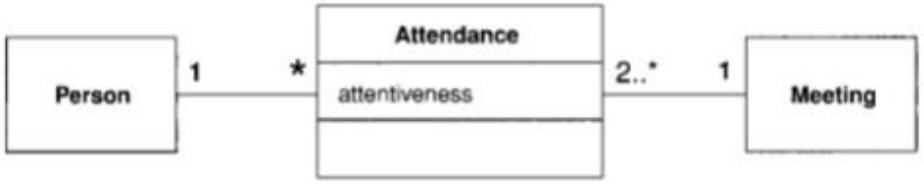

Figure 5.13 *Promoting an association class to a full class*

Template (Parameterized) Class

This concept is most obviously useful for working with collections in a strongly typed language. This way, you can define behavior for sets in general by defining a template class Set. A use of a parameterized class, such as Set<Employee>, is called a derivation.

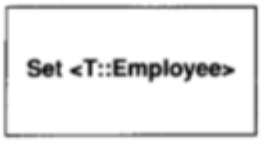

Figure 5.18 *Bound element (version 1)*

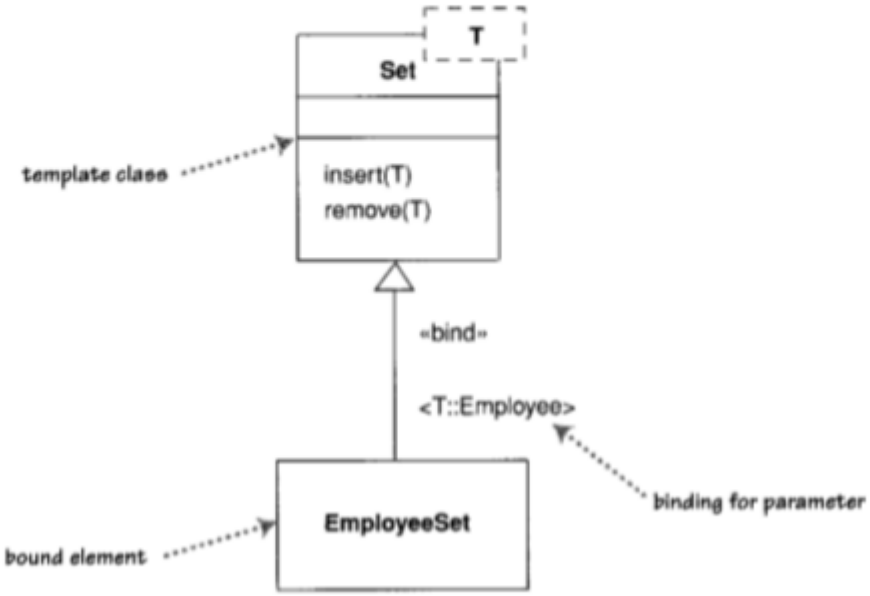

Figure 5.19 *Bound element (version 2)*

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. They are shown as the dass with the «enumeration» keyword.

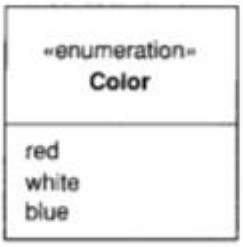

Figure 5.20 *Enumeration*

Active Class

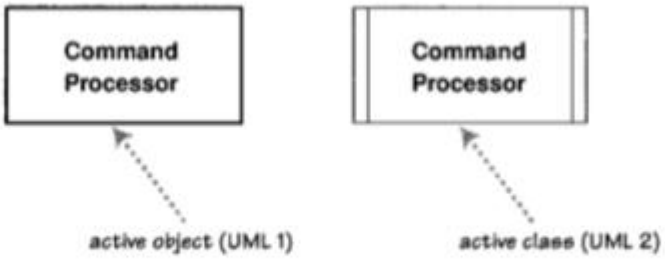

Figure 5.21 *Active class*

Messages

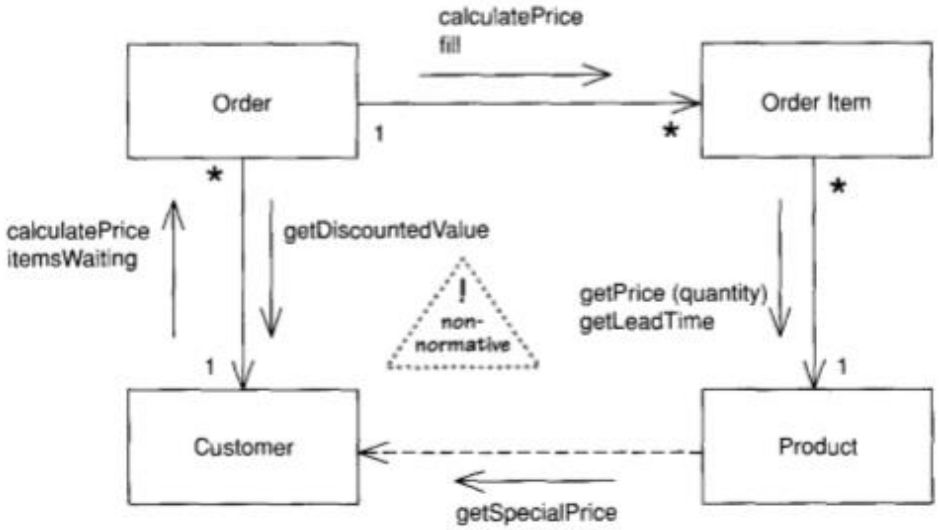

Figure 5.22 *Classes with messages*

Chapter 10:

State Machine Diagrams

State machine diagrams are a familiar technique to describe the behavior of a System.

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.

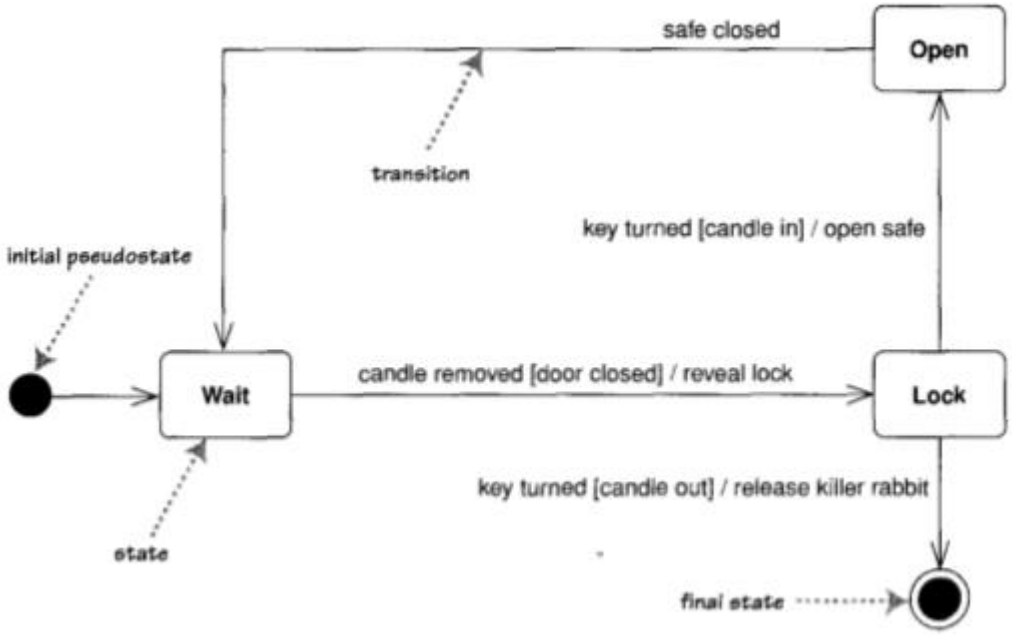

Figure 10.1 *A simple state machine diagram*

Internal Activities

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

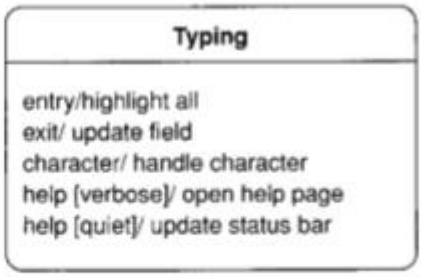

Figure 10.2 *Internal events shown with the typing state of a text field*

Activity States

In the states I've described so far, the object is quiet and waiting für the next event before it does something.

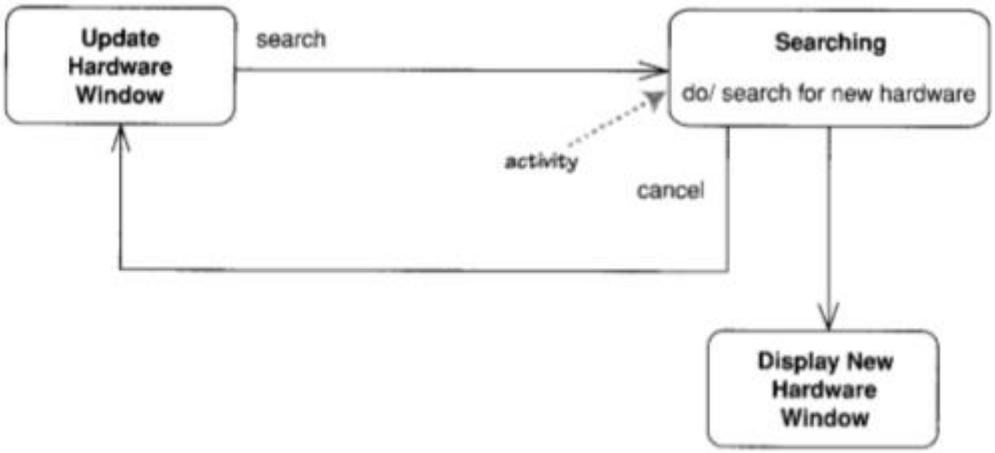

Figure 10.3 *A state with an activity*

Superstates

Without the superstate, you would have to draw a cancel transition for all three states within the Enter Connection Details state.

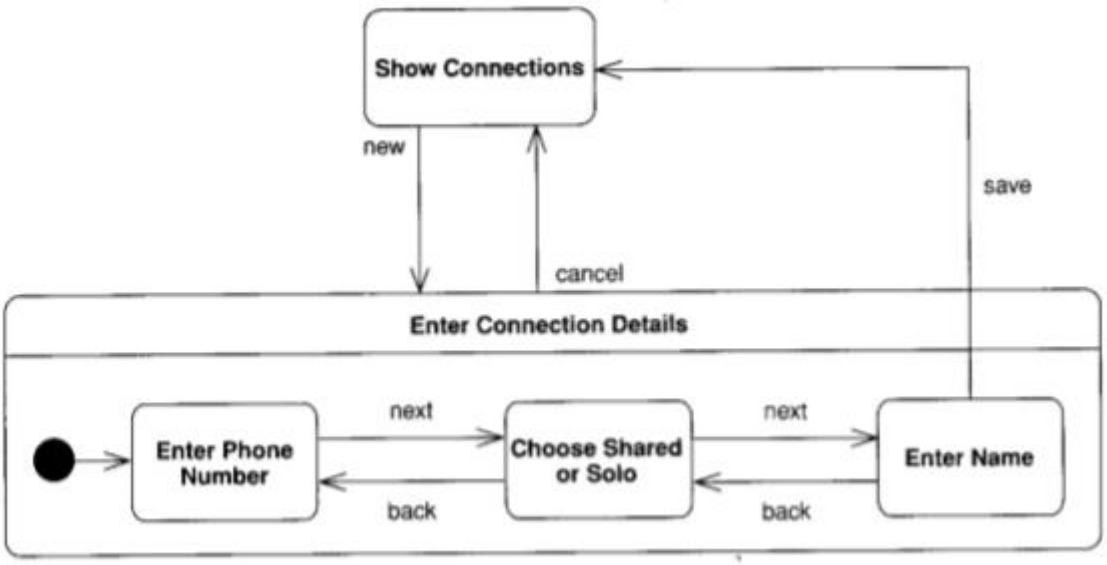

Figure 10.4 *Superstate with nested substates*

Concurrent States

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

Implementing State Diagrams

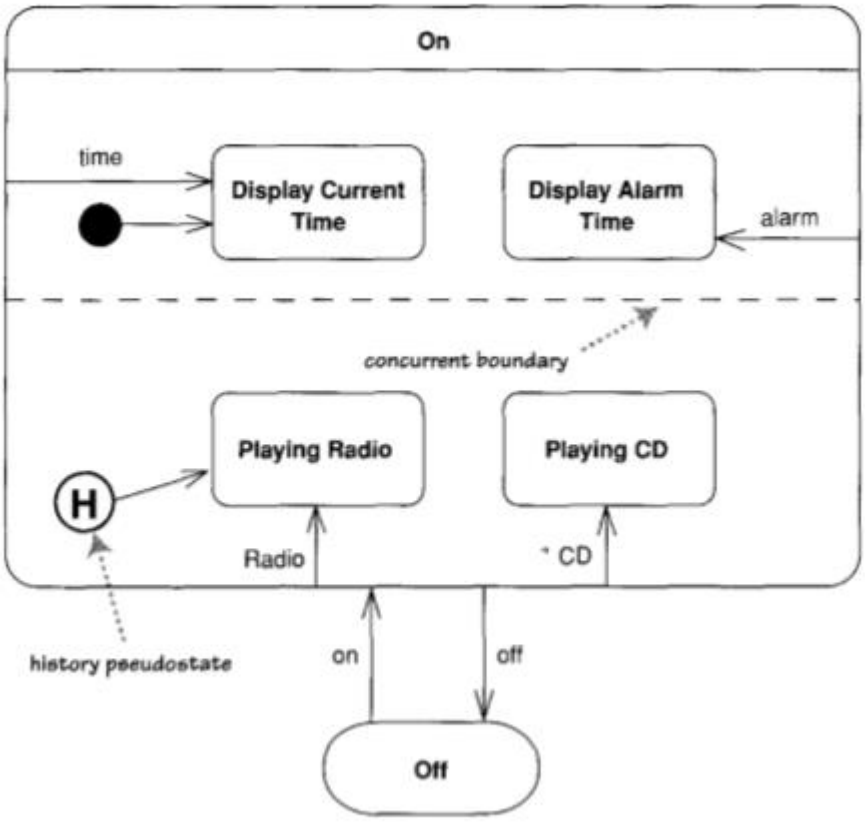

Figure 10.5 *Concurrent orthogonal states*

The State pattern [Gang of Four] creates a hierarchy of state classes to handle behavior of the states. Each state in the diagram has one state subdass.

The state table approach captures the state diagram Information as data. So Figure 10.1 might end up represented in a table like Table 10 .1.

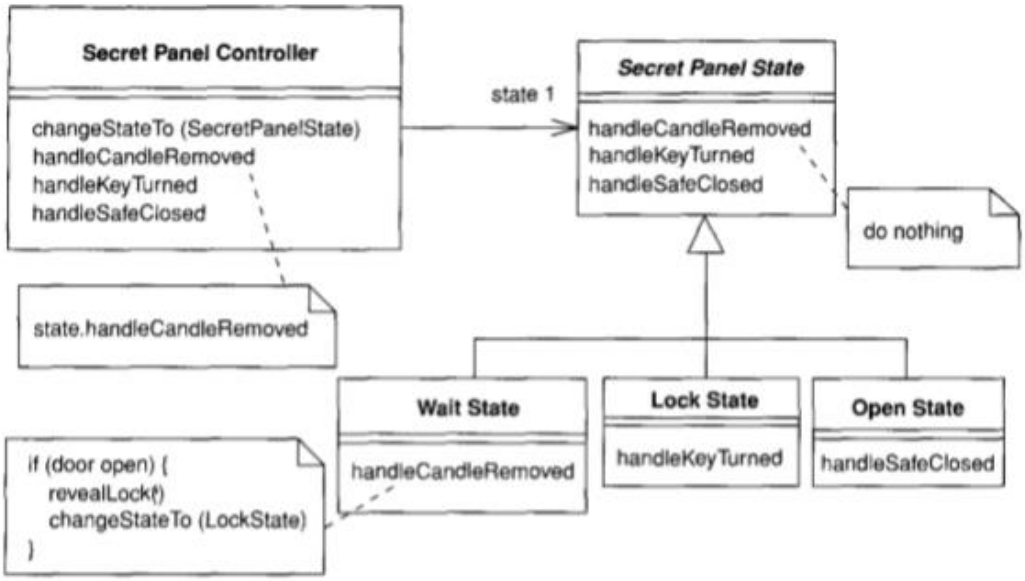

Figure 10.7 *A State pattern implementation for Figure 10.1*

Table 10.1 *A State Table for Figure 10.1*

Source State	Target State	Event	Guard	Procedure
Wait	Lock	Candle removed	Door open	Reveal lock
Lock	Open	Key turned	Candle in	Open safe
Lock	Final	Key turned	Candle out	Release killer rabbit
Open	Wait	Safe closed		

When to Use State Diagrams

Stare diagrams are good at describing the behavior of an object across several use cases. Stare diagrams are not very good at describing behavior that involves a number of objects collaborating. As such, it is useful to combine state diagrams with other techniques. For instance, interaction diagrams (see Chapter 4) are good at describing the behavior of several objects in a single use case, and activity diagrams (see Chapter 11) are good at showing the general sequence of activities for several objects and use cases.