Chapter 3

3 Objects and Object Modeling

Virtually all modern software systems are object-oriented and are developed using object-oriented modeling. The omnipresence of objects in information systems places knowledge demands on virtually all stakeholders of software projects – not just developers, but also customers (users and system owners). To be able to communicate effectively, all stakeholders must have common understanding of the object technology and the object modeling language. For customers, the knowledge of objects must be just sufficient for understanding main concepts and modeling constructs. For developers, the knowledge must be in-depth and at the level where it can be applied to build models and implement the software.

The main difficulty in learning *object technology* relates to the absence of an obvious starting point and the lack of a clear path of investigation. There is not a top-down or bottom-up learning approach that we know of. By necessity, the approach has to be a sort of 'middle-out'. No matter how much we advance the learning process, we seem always to be in the middle of that learning (as new issues keep emerging). The first major test for the successful learning process is passed when the reader understands the in-depth meaning of the fact that in an object-oriented system 'everything is an object.'

The concepts of object technology are best explained using visual representations of the Unified Modeling Language (UML) – the standard object-oriented *modeling language* of today (OMG, 2004). The strength of UML is in providing the whole set of integrated modeling techniques that allow to build multiple models representing complementary views on the system. Accordingly, this chapter uses UML in two ways – to represent all fundamental concepts of object technology and to explain models and modeling that can be done by means of UML.

3.1 Fundamentals of object technology

A good way to explain object-orientation in information systems is by providing an analogy with real-life concrete objects. The world around us consists of *objects* in some observable *states* (determined by current values of the objects' attributes) and exhibiting some *behavior* (determined by operations (functions) performed by these objects). Each object is uniquely *identified* among other objects.

For example, a coffee mug on my desk is in a `filled` *state* because it is shaped to hold liquids and there is still coffee in it. When there is no more coffee in it, the state of the mug can be defined as `empty`. If it falls on the floor and breaks, it will be in a `broken` state.

My coffee mug is rather passive – it does not have *behavior* of its own. However, the same cannot be said of my dog or a eucalyptus tree outside my window. My dog barks, the tree grows, etc. So, some real-life objects have behavior.

All real-life objects have also *identity* – a fixed property by which we identify one object from another. If I had two coffee mugs on my desk from the same mug set, I could say that the two mugs are *equal* but *not identical*. They are equal because they have the same state – the same values for all their attributes (so, they are the same size and shape, are black, and are empty). However, in object-oriented parlance, they are not identical because there are two of them and I have a choice of which one to use.

Real-life objects that possess the three properties (state, behavior, identity) build up *natural behavioral systems*. Natural systems are by far the most *complex systems* that we know. No computer system has come close to the inherent complexity of an animal or a plant.

Despite their complexity, natural systems tend to work very well – they exhibit interesting behavior, can adjust to external and internal changes, can evolve over time, etc. The lesson is obvious. Perhaps we should construct *artificial systems* by emulating the structure and behavior of natural systems (cp. Maciaszek *et al.*, 1996b).

Artificial systems are models of reality. A coffee mug on my computer screen is as much a model of the real 'thing' as is a dog or a eucalyptus tree on my screen. A coffee mug can, therefore, be modeled with behavioral properties. It can, for example, fall on the floor if knocked over. The 'fall' action can be modeled as a behavioral *operation* of the mug. Another consequential operation of the mug may be to 'break' when hitting the floor. Most, if not all, objects in a computer system 'come alive' – they have behavior.

3.1.1 Instance object

An object is an *instance* of a 'thing.' It may be one of the many instances of the same 'thing.' My mug is an instance in the set of possible mugs.

A generic description of a 'thing' is called a *class*. Hence, an object is an instance of a class. But, as discussed later in this chapter, a class itself may also need to be instantiated – it may be an object. For this reason, we need to distinguish between an *instance object* and a *class object*.

For brevity, an instance object is frequently called an *object* or an *instance*. It is confusing to call it an 'object instance.' Likewise, it is confusing to use the term 'object class.' Yes, a class is a template for objects with the same attributes and operations, but a class itself can be instantiated as an object (and it would be strange to call such a creation an 'object class object').

An object-oriented system consists of collaborating objects. Everything in an object-oriented system is an object, be it an object of an instance (*instance object*) or an object of a class (*class object*).

3.1.1.1 Object notation

The UML notation for an object is a rectangle with two compartments. The upper compartment contains the name of an object and the name of a class to which the object belongs. The syntax is:

```
objectname: classname
```

The lower compartment contains the list of attribute names and values. The types of attributes can also be shown. The syntax is:

`attributename: [type] = value`

Figure 3-1 demonstrates four different ways that an object can be shown graphically. The example shows a `Course` object named `c1`. The object has two attributes. The types of the attributes are not shown – they have been specified in the definition of the class. The attribute value compartment may be suppressed if not required to represent a particular modeling situation. Similarly, the object name may be omitted when representing an anonymous object of the given class. Finally, the class of the object may be suppressed. The presence or absence of the colon indicates if the given label represents a class or an object.

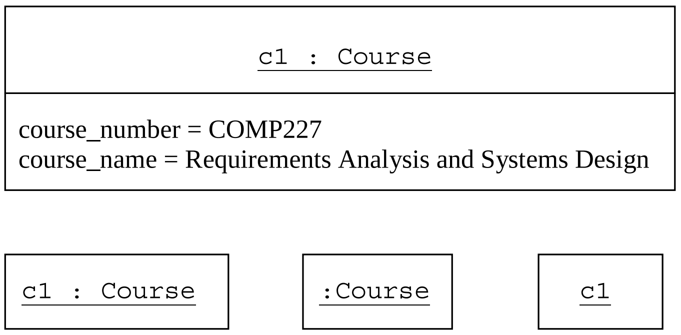


Figure 3-1 Instance object

It is important to note that the object notation does not provide a 'compartment' for listing the *operations* that an instance object can execute. This is because the operations are identical for all instance objects and it would be redundant to store them repeatedly in each instance object. Operations may be stored in a *class object* or they may be associated with instance objects by other means (implemented in the underlying object-oriented system software).

As an aside, there are some less popular programming languages, such as Self, that allow attaching operations to objects (not just classes) at run time. These languages are known as *prototypical* or *delegation-based* languages. For such situations, UML allows a third compartment containing operations as a language-specific extension (UML, 2003).

3.1.1.2 How do objects collaborate?

The number of objects of a particular class can be very large. This is true in particular for *business objects* representing business concepts (known as *entity classes*, such as `Invoice` or `Employee`). It is impractical and infeasible to visualize many objects on a diagram.

Objects are normally drawn to only exemplify a system at a point in time or to exemplify how they *collaborate* over time to do certain tasks. For example, to order products a *collaboration* may need to be established between a `Stock` object and a `Purchase` object. To be precise, objects on collaboration diagrams are so called *roles*, not objects – they describe many possible objects. Graphically, the roles are presented using the notation of anonymous objects.

System tasks are performed by sets of objects that invoke *operations* (behavior) on each other. We say that they exchange *messages*. The messages call operations on objects that can result in the change of objects' states and can invoke other operations.

Figure 3-2 shows the flow of messages between four objects. The brackets after the message names indicate that a message can take parameters (like in a traditional programming call to a function). The object `Order` requests the object `Shipment` to "ship itself" (to this aim, `Order` passes itself to `Shipment` as an actual parameter to `shipOrder()`). `Shipment` asks `Stock` in `getProducts()` message to provide appropriate quantity of products. At this point, `Stock` analyzes its inventory levels for products to be shipped by performing `analyzeLevels()`. If the stock requires replenishment, `Stock` requests `Purchase` to reorder more products in `reorder()` message.

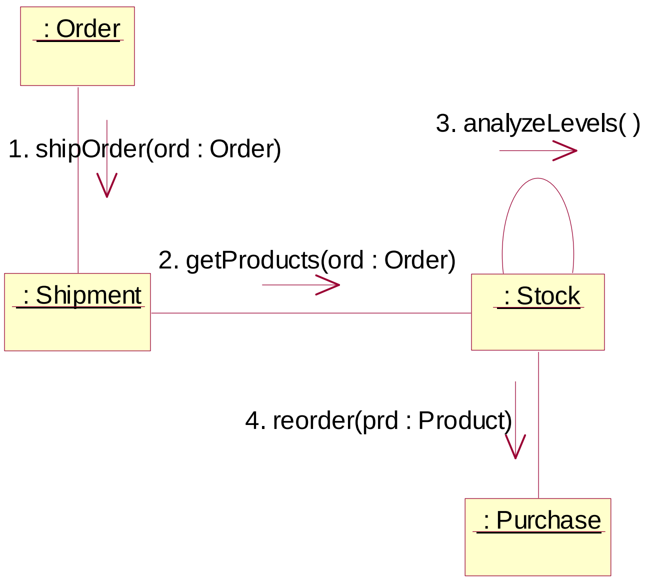

: Order
1. shipOrder(ord : Order)
3. analyzeLevels( )
2. getProducts(ord : Order)
: Shipment
: Stock
4. reorder(prd : Product)
: Purchase

Figure 3-2 Object collaboration

Although the object collaboration model in Figure 3-2 is shown as a sequence of numbered messages, in general the flow of messages does not impose a strict temporal order on the activation of objects. For example, `analyzeLevels()` can be activated independently of `shipOrder()` and `getProducts()`, not in a way related to shipments. For these reasons, the numbering of messages is frequently not used in object collaboration models.

3.1.1.3 Identity and object communication

The question is how an object knows the *identity* of another object to which it wants to send a message. How does an `Order` object know the `Shipment` object so that the message `shipOrder()` reaches its destination?

The answer is that each object is given an *object identifier* (OID) when it is created. The OID is the *handle* on object – a unique number that remains with the object for its entire life. If an object `X` wants to send a message to an object `Y` then `X` has to somehow know the OID of Y.

There are two practical solutions to establishing OID *links* between objects. The solutions involve:

- persistent OID links; and
- transient OID links.

The distinction between these two kinds of links has to do with the longevity of objects. Some objects live only as long as the program executes – they are created by the program and destroyed during the program execution or when the program finishes its execution. These are *transient objects*. Other objects outlive the execution of the program – they are stored in the persistent disk storage when the program finishes and they are available for the next execution of the program. These are *persistent objects*.

3.1.1.3.1 Persistent link

A *persistent link* is an object reference (or a set of object references) in one object in persistent storage that links that object to another object in persistent storage (or to the set of other objects). Hence, to persistently link a `Course` object to its `Teacher` object, the object `Course` must contain a link attribute the value of which is the OID of the object `Teacher`. The link is persistent because the OID is physically stored in the object `Course`, as shown in Figure 3-3.

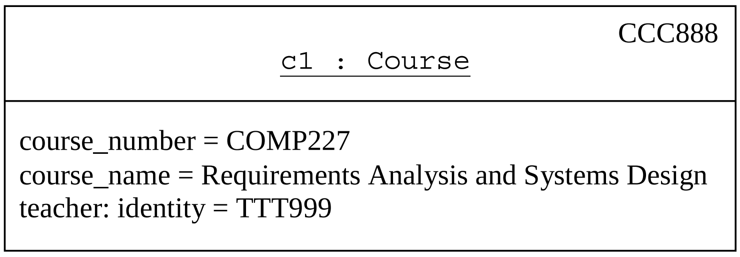

CCC888
c1 : Course
course_number = COMP227
course_name = Requirements Analysis and Systems Design
teacher: identity = TTT999

Figure 3-3 Representation of a persistent link

The OID of object `c1` is marked here as `CCC888`. The object contains a link attribute named `teacher`. The type of this attribute is `identity`. Its value is the OID of a `Teacher` object, shown here as `TTT999`. The OID is a logical address of the object. This can implemented as the computer identification number plus the time in milliseconds when the object was instantiated. The programming language environment is able to convert the logical address to the physical disk address where the object `Teacher` is *persistently* stored.

Once the objects `Course` and `Teacher` are transferred to the program's memory, the value of the `teacher` attribute will be *swizzled* to a memory pointer, thus establishing a memory-level collaboration between the objects. (Swizzling is not the UML term – it is used in object databases where the transfers of objects between the persistent storage and the transient memory are frequent.)

Figure 3-3 illustrates how persistent links are represented in objects. In UML modeling, the links between objects can be drawn as in Figure 3-4. The links are represented as *instances of an association* between the objects `Course` and `Teacher`.

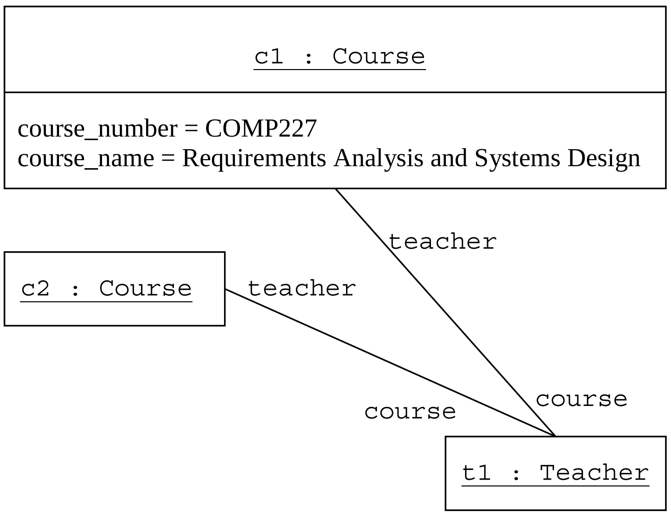


Figure 3-4 Persistent links in UML object model

Normally, collaboration links on business objects allow for *navigation* in both directions. Each `Course` object is linked to its `Teacher` object, and a `Teacher` object can navigate to `Course` objects. It is possible, though not frequent in case of business objects, to allow only for navigation in one direction.

3.1.1.3.2 Transient link

What if no persistent link is defined between `Course` and `Teacher`, and there is still a need to send a message from object `t1` to object `c1` to invoke the operation `getCourseName()`? The application program must have other means to find out the identity of object `c1` and create a *transient link* from object `t1` to object `c1` (Riel, 1996).

Fortunately, a programmer has many techniques that can lead to initialization of the variable `crs_ref` with an OID of memory-resident object `c1`. To start with, it is possible that earlier in the program a link between objects `c1` and `t1` has been established and `crs_ref` still holds the correct OID. For example, the program has executed a search operation on the teacher's `t1` availability and on the timetable for courses and has determined that teacher `t1` should teach course `c1`.

An alternative possibility is that the program has access to a persistently stored table that maps course numbers to teacher names. It can then search on `Course` objects to find all courses taught by the teacher `t1` and request the user to determine the course that the message `getCourseName` is to be sent to.

It is also possible that the very task of a program is to create courses and teachers before storing them in a database. There is no persistent link between teachers and courses, but the user enters the information so that each course clearly identifies a teacher in charge. The program can then store the transient links in the program's variables (such as `crs_ref`) and these variables can be later used (during the same program execution) to send messages between `Teacher` and `Course` objects.

In short, there are quite a few ways to establish transient links between objects that are not persistently linked by associations between relevant classes. *Transient links* are program variables that contain OID values of objects that are currently in the program's memory. The mapping (*swizzling*) between the transient and persistent OIDs should be the responsibility of the underlying programming environment, such as an object database system.

3.1.1.3.3 Message passing

Once an object is linked to another object, it can send a message along the link to request a *service* from the other object. That is, an object can invoke an *operation* on another object by sending a *message* to it. In a typical scenario, to point to an object, the sender will use a program's variable containing a link value (OID value) of that object.

For example, a *message* sent by an object `Teacher` to find the name of an object `Course` could look like:

```
crs_ref.getCourseName(out crs_name)
```

In the example, the specific object of class `Course` that will execute `getCourseName` is pointed to by the current value of the link variable `crs_ref`. The output (`out`) argument `crs_name` is a variable to be initialized with the value returned by the operation `getCourseName` implemented in the class `Course`.

The example assumes that the programming language distinguishes between input (`in`), output (`out`), and input/output (`in out`) arguments. Popular object-oriented languages, such as Java, do not make such distinction. In Java, message arguments of primitive data types (such as `crs_name` in the example) are passed to operations by value. "Pass by value" means that these are effectively input arguments – the operation cannot change the value passed. The change is not possible because the operation acts on a copy of the argument.

In case of message arguments of non-primitive data types (i.e. arguments which are references to user-defined objects), "pass by value" means that the operation receives the reference to the argument, not its value. Because there is only one reference (not two copies of it), the operation can use it to access and possibly modify the attribute values within the passed object. This in effect eliminates the need for explicit input/output arguments.

Finally, the need for explicit output arguments is substituted in Java by a return type of an operation invoked by a message call. The return type may be primitive or non-primitive. An operation can return the maximum of one return value or no value at all (i.e. a return type of `void`). When multiple values need to be returned, the programmer has an option of defining a non-primitive aggregate object type that can contain all the values to be returned.

3.1.2 Class

A *class* is the descriptor for a set of objects with the same attributes and operations. It serves as a *template* for object creation. Each object created from the template contains the attribute *values* that conform to attribute *types* defined in the class. Each object can invoke operations defined in its class.

Graphically, a class is represented as a rectangle with three compartments separated by horizontal lines, as shown in Figure 3-5. The top compartment holds the class name. The middle compartment declares all attributes for the class. The bottom compartment contains definitions of operations.

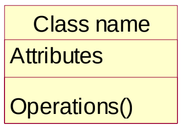


Figure 3-5 Class compartments

3.1.2.1 Attribute

An attribute is the *type-value* pair. Classes define *attribute types*. Objects contain *attribute values*. Figure 3-6 illustrates two classes with attribute names and attribute types defined. Two different attribute naming conventions are shown. The attribute names in `Course` use the notation popularized by the database community. The programming language community prefers the convention shown in `Order`.

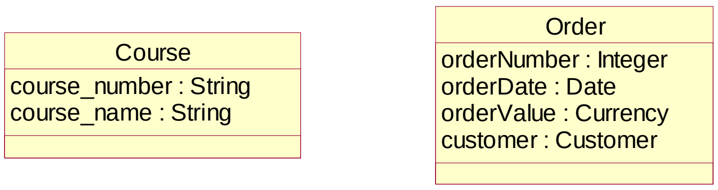


Figure 3-6 Attributes

An attribute type can be a built-in *primitive type* or it can be a user-defined *non-primitive type*. A primitive type is the type directly understood and supported by an underlying object-oriented software environment. All attribute types in Figure 3-6, except `customer`, designate primitive types. The type of `customer` is a *class* (non-primitive type). Note, however, that in UML analysis models, non-primitive attribute types are not visualized within the Attributes compartment (this is discussed next).

3.1.2.1.1 Attribute type that designates a class

An attribute type can also designate a class. In a particular object of the class, such an attribute contains an object identifier (OID) value pointing to an object of another class. In UML analysis models, attributes with class-based types (rather than primitive types) are not listed in the middle class compartment. Instead, the *associations* between classes represent them. Figure 3-7 shows such an association between two classes.

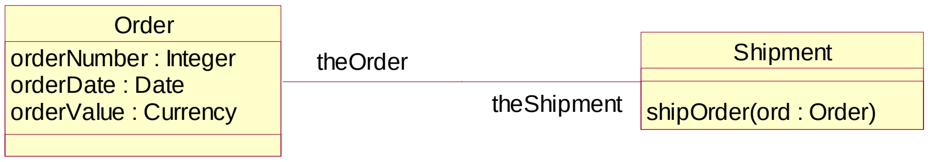


Figure 3-7 Rolenames that designate classes (analysis model)

The two names on the association line (`theShipment` and `theOrder`) represent so-called role names. A *role name* identifies the meaning for the association end and it is used to *navigate* to an object of the other class in the association.

In the implemented system, the role name (on the opposite end of the association) becomes a class attribute whose type is the class pointed to by the role name. This means that an attribute represents an association to another class. Figure 3-8 shows two classes from Figure 3-7 as eventually implemented.

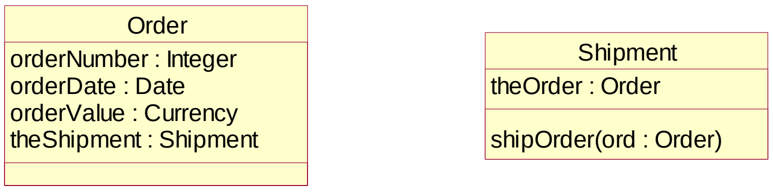


Figure 3-8 Attributes that designate classes (implementation model)

3.1.2.1.2 Attribute visibility

As explained in Section 3.1.1.2, objects collaborate by sending messages to each other. A message invokes a class operation. The operation services the calling object's request by accessing attribute values in its own object and, if necessary, by sending messages to other objects. For this scenario to be possible, the operations must be *visible* to the outside objects (messages must see the operations). Such operations are said to have *public visibility*.

In a well-designed and implemented object-oriented system, most operations are *public* but most attributes are *private*. Attribute values are hidden from other classes. Objects of one class can only request the services (operations) published in the public interface of another class. They are not allowed to directly manipulate other objects' attributes.

It is said that operations *encapsulate* attributes. Note, however, that the encapsulation applies to classes. One object cannot hide (encapsulate) anything from another object of the same class.

The visibility is normally designated by a plus or minus symbol:

- + for public visibility
- – for private visibility

These symbols are replaced in some CASE tools by graphical icons. Figure 3-9 demonstrates two graphical representations to signify attribute visibility. The graphical icon of a lock designates the private visibility.

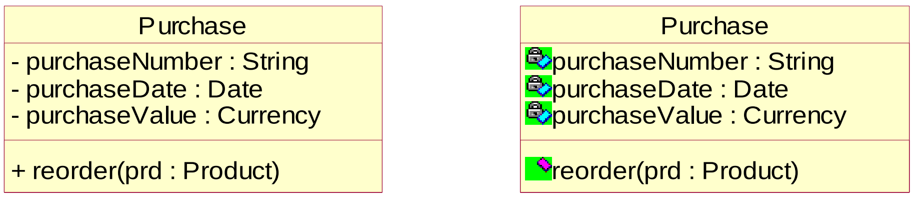


Figure 3-9 Private attributes and public operations

3.1.2.2 Operation

An object contains data (attributes) and algorithms (operations) to act on these data. An operation is declared in a class. A procedure that implements the operation is called a *method*.

The operation (or the method, to be precise) is invoked by a message sent to it. The name of the message and the name of the operation are the same. The operation can contain a list of formal arguments (parameters) that can be set to specific values by means of actual arguments in the message call. The operation can return a value to the calling object.

The operation name together with a list of formal argument types is called the *signature* of an operation. The signature must be unique within a class. This means that a class may have many operations with the same name, provided that the lists of parameter types vary.

3.1.2.2.1 Operations in object collaboration

An object-oriented program executes by reacting to random events from the user. The events come from the keyboard, mouse clicks, menu items, action buttons, and any other input devices. A user-generated *event* converts to a *message* sent to an *object*. To accomplish a task many objects may need to collaborate. Objects collaborate by invoking *operations* in other objects (Section 3.1.1.2).

Figure 3-10 shows operations in classes necessary to support the object collaboration demonstrated in Figure 3-2. Each message in Figure 3-2 requires an operation in the class designated by the message's destination.

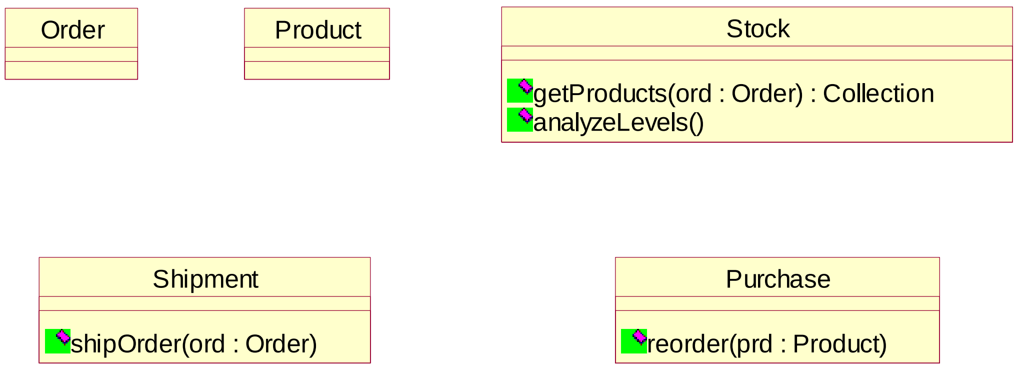


Figure 3-10 Operations in object collaboration

The classes `Order` and `Product` (the latter not shown in Figure 3-2) do not have any operations in this simple example. The `Order` is initiated by the `Order` object when it requests that a `Shipment` object ship it. As a result of the shipment, the stock may need to be replenished with new products.

The `getProducts()` operation demonstrates the return type of `Collection`. This is the collection (set, list, or something similar) of products returned by `Stock` to `Shipment`. The `Collection` type is provided by the programming language. The Java collection comes from the library `java.util.Collection`.

3.1.2.2.2 Operation visibility and scope

The visibility of an operation defines whether the operation is visible to objects of classes other than a class that defines the operation. If it is visible then its visibility is *public*. It is *private*, otherwise. The icons in front of operation names in Figure 3-10 denote the visibility that is public.

Most operations in an object-oriented system would have public visibility. For an object to provide a service to the outside world, the 'service' operation must be visible. However, most objects will also have a number of internal housekeeping operations. These will be given private visibility. They are only accessible to objects of a class in which they have been defined.

The operation visibility needs to be distinguished from an *operation scope*. The operation may be invoked on an *instance object* (Section 3.1.1) or it may be invoked on a *class object* (Section 3.1.2.3). In the former case, the operation is said to have the *instance scope*. In the latter case – the *class scope*. For example, the operation to find an employee's age has the instance scope, but the operation to calculate the average age of all employees has the class scope.

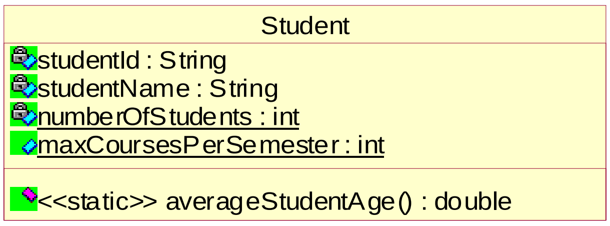
3.1.2.3 Class object

In Section 3.1.1, the distinction was made between *instance objects* and *class objects*. A *class object* is an object with class-scope attributes and/or class-scope operations. The class-scope implies here a global attribute or operation that can be accessed/called on the class itself, not necessarily on an instance object.

Note, however, that in practice, most programming languages do not implement the concept of a class object by instantiating such an object. They instead provide a syntax capability to refer to the class name in order to access a class-scope attribute or call a class-scope operation.

The most common *class-scope attributes* are attributes that hold default values or aggregate values (such as sums, counts, averages). The most common *class-scope operations* are operations to create and destroy instance objects and operations that calculate aggregate values.

Figure 3-11 shows the class `Student` with two class-scope attributes (underlined) and a class-scope operation (identified by the stereotype <<`static`>>). Note that the attribute `numberOfStudents` is private, but the attribute `maxCoursesPerSemester` is public. Every student has the same allowed maximum number of courses per semester. The operation to calculate average age of students has the class-scope because it needs to access individual ages of students (in `Student` instance objects), sum them up and divide the sum by the total number of students kept in `numberOfStudents`.



```
public class Student
{
   private String studentId;   //accessible via Student’s operations
   private String studentName; //accessible via Student’s operations
   private static int numberOfStudents;
   //accessible only to Student’s static methods, such as averageStudentAge()
   public static int maxCoursesPerSemester;
   //accessible via Student:: maxCoursesPerSemester

   public static double averageStudentAge()
   { implementation code here }
   //callable by referring to the class name – Student::averageStudentAge()
   //callable also with an object of the class – std.averageStudentAge()
}
```

Figure 3-11 Java class with class-scope attributes and operations

Figure 3-11 shows also Java code corresponding to the graphical model. Java uses the keyword `static` to distinguish between instance and class properties. In effect, Java defines two kinds of objects (instance and class objects) in one class definition (Lee and Tepfenhart, 2002).

The two instance attributes (`studentId` and `studentName`) result in their own copies (storage space) in each instance of the class. Because these two attributes have `private` visibility, they are accessible only to operations defined in `Student` class.

Class-scope (`static`) attributes are stored as single copies (occupy single storage space). In case of private static attribute (`numberOfStudents`), this single storage space is shared with all instances of `Student` and is accessible to `Student’s` static operations. In case of public static attribute (`maxCoursePerSemester`), the single storage space is shared with all instances of all classes and is accessible with the class name, i.e. `Student::numberOfStudents`.

Class scope (`static`) operations are callable, if `public`, from any instance of any class. They can be called by referring to the class name (`Student::averageStudentAge()`) or with an object of the class (e.g. `std.averageStudentAge()`).

3.1.3 Variables, methods, and constructors

The discussion so far has used, as far as possible, the generic terminology present in UML analysis models. However, to explain object implementation principles, there is a need to use the terminology present in UML design models and in programming language, such as Java. Frequently, the analysis and design terminology is the same, but sometimes the terminology differs and the mapping between corresponding analysis and design/implementation terms is not exactly one-to-one.

This section introduces the concepts of variable and method. Variable maps from the notion of attribute (variable implements an attribute). Method maps from the notion of operation (method implements an operation).

A *variable* is the name for a storage space that main contain values for a specific data type. A variable can be declared within a class or within an operation (method body) of the class. In the first case, the variable is a *data member* of the class. In the second case, the variable is not a data member – it is a *local variable*. A local variable is valid only within the scope of the method (i.e. as long as the method executes).

Data members can have instance-scope (*instance variables*) or class-scope (*class variables*) (Section 3.1.2.3). There are two categories of instance variables – those that implement *attributes* and those that implement *associations*. The former are variables with a primitive data type (they store attribute values). The latter are variables with a non-primitive data type (they store references to objects and, therefore, they implement associations).

It is important to understand that variables storing references to objects are *not* objects (lethbridge and Laganiere, 2001). During a single program execution the same variable can refer to different objects and the same objects can be referred to by different variables. A variable can also contain a `null` value, which means that it does not refer to any object at all.

Data members (instance and class variables) can be initialized to any *nonconstant* or *constant* value/object. A constant variable cannot change its value after the value has been assigned to it. Constants cannot be defined for local variables. In Java, constants are defined with the `final` keyword.

Instance variables that implement attributes can be initialized at the time they are defined within a class. Instance variables that implement associations are normally initialized programmatically, frequently in the constructor, which initializes objects of the class concerned.

A *method* is the implementation of an operation (a service) that belongs to a class (Lee and Tepfenhart, 2002). A method has the name and a *signature* – the list of formal arguments (parameters). Two methods with the same name and different signatures are considered different. Such methods are known as *overloaded* methods.

A method may *return* (to the calling object) a single value of some primitive or non-primitive type. Formally, all methods must have a return type, but the type may be `void`. The method name, its signature and its return type are together known as the method *prototype*.

A *constructor* is a special method (the purists would argue that it is not a method at all) that serves the purpose of instantiating objects of the class. Each class must have at least one constructor, but it can have more than one (i.e. constructors can be overloaded). A constructor name is the same as the name of the class for which it is declared. Constructors do not have return types.

In Java, a constructor is called with the `new` keyword, e.g.

```
        Student std22 = new Student();
```

The constructor `Student()` in the example is so called default constructor, generated automatically by Java if omitted in the class definition. The default constructor creates an object with default values assigned for all class variables. In Java, the default values are: `0` for numbers, `'0'` for characters, `false` for Booleans, and `nulls` for objects.

3.1.4 Association

An *association* is one kind of relationship between classes. Other kinds of relationships include: generalization, aggregation, dependency and a few more.

An association relationship provides a linkage between objects of given classes. Objects needing to communicate with each other can use the linkage. If possible, messages between objects should be always sent along association relationships. This has an important documentary advantage – the static compile-time structures (i.e. associations) document all possible dynamic message passing that is allowed at run-time.

Figure 3-12 shows the association named `OrdShip` between classes `Order` and `Shipment`. The association allows for an `Order` object to be shipped (to be linked to) more than one `Shipment` object (indicated by the association multiplicity of `n`). Also a `Shipment` object can carry (be linked to) more than one `Order` object.

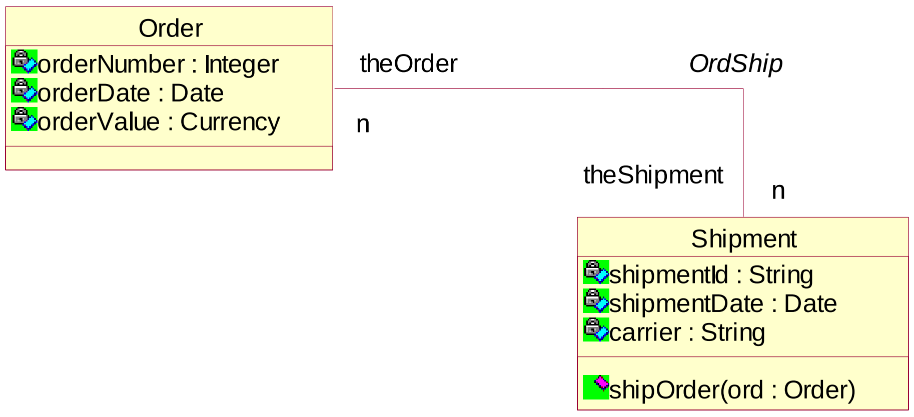


Figure 3-12 Association

In the simplest case of one-to-one association between an `Order` object and a `Shipment` object the processing scenario could be as follows. An `Order` object needs to get shipped. To this aim, it instantiates new `Shipment` object by invoking one of `Shipment's` constructors. As a result of instantiation, `Order` obtains the reference to new `Shipment` object.

Knowing this, `Order` can send the `shipOrder()` message to `Shipment`, passing itself to `Shipment` in the actual argument of `ShipOrder()`. This way, `Shipment` obtains the reference to `Order`. The only remaining action to establish the association is to assign the `Shipment` reference to the variable `theShipment` and, vice versa, the `Order` reference to the variable `theOrder`.

Typically associations on entity classes (business objects), as in Figure 3-12, are both-directional.. However, single-directional associations may be sufficient in associations on other categories of classes, such as between classes representing GUI window, programming logic, user events, etc.

3.1.4.1 Association degree

Association degree defines the number of classes connected by the association. The most frequent association is of *degree two*. This is called a *binary association*. The association in Figure 3-12 is binary.

Association can also be defined on a single class. This is called a *unary* (or *singular*) *association* (Maciaszek, 1990). The unary association establishes links between objects of a single class.

Figure 3-13 is a typical example of a unary association. It captures a hierarchical structure of employment. An `Employee` object is `managedBy` one other `Employee` object or by nobody (i.e. an employee who is, say, the Chief Executive Officer (CEO) is not managed by anybody). An `Employee` object may be the `managerOf` many employees, unless the employee is at the bottom of the employment ladder and is not managing anybody.

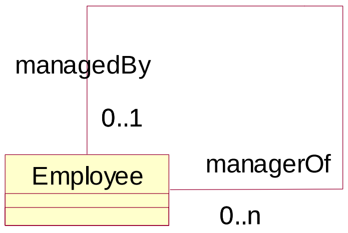


Figure 3-13 Unary association

Associations of degree 3 (*ternary associations*) are also possible, but not recommended (Maciaszek, 1990).

3.1.4.2 Association multiplicity

Association multiplicity defines how many objects may be represented by a *role name*. The multiplicity states how many objects of a target class (pointed to by the role name) can be associated with a single object of the source class.

The multiplicity is shown as a range of integers `i1..i2`. The integer `i1` defines the minimum number of connected objects, and `i2` – the maximum number (the maximum number can be shown as `n` if the precise maximum integer value is not known or is not fixed). The minimum number does not have to be specified if such information is not essential in the model at the level of abstraction applied (as in Figure 3-12).

The most frequent multiplicities are:

```
0..1
0..n
1..1
1..n
n
```

Figure 3-14 demonstrates two associations on the classes `Teacher` and `CourseOffering`. One association captures the assignment of teachers to current course offerings. The other determines which teacher is in charge of an offering. A teacher can teach many offerings or none (e.g. if a teacher is on leave). One or more teachers teach a course offering. One of these teachers is in charge of the offering. In general, a teacher can be in charge of many course offerings or none. One and only one teacher manages a course offering.

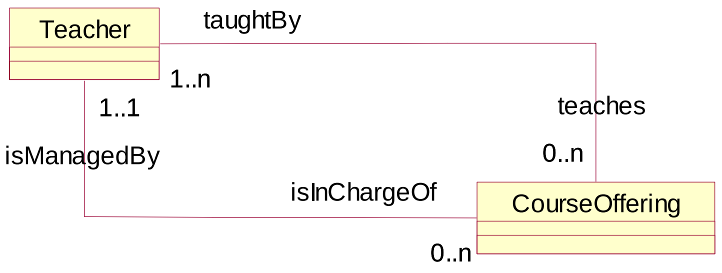


Figure 3-14 Association multiplicity

The association multiplicity in UML is an imprecise term. The 'zero' and 'one' minimum multiplicity can be seen as a distinct semantic notion of *membership* or *participation* (Maciaszek, 1990). The 'zero' minimum multiplicity signifies an *optional membership* of an object in the association. The 'one' multiplicity signifies a *mandatory membership*. For example, a `CourseOffering` object must be managed by a `Teacher` object.

The membership property has some interesting semantics of its own. For example, a particular mandatory membership may additionally imply that the membership is *fixed*, i.e. once an object is linked to a target object in the association it cannot be reconnected to another target object in the same association.

3.1.4.3 Association link and extent

An association *link* is an instance of the association. It is a *tuple* of references to objects. The tuple can be, for example, a *set* of references or a *list* (ordered set) of references. In general the tuple can contain one reference only. The link represents also the *role name*, as discussed earlier. The *extent* is a set of links.

Figure 3-15 is a particular instantiation of the association `OrdShip` in Figure 3-12. There are five links in Figure 3-15. Hence the extent of the association is five.

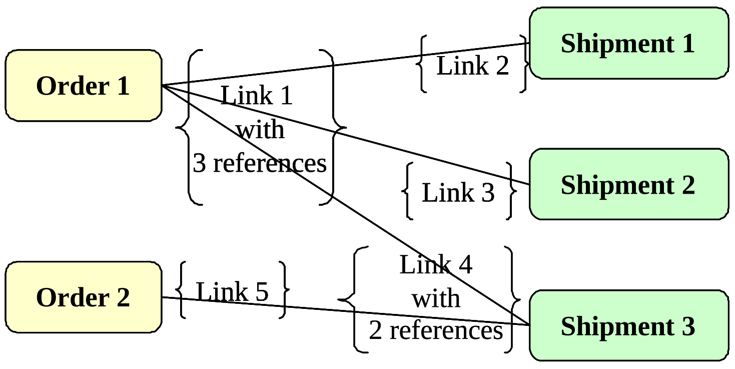


Figure 3-15 Links and extents

The understanding of association links and extents is important to the overall comprehension of the association concept, but the links and extents are not meant to be modeled or otherwise apparent.

3.1.4.4 Association class

Sometimes an association has attributes (and/or operations) of its own. Such an association must be modeled as a class (because attributes can only be defined in a class). Each object of an *association class* has attribute values and links to the objects of associated classes. Because an association class is a class, it can be associated with other classes in the model in the normal way.

Figure 2.15 shows the association class `Assessment`. An object of the class `Assessment` stores the list of marks, the total mark and the grade obtained by a `Student` in a `ClassOffering`.

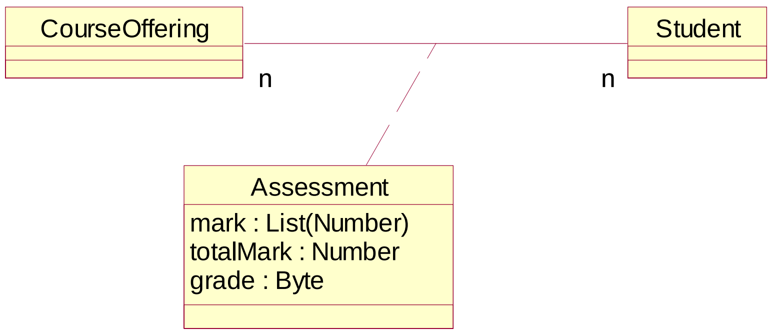


Figure 3-16 Association class

The type of the attribute `mark` is `List(Number)`. This is a so-called *parameterized type*. `Number` is the parameter of the class `List`, where `List` defines an ordered set of values. The attribute `mark` contains the list of all marks that a student obtained in a class offering. That is, if the student 'Fred' takes the course offering 'COMP227' there will eventually be a list (an ordered set) of marks for him on that course offering. That list of marks will be stored in an `Assessment` object that represents the association between 'Fred' and 'COMP227'.

3.1.5 Aggregation and composition

An *aggregation* is a whole-part relationship between a class representing an assembly of components (*superset class*) and the classes representing the components (*subset classes*). A superset class contains a subset class (or classes).

The containment property can be strong (*aggregation by value*) or weak (*aggregation by reference*). In UML, the aggregation by value is called *composition*, and the aggregation by reference is simply called *aggregation*.

From the system modeling perspective, the aggregation is a special kind of association with additional semantics. In particular, the aggregation is transitive and asymmetric. *Transitivity* means that if class A contains class B and class B contains class C, then A contains C. *Asymmetry* means that if A contains B, then B cannot contain A.

Composition has an additional property of *existence dependency*. An object of a subset class cannot exist without being linked to an object of the superset class. This implies that if a superset object is deleted (destroyed), then its subset objects must also be deleted.

The *composition* is signified by the *filled diamond* 'adornment' on the end of the association line connected to the superset class. The *aggregation*, that is not a composition, is marked with the *hollow diamond*. Note, however, that the hollow diamond can also be used if the modeler does not want to make the decision if the aggregation is a composition, or not.

Figure 2.16 shows a composition on the left, and a normal aggregation on the right. Any `Book` object is a composition of `Chapter` objects, and any `Chapter` is a composition of `Section` objects. A `Chapter` object does not have an independent life; it exists only within the `Book` object. The same cannot be said about `BeerBottle` objects. The `BeerBottle` objects can exist outside of their container – a `Crate` object.

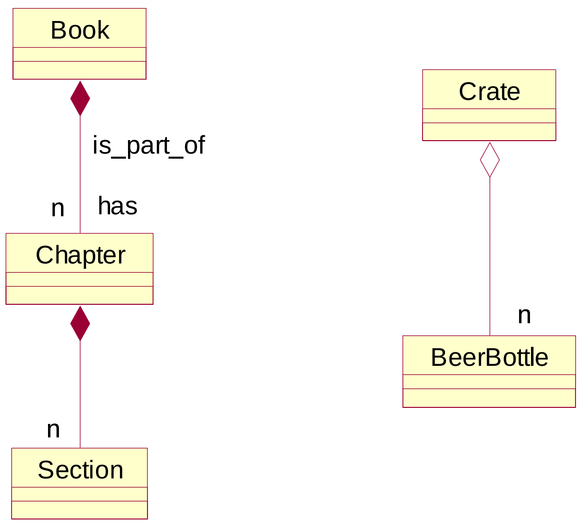


Figure 3-17 Composition and aggregation

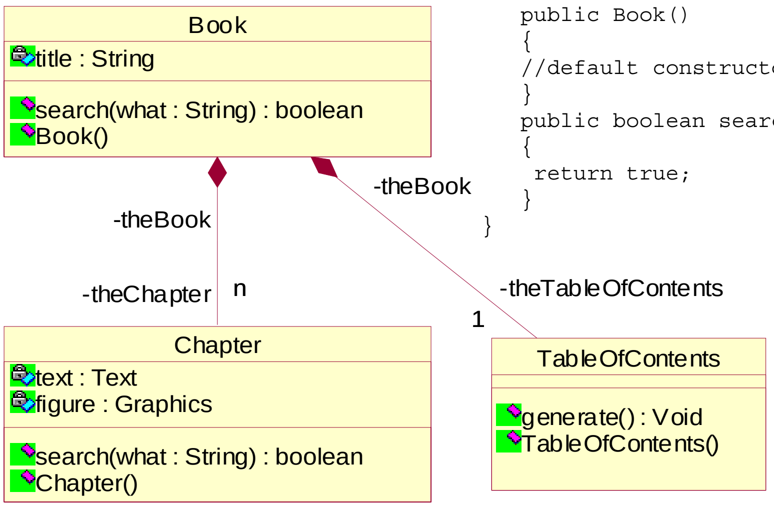
Aggregation and composition (as well as the associated notion of *delegation* introduced in Section 3.1.5.3) are very useful concepts in object technology. It is unfortunate, however, that commercial programming languages give relatively little support to these concepts. In many languages, aggregation and composition is implemented not differently than the implementation of association, i.e. by means of *buried references* (Lee and Tepfenhart, 2002) (Section 3.1.5.1). Java provides an alternative implementation by means of *inner classes* (Section 3.1.5.2).

3.1.5.1 Buried reference

Buried reference implements an aggregation by means of a variable with private visibility that references the subset object. This is not different to the implementation of an association by means of a private reference.. Such implementation does not support any other semantics of aggregation. Hence, for example, to ensure that the deletion of a superset object deletes also its subset objects, the programmer has to implement such deletions in the application code.

Figure 3-18 is an example illustrating buried references. `Book` is modeled as the composition of many `Chapter` objects and one `TableOfContents` object. Consequently, the `Book` class has two buried references – private variables `theChapter` and `theTableOfContents`. Having these references as private hides the identities of the book's chapters and its table of content from other classes. This is, however, of little benefit because the visibility of classes cannot be private with regard to other classes. In the example, `Chapter` and `TableOfContents` have the *package* visibility (the package visibility is implied by the absence of the keyword public in front of these classes' names). The package visibility makes the classes visibility to all classes in the same package (in Java, every class must be assigned to one and only one package).

```
public class Book
{
   private String title;
   private Chapter theChapter[];
   private TableOfContents theTableOfContents;
   public Book()
   {
   //default constructor
   }
   public boolean search(String what)
   {
    return true;
   }
}
```



```
class Chapter
{
   private Text text;
   private Graphics figure;
   private Book theBook;
   public Chapter()
   {
   //default constructor
   }
   public boolean search(String what)
   {
    return true;
   }
}
```

```
class TableOfContents
{
   private Book theBook;
   public TableOfContents()
   {
   //default constructor
   }
   public Void generate()
   {
    return null;
   }
}
```

Figure 3-18 Buried reference

The presence of backward references from `Chapter` and `TableOfContents` to `Book` is also the result of imperfect implementation of aggregation/composition. Subset objects must somehow know their owner.

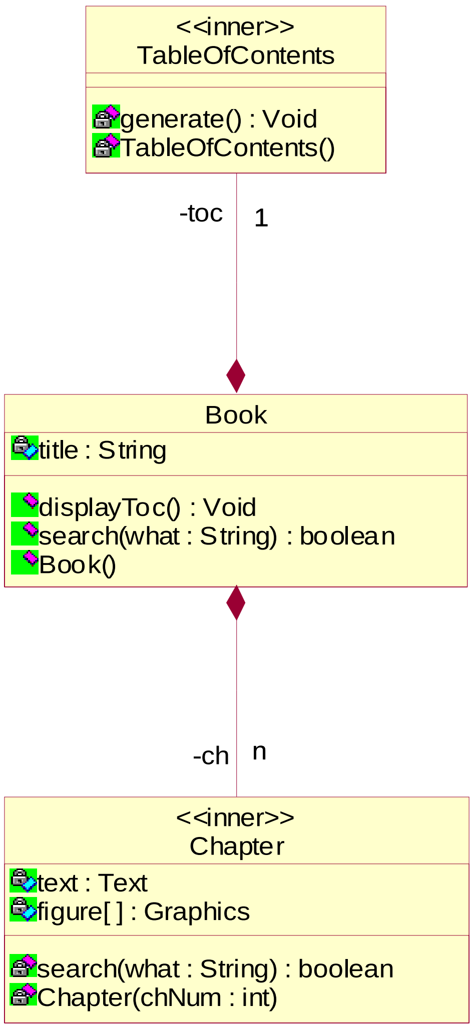
Figure 3-18 illustrates how the superset object can demonstrate to the outside world that it is the owner of subset objects. This is shown in the implementation of the `search()` operation. The service to search for some string in the book is available in the `Book` class. The requestor can send the `search()` message to a `Book` object. The `Book` object can then forward this request to its `Chapter` objects by invoking the `search()` method on these objects. In effect, the requestor relies exclusively on the `Book` object to get the search results back.

3.1.5.2 Inner class

In Java, it is possible to define a class as internal member of another class. The member class can have class-scope, i.e. be declared as static. It is then called a *nested class*. Alternatively, the member class can have instance-scope, and it is then called an *inner class* (Eckel, 2003). It turns out that the inner class could be used as the best Java-supported implementation of aggregation/composition.

An inner class reflects the relationship between the superset instance and its inner subset instances. The superset instance has the natural control over its subset objects because it owns them. In the opposite direction, the subset instances have direct access to all members of the superset object, including the private members.

Figure 3-19 shows how essentially the same model as in Figure 3-18 can be implemented with inner classes. In typical situations, an outer class obtains a reference to an inner class by either instantiating the inner object within its own constructor or by having a method that instantiates the inner object. The example uses the former approach. The `Book()` constructor instantiates `TableOfContents` and `Chapter` objects from its constructor (but it could construct them from other private methods to achieve the same level of encapsulation). `Book` redirects all queries to its content to appropriate `TableOfContents` or `Chapter` objects.



```
public class Book
{
   private String title;
   private Chapter[] ch;
   private TableOfContents toc;
   public Book(...)
   { ...
      toc = new TableOfContents();
      ch = new Chapter[numberChapters];
      for (int i=0; i<numberChapters; i++)
         ch[i] = new Chapter();
   }
   public Void displayToc()
   {
      toc.generate();
      return;
   }
   public boolean search(String what)
   {
      for(int i=0;i<ch.length;i++)
         if( ch[i].search(what))
            return true;
      return false;
   }
   private class Chapter
   {
      private Text text;
      private Graphics[] figure;
      private boolean search(String what)
      {...
      }
   }
   private class TableOfContents
   {
      private Void generate()
      {...
      }
   }
}
```

Figure 3-19 Inner classes
[With contribution of Bruc Lee Liong]

The inner classes have an additional advantage in the implementation of aggregation/composition – they can be made *private* (whereas the normal classes can only have public or package visibility). Private inner classes can only be accessed from the outer class. This completely hides the implementation of inner classes from all but the outer class and it reduces dependencies due to changes of this implementation. In Figure 3-19, `TableOfContents` and `Chapter` are only known to `Book`, but not to other classes in the system.

Other important benefits of using inner classes are related to the notions of *interface* and *inheritance*. Both these notions are discussed later in this chapter. To those already initiated to these concepts, suffice to say that an inner class can implement an interface as well as it can extend (inherit from) a class (Eckel, 2003). The first technique can further reduce dependencies of the program's classes on the inner class, even if the inner class is public. Everything that is made available to the program's classes are references to one or more interfaces that the inner class invisibly implements. The second technique allows in effect *multiple* *implementation inheritance*, even though Java is a single inheritance language. The multiple inheritance comes from the fact that the outer class and its inner classes can independently inherit from other classes.

3.1.5.3 Delegation

Associated with aggregation/composition is the powerful technique of *delegation*. Delegation is a good replacement for inheritance as the code reuse technique (Gamma *et al*., 1995). Although delegation can be used between any classes, it is at its best when applied on classes related by aggregation/composition.

The idea of delegation is as the name suggest. If an object receives a request to perform one of its services and is unable to deliver the service, it can delegate the work to one of its component objects. Delegating the work to another object does not relieve the original recipient of the message from the responsibility for the service. The work is delegated, not the responsibility.

Figure 3-19 gives an example of delegation involving inner class. The delegation can happen on the `search()` method. The `Book` class has `search()` in its public interface, thus promising this service to the outside world. When Book receives a request to perform `search()`, it delegates the work to its `Chapter` objects. `Chapter` performs the work, `Book` gets the glory. The requestor of the service does not even have a choice of directing the message to `Chapter` because `Chapter` has private visibility.

Technically, the scenario in Figure 3-19 is known as *forwarding* of the message, not *delegating*. Delegation is a more complex form of forwarding such that an object delegating the service is passing along a reference to itself (Gamma *et al*., 2000). However, in the case of inner classes there is no need to pass a reference of the delegating object because inner objects have direct access to all members of the outer class (by means of language-implemented hidden references).

Figure 3-19 does not illustrate the *code reuse* aspect of delegation and the related benefit of improved *supportability* of programs relying on delegation. The benefits of reusability and supportability require combining of delegation with interfaces and/or abstract classes (both concepts are discussed later).

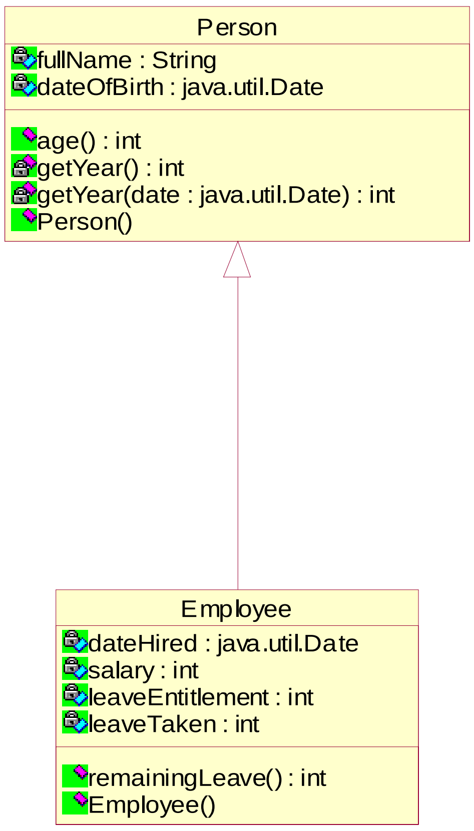
The idea is that changes to objects that do the work (to which the work was delegated) do not affect the program as long as the work is done. As an example, assume that `Chapter` and `Section` objects have the same type, i.e. they inherit from the same superclass (ideally abstract class) or implement the same interface. It will be then possible to replace at run-time the `Chapter` instances by `Section` instances in doing the `search()` operation. This replacement will not be noticed by the client object that requested the `search()` service.

3.1.6 Generalization and inheritance

A *generalization* relationship is a kind-of relationship between a more generic class (*superclass* or *parent*) and a more specialized kind of that class (*subclass* or *child*). The subclass is a kind of superclass. An object of the subclass can be used where the superclass is allowed.

Generalization makes it unnecessary to restate already defined properties. The attributes and operations already defined for a superclass may be *reused* in a subclass. A subclass is said to *inherit* the attributes and methods of its parent class. Generalization facilitates incremental specification, exploitation of common properties between classes and better localization of changes.

A generalization is drawn as a hollow triangle on the relationship end connected to the parent class. In Figure 2.17, `Person` is the superclass and `Employee` is the subclass. The class `Employee` inherits all attributes and operations of the class `Person` (but the private members of `Person` are not accessible to the `Employee` objects). The inherited properties are not visibly shown in the subclass box – the generalization relationship forces the inheritance in the background.



```
import java.util.Date;
import java.util.Calendar;
import java.util.GregorianCalendar;
public class Person
{
   private String fullName;
   private Date dateOfBirth;
   public Person()
   {...}
   public int age(){
      return getYear() - getYear(dateOfBirth);
   }
   private int getYear(){
      return getYear(new Date(System.currentTimeMillis()));
   }
   private int getYear(Date date){
      Calendar cal = GregorianCalendar.getInstance();
      cal.setTime(date);
      return cal.get(Calendar.YEAR);
   }
}

public class Employee extends Person
{
   private Date dateHired;
   private int salary;
   private int leaveEntitlement;
   private int leaveTaken;
   public Employee()
   {...}
   public int remainingLeave(){
      return leaveEntitlement - leaveTaken;
   }
}
```

Figure 3-20 Generalization
[With contribution of Bruc Lee Liong]

Note that inheritance applies to classes, not to objects. It applies to types, not to values. The class `Employee` inherits the definitions of attributes `fullName` and `dateOfBirth`. This is by virtue of the fact that an instance of `Employee` is also an instance of `Person`. Hence, the constructor `Employee()` can set the values for `fullName` and `dateOfBirth` and for the remaining four data members.

But here is a twist. The two attributes in `Person` have private visibility. This implies that the class `Employee` cannot access the `fullName` and `dateOfBirth` values in a `Person` object. This is logical. Somebody named "Joe Guy" is either an employee and a person or he is just a person. If Joe is an employee (and a person), he has its own values for `fullName` and `dateOfBirth` (as well as for `dateHired`, etc.). If Joe is just a person (and not an employee), he again has values for `fullName` and `dateOfBirth` (but not for `dateHired`, etc.).

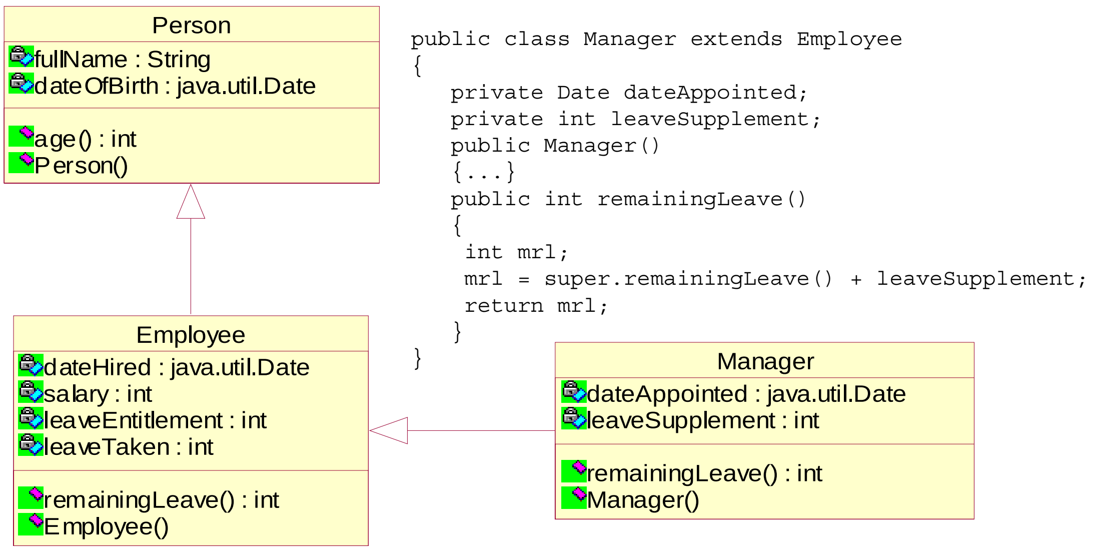
Although an instance of `Employee` has no right to access attribute values in an instance of `Person`, it can call the `Person`'s `age()` method, without referring to the `Person` class name. The inherited `age()` method will access the `dateOfBirth` value of an object on which it is called (i.e., the `Employee` object). Other classes in the program can also call the `age()` method (because of its public visibility), but any such call must either refer to the `Person` class name (`Person::age()`) or call it with a `Person` instance (`person.age()`).

The two private operations in `Person` (two `getYear()` methods) are used internally by the `age()` method to compute the current age of a `Person` object. Java support for manipulating on date values is a bit awkward. Java provides several libraries to do the job (as per the `import` statements in Figure 3-20). The first method `getYear()` returns the current year. The second parameterized method `getYear(Date date)` returns the year for some date in the past.

3.1.6.1 Polymorphism

A method inherited by a subclass is frequently used as inherited in that subclass (i.e. without modification). The operation `age()` works identically for the objects of classes `Person` and `Employee`. However, there are times when an operation needs to be *overridden* (modified) in a subclass to correspond to semantic variations of the subclass.

For example, `Employee.remainingLeave()` is computed by subtracting `leaveTaken` from `leaveEntitlement` (Figure 3-20). However, the employee who is a manager gains a yearly `leaveSupplement`. If we now add the class `Manager` to the generalization hierarchy (as shown in Figure 3-21), the operation `Manager.remainingLeave()` would override the operation `Employee.remainingLeave()`. This is indicated in Figure 3-21 by duplicating the operation name in the `Manager` subclass.



```
public class Manager extends Employee
{
   private Date dateAppointed;
   private int leaveSupplement;
   public Manager()
   {...}
   public int remainingLeave()
   {
    int mrl;
    mrl = super.remainingLeave() + leaveSupplement;
    return mrl;
   }
}
```

Figure 3-21 Polymorphism

The operation `remainingLeave()` has been *overridden*. There are two implementations (two *methods*) of the operation. We can now send the message `remainingLeave()` to an `Employee` object or to a `Manager` object and we will get a different method executed. We may not even know or care which object is targeted: `Employee` or `Manager` – the proper method will execute.

The operation `remainingLeave()` is *polymorphic*. There are two implementations (methods) for this operation. Both methods have the same name and identical *signatures* – the number and types of parameters (in this case, the parameter list is empty).

Polymorphism and inheritance go hand in hand. Polymorphism without inheritance is of limited use. *Inheritance* permits incremental description of a subclass by reusing and then extending the superclass descriptions. The operation `Manager.remainingLeave()` is probably implemented by invoking the functionality of `Employee.remainingLeave()` and then by adding `leaveSupplement` to the value returned from `Employee.remainingLeave()`.

3.1.6.2 Overriding vs overloading

Overriding must not be confused with overloading. *Overriding* is the mechanism to achieve polymorphic operations. Overriden methods have identical names and signatures and are placed in different classes of the same inheritance hierarchy. The decision of which method is invoked is taken dynamically at run-time. The decision is based on the class of the object to which the variable (from which the method is called) points to, and not on the type of the variable (Lee and Tepfenhart, 2002).

If the variable points to a subclass object in the inheritance tree, and the overridden method exists for that subclass, then this overridden method will be invoked. If, however, the method is not declared in the subclass, the programming environment will search up the inheritance tree to find the method in a superclass of the subclass. The search will continue up to the base class, if the method has not been overridden in any subclasses. If this is the case, the method of the base class will be invoked.

Overloading refers to a situation when multiple methods with the same name are declared in the same class. The methods have the same names but they have different signatures and possibly also different return types. Unlike overriding, which is a run time phenomenon, overloading can be resolved at compile time.

Figure 3-20 illustrates overloading in the private methods `getYear()` and `getYear(Date date)`. The former returns the current calendar year. The latter returns the calendar year of some concrete date passed to it as parameter. The program statically decides which method to call. In fact, both methods are called by the public `age()` method declared within the same class (the `Person` class).

3.1.6.3 Multiple inheritance

In some languages, such as C++, a subclass can inherit from more than one superclass. This is called multiple implementation inheritance. *Multiple inheritance* can lead to inheritance conflicts that have to be explicitly resolved by the programmer.

In Figure 3-22, the class `Tutor` inherits from the classes `Teacher` and `PostgraduateStudent`. `Teacher` in turn inherits from `Person` and so does `PostgraduateStudent` (via `Student`). As a result, `Tutor` would inherit twice the attributes and operations of `Person` unless the programmer instructs the programming environment to inherit only once by using either the left or right inheritance path (or unless the programming environment enforces some default behavior, acceptable to the programmer, that eradicates the duplicated inheritance).

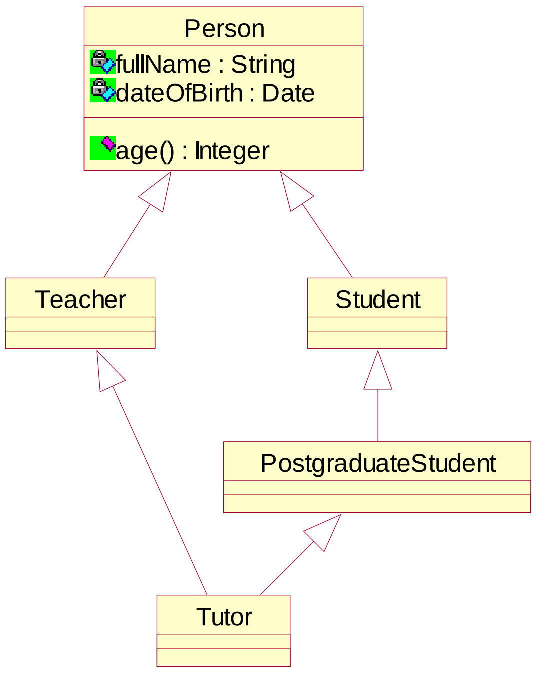


Figure 3-22 Multiple inheritance

Note that Java does not permit multiple implementation inheritance. Java provides alternative mechanisms, in particular interfaces (including multiple interface inheritance) and inner classes, to implement the class structures and behavior corresponding to the model in Figure 3-22.

3.1.6.4 Multiple classification

In most current object-oriented programming environments, an object can belong to only one class. This is a troublesome restriction because in reality objects can belong to multiple classes.

Multiple classification is different from multiple inheritance. In multiple classification an object is simultaneously the instance of two or more classes. In multiple inheritance a class may have many superclasses, but an object is created as an instance of only one of these classes. The "knowledge" of other classes is only available to the object via inheritance.

In the multiple inheritance example in Figure 3-22, each `Person` object (such as `Mary` or `Peter`) belongs to a single class (the most *specific* class that applies to it). If `Mary` is a `PostgraduateStudent`, but not a `Tutor`, then `Mary's` class is `PostgraduateStudent`.

The problem arises if `Person` is specialized in a few orthogonal hierarchies. For example, a `Person` can be an `Employee` or `Student`, `Male` or `Female`, `Child` or `Adult`, etc. Without multiple classification, we would need to define classes for each legal combination between the orthogonal hierarchies to have, for example, a class for a `Person` object who is a child female student (i.e. the class that could be called `ChildFemaleStudent`) (Fowler and Scott, 2000).

3.1.6.5 Dynamic classification

In most current object-oriented programming environments, an object cannot change its class after it has been instantiated (created). This is another troublesome restriction because in reality objects do change classes dynamically.

Dynamic classification is a direct consequence of multiple classification. An object does not only belong to multiple classes but it can gain or lose classes over its lifetime.

Under the dynamic classification scheme, a `Person` object can be just an employee one day and a manager (and employee) another day. Without dynamic classification, the business changes such as promotion of employees are hard (or even impossible) to implement. The implementation problem arises because definitions of object identifiers (OIDs) include the identification of the class to which an object belongs. Dynamic classification would necessitate changes of OIDs, which would defeat the very idea of OIDs (section 3.1.1.3).

The lack of support for multiple and dynamic classification in programming languages translates to the similar lack of support in UML modeling. Consequently, no graphical models are shown here to enhance the explanation.

3.1.7 Abstract class

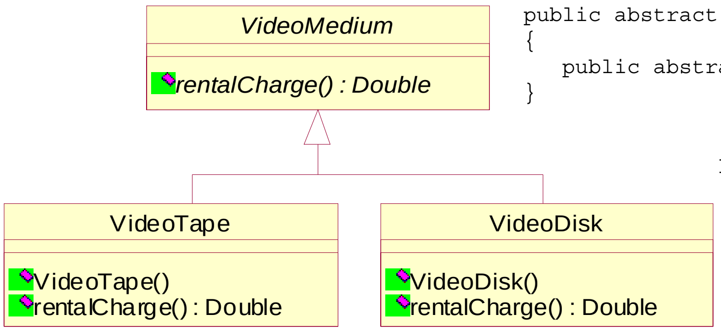
Abstract class is an important modeling concept that follows on from the notion of inheritance. An abstract class is a parent class that will not have direct instance objects. Only subclasses of the abstract parent class can be instantiated.

In a typical scenario, a class is abstract because at least one of its operations is abstract. An *abstract operation* has its name and signature defined in the abstract parent class, but the implementation of the operation (the method) is deferred to *concrete* child classes.

The reason why an abstract class cannot instantiate objects is because it has at least one abstract operation. If it was allowed for an abstract class to create an object then a message to that object's abstract operation would cause a run-time error (because there would not be an implementation for the abstract operation in the class of that object).

A class can only be abstract if it is a superclass that is completely partitioned into subclasses. The partitioning is complete if the subclasses contain all possible objects that can be instantiated in the inheritance hierarchy. There are no 'stray' objects (Page-Jones, 2000). The class `Person` in Figure 3-22 is not abstract because we may want to instantiate objects of `Person` that are not teachers or students. It is also possible that we may want to add more subclasses of `Person` in the future (such as `AdminEmployee`).

Figure 3-23 shows the abstract class `VideoMedium` (in UML, the name of the abstract class is shown in italics). The class contains the abstract operation `rentalCharge()`. Understandably, rental charges are calculated differently for video tapes and for video disks. There will be two different implementations of `rentalCharge()` – in classes `VideoTape` and `VideoDisk`.



```
public abstract class VideoMedium
{
   public abstract Double rentalCharge();
}
```

```
public class VideoDisk extends VideoMedium
{
   public VideoDisk()
   {...}
   public Double rentalCharge()
   {
    return null;
   }
}
```

Figure 3-23 Abstract class

Abstract classes do not have instances, but they are very useful in modeling. They create a high-level modeling 'vocabulary' without which the modeling language would be deficient.

3.1.8 Interface

The idea of abstract classes is brought to complete fruition in Java interfaces. An *interface* is a definition of a semantic type with attributes (constants only) and operations but without actual declarations of operations (i.e. without implementation). The actual declarations are provided by one or more classes that undertake to implement the interface.

A program can use an interface variable in lieu of a class variable, thus separating the client class from the actual supplier of the implemented method. The client object can determine only at run time the value of the interface variable and invoke appropriate method on the supplier object.

3.1.8.1 Interface vs abstract class

Abstract classes are a powerful mechanism but they are not helpful in resolving multiple inheritance problems and they are not free from other undesired side effects of implementation inheritance, to be discussed exhaustively later in the book. One of such side effects is the *fragile base class* problem – any change in the implementation of the base class has a largely unpredictable effect on the subclasses that inherit from that base class. Since an abstract class can have some methods fully or partially implemented, it can become a fragile base class.

Figure 3-24 demonstrates how an abstract class is of no help in resolving a modeling situation that seems to ask for multiple inheritance. Assuming that the Video Store (Section 1.5.2) rents not just the movies but also the video playing equipment, the modeler may be tempted to inherit from `VideoMedium`. However, under Java single inheritance mechanism, such inheritance will not be allowed. This is because `VideoPlayer` inherits already from `VideoEquipment` and `VideoMedium` is a class (albeit abstract).

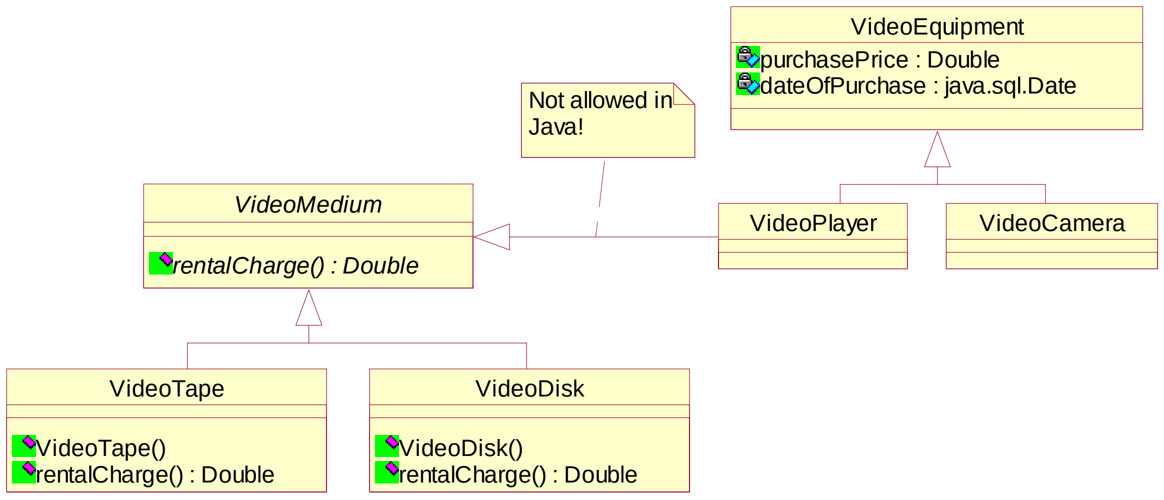


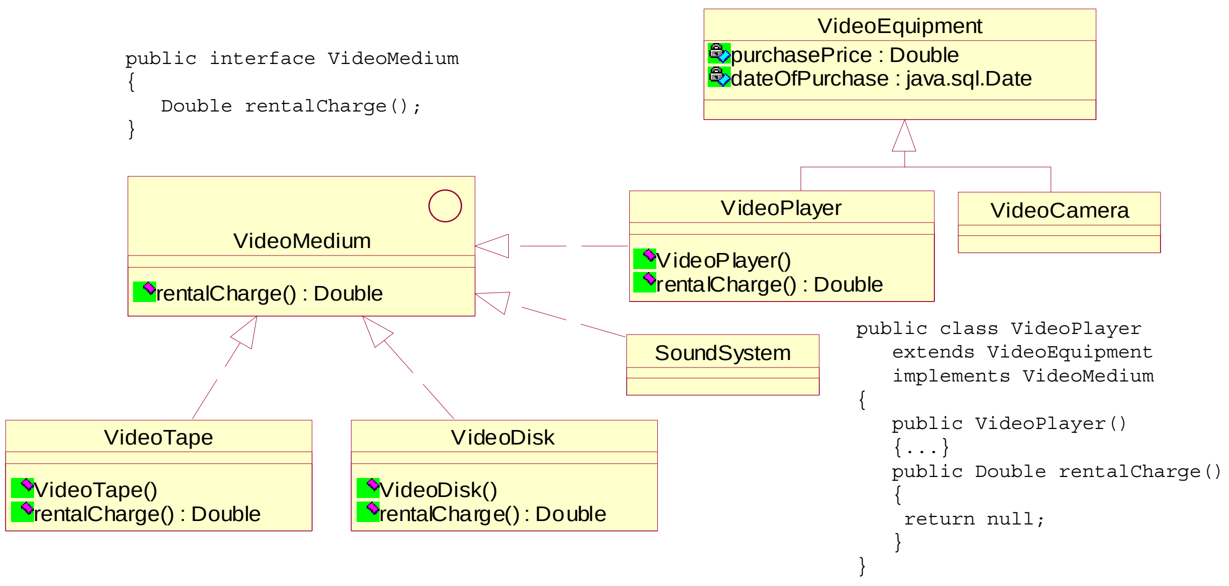
Figure 3-24 Multiple implementation inheritance not allowed in Java (Video Store)

The notion of *interface* comes to rescue and it provides other advantages in the process (Lee and Tepfenhart, 2002; Maciaszek and Liong, 2004). Like abstract class, interface defines a set of attributes and operations, but no objects of it can be instantiated. Unlike abstract class, interface does not implement (even partially) any of its methods.

The total lack of implementation in interface seems to be similar to the notion of a *pure abstract class*, available in C++. But even here there is a difference. In case of pure abstract class, the only classes that want to implement the pure methods must be subclasses of that pure abstract class. In case of interface, any class in the system can implement the interface. Moreover, the class can implement any number of interfaces.

3.1.8.2 Implementing interface

Figure 3-25 shows a Video Store model in which the abstract class of Figure 3-24 is replaced by the `VideoMedium` interface. Graphically, the interface is recognized by the circle in the name compartment (this is one of few possible UML visualizations). All methods of an interface are implicitly public and abstract, so there is no need to use these keywords in the method's prototype.



```
public interface VideoMedium
{
   Double rentalCharge();
}
```

```
public class VideoPlayer
   extends VideoEquipment
   implements VideoMedium
{
   public VideoPlayer()
   {...}
   public Double rentalCharge()
   {
    return null;
   }
}
```

Figure 3-25 Implementing Java interface (Video Store)

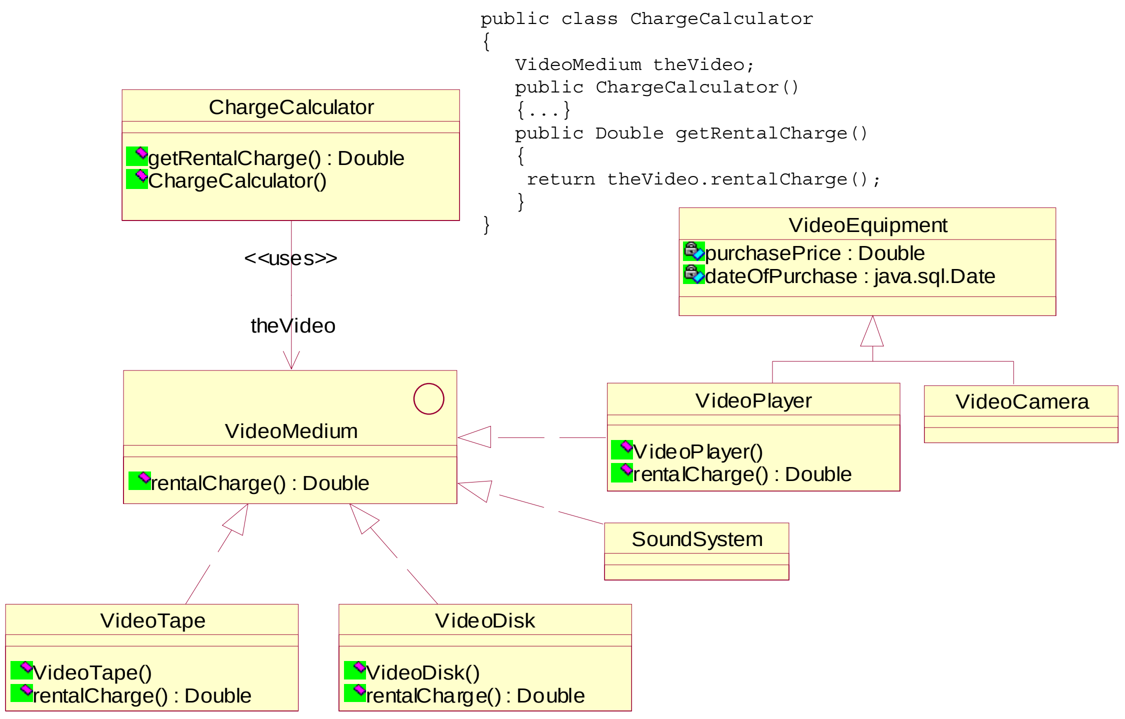
Although not shown in Figure 3-25, a Java interface can also include constant declarations (i.e. attributes that are public, static and final). This is a restriction. A more powerful mechanism should allow declaration in the interface of any attribute typed as another interface or class. This would in effect allow the declaration of associations between interfaces and between an interface and classes. Such mechanism is not yet supported by Java, but it is alrady envisaged for introduction in the forthcoming UML standards.

Also not shown in Figure 3-25, an interface can inherit from another interface (i.e. it can extend another interface). Figure 3-25 illustrates how a class (`VideoPlayer`) can extend a class (`VideoEquiment`) and at the same time implement one or more interfaces (`VideoMedium`).

3.1.8.3 Using interface

The power of interfaces does not come only from providing a handy resolution to multiple implementation inheritance. Even more importantly, an interface defines a *reference type* that allows separation of client objects from the implementation changes in the supplier objects.

An interface name can be referred to anywhere in the program where the client needs to refer to the class that implements this interface. As a result, the implementation of the interface can change and the client class can work as before and may not even notice the change. This feature of interfaces facilitates greatly the *supportability* of the system.



```
public class ChargeCalculator
{
   VideoMedium theVideo;
   public ChargeCalculator()
   {...}
   public Double getRentalCharge()
   {
    return theVideo.rentalCharge();
   }
}
```

Figure 3-26 Using interface to eliminate dependency to the supplier (Video Store)

As an aside, Figure 3-26 illustrates also the use of single-directional association. The `«uses»` association is single-directional from `ChargeCalculator` to `VideoMedium` (this is indicated graphically by the arrow). A `ChangeCalculator` objects "knows" of `VideoMedium` object (in the `theVideo` variable), but the model does not maintain the reverse link from `VideoMedium` to `ChargeCalculator`.

3.2 Fundamentals of object modeling

This section presents fundamentals of visual object modeling by demonstrating various UML diagrams and by explaining how they fit together. Each UML diagram emphasizes a particular *view* on the system. To understand the system in its entirety, multiple UML diagrams, representing different views, have to be developed and integrated.

At the most generic level, the UML standard distinguishes three kinds of models – each with its own set of diagrams and related constructs:

1. The *state model* represents the *static view* of the system – it models data requirements and identifies operations that act on these data. The state model represents data structures and their relationships. Operations are added to the state model once they are defined in the behavioral model. The main visualization technique for state modeling is the class diagram.

2. The *behavior model* represents the *operational view* of the system – it models function requirements. The behavior model represents business transactions, operations and algorithms on data. There

are several visualization techniques for behavior modeling – use case diagram, sequence diagram, collaboration diagram, and activity diagram.

3. The *state change model* represents the *dynamic view* of the system – it models object evolution over time, as permitted by business rules. The state change model represents possible changes to object states (where the *state* is defined by the current values of an object's attributes and relationships with other objects). The main visualization technique for state change modeling is the statechart diagram.

3.2.1 Use case modeling

The *use case model* is the main UML representative and the focal point of behavior modeling. In fact, the importance of use cases goes even further. Use cases drive the entire software development lifecycle – from requirements analysis to testing and maintenance. They are the focal point and the reference to most development activities (section 2.5, Figure 2-2).

System behavior is what a system does when responding to external events. In UML, the outwardly visible and testable system behavior is captured in use cases. A *use case* performs a business function that is *outwardly visible* to an actor and that can be separately *testable* later in the development process. An *actor* represents whoever or whatever (person, machine, etc.) that interacts with a use case. The actor interacts with the use case in expectation of receiving a useful result.

A use case diagram is a visual representation of actors and use cases together with any additional definitions and specifications. A use case diagram is not just a diagram but also a fully documented model of the system's intended behavior. The same understanding applies to other UML diagrams. Unless stated otherwise, the notion of *UML diagram* is synonymous with *UML model*.

It is worthwhile to emphasize again the point made in Section 2.5.2 that a use case model can be viewed as a generic technique of describing all business processes, not just information system processes. When used in such capacity, a use case model would include all social business processes and would then identify which of these processes should be automated (and become information system processes). However, despite attractiveness of this proposition, this is not typical practice in system modeling. Normally only automated processes are captured.

3.2.1.1 Actors

Actors and use cases are determined from the analysis of function requirements. Function requirements are materialized in use cases. Use cases satisfy function requirements by providing a result of value to an actor. It is immaterial whether the business analyst chooses to first identify actors and then use cases or the other way around.

An actor is a *role* that somebody or something external to system plays with regard to a use case. An actor is not a particular instance of somebody or something. So, somebody named "Joe" is not an actor. "Joe" can play a role of customer and be represented in the use case model by the actor `Customer`. In general, `Customer` does not even have to be a person. It could be an organization.

A typical graphical image for an actor is a 'stick person' (Figure 3-27). In general, an actor can be shown as a *class* rectangular symbol. Like a normal class, an actor can have attributes and operations (events that it sends or receives). Figure 3-27 demonstrates three graphical representations for actors.

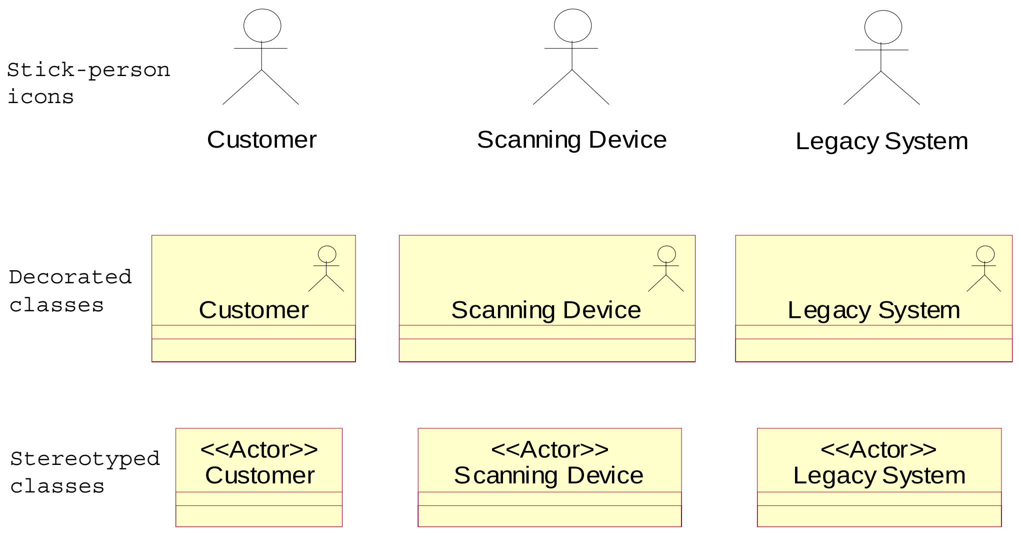


Figure 3-27 Three graphical representations for actors

3.2.1.2 Use cases

A *use case* represents a complete unit of functionality of value to an actor. A use case is a sequence of events of interest to an actor. The actor may either stimulate the use case or the use case can communicate something to the actor.

An actor who does not communicate with a use case is meaningless, but the converse is not necessarily true (i.e. a use case that does not communicate with an actor may be allowed in some situations). There may be some use cases that generalize or specialize main use cases and do not directly interact with actors. They are used internally in the use case model and assist the main use cases in providing results to actors. This said, purists may disallow modeling in which a use case does not communicate with an actor.

Use cases can be derived from the identification of tasks of the actor. The question to ask is: 'What are the actor's responsibilities towards the system and expectations from the system?' Use cases can also be determined from direct analysis of function requirements. In many instances, a *function requirement* maps directly to a *use case*.

Table 3-1 shows how selected function requirements for the Video Store system can be used to identify actors and use cases. There are two actors involved in all four requirements but clearly the level of actors' involvement varies from requirement to requirement. The `Scanning Device` is not selected as an actor – it is considered internal to the system.

Table 3-1 Assignment of requirements to actors and use cases (Video Store)

Req#	Requirement	Actor	Use case
1	Before a video can be rented out, the system confirms	`Customer, Employee`	`Scan Membership Card`

	customer's identity and standing by swiping over scanner his/her Video Store membership card.		
2	A video tape or disk can be swiped over scanner to obtain its description and price (fee) as part of customer's enquiry or rental request.	`Customer, Employee`	`Scan Video Medium`

3	Customer pays the nominal fee before the video can be rented out. The payment may be with cash or debit/credit card.	`Customer, Employee`	`Accept Payment`
			`Charge Payment to Card`

4	The system verifies all conditions for renting out the video, acknowledges that the transaction can go ahead, and can print the receipt for the customer.	`Employee, Customer`	`Print Receipt`

The use cases can be named using the system's or the actors' viewpoint. Table 3-1 leans towards the system's viewpoint. Hence, for example, `Accept Payment`, not `Make Payment`.

Naming use cases from the internal system's perspective is not always a recommended practice. It is easy to argue that they should be named from the external actor's perspective. However, the latter approach makes it difficult to smoothly connect use cases and models/artifacts developed later in the lifecycle, because these models/artifacts take a strong system's orientation.

Figure 3-28 illustrates use cases identified in Table 3-1. A use case is drawn in UML as an ellipse with the name inside the ellipse or below it.

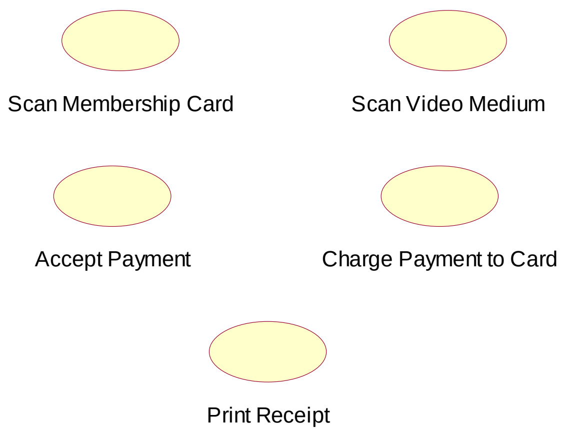


Figure 3-28 Use cases (Video Store)

3.2.1.3 Use case diagram

The use case diagram assigns use cases to actors. It also allows the user to establish relationships between use cases, if any. These relationships are discussed in Chapter 4.

The use case diagram is the principal visualization technique for a behavioral model of the system. The diagram elements (use cases and actors) need to be further described to provide a complete *use case model* (see next Section).

Figure 3-29 incorporates the use cases of Figure 3-28 into a diagram with actors. The model shows that only the `Employee` actor engages directly with use cases. `Customer` is the secondary actor that depends on `Employee` to achieve most of the goals. Hence, the dependency relationship between `Customer` and `Employee`. The direct communication between `Customer` and `Charge Payment to Card` signifies the need for the `Customer` to enter PIN number and confirm the payment on the card scanning device.

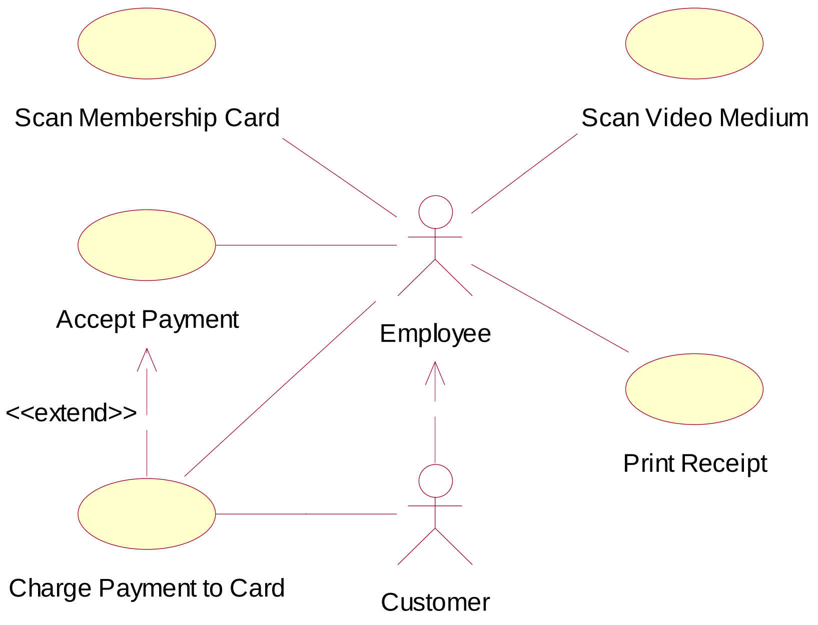


Figure 3-29 Use case diagram (Video Store)

In general, use case diagrams allow a few kinds of relationships between modeling elements. These relationships are discussed in Chapter 4. The `«extend»` relationship in Figure 3-29 signifies that the functionality of `Accept Payment` may be sometimes extended (supported) by the use case `Charge Payment to Card`.

The relative placement of actors and use cases on the diagram is arbitrary and left to the "visual senses" of the modeler.

3.2.1.4 Documenting use cases

Each use case has to be described in a *flow of events* document. This textual document defines what the system has to do when the actor activates a use case. The structure of a *use case document* can vary, but a typical description would contain (cp. Quatrani, 2000):

- Brief description
- Actors involved
- Preconditions necessary for the use case to start
- Detailed description of flow of events that includes:
- Main flow of events, that can be broken down to show:
- Subflows of events (subflows can be further divided into smaller subflows to improve document readability)
- Alternative flows to define exceptional situations
- Postconditions that define the state of the system after the use case ends

The use case document evolves with the development progress. In the early stage of requirements determination, only a brief description is written. Other parts of the document are written gradually and iteratively. A complete document emerges at the end of the requirements specification phase. At that stage, the prototypes of GUI screens can be added to the document. Later on, the use case document will be used to produce the user documentation for the implemented system.

Table 3-2 is an example of the narrative specification for the use case `Accept Payment` from Figure 3-29. The specification includes the specification for `Charge Payment to Card`, which extends `Accept Payment`. The tabular form is not the usual way of documenting use cases. Use case documents can consist of many pages (ten or so on average) and a normal document structure, complete with a table of contents, would be the norm.

Table 3-2 Narrative specification for use case (Video Store)

Use case	**Accept Payment**
Brief description	This use case allows an `Employee` to accept the payment from `Customer` for a video rental. The payment may be made in cash or by debit/credit card.
Actors	`Employee` (primary), `Customer` (secondary).
Preconditions	`Customer` expresses readiness to rent the video and he/she possesses valid membership card and the video is available for rental.
Main flow	The use case begins when the `Customer` decides to pay for the video rental and offers cash or debit/credit card

	payment. The Employee requests the system to display the rental charge together with basic customer and video details. If the `Customer` offers cash payment, the `Employee` handles the cash, confirms to the system that the payment has been received, and asks the system to record the payment as made. If the `Customer` offers debit/credit card payment, the `Employee` swipes the card, requests the `Customer` to type the card's PIN number, select debit or credit account, and transmit the payment. Once the payment is electronically confirmed by the card provider, the system records the payment as made. The use case ends.
Alternative flows	The `Customer` does not have sufficient cash and does not offer the card payment. The `Employee` asks the system to verify the `Customer's` rating (derived from the customer's history of payments). The `Employee` decides if to rent out the video with no or with partial payment. Depending on the decision, the `Employee` cancels the transaction (and the use case terminates) or proceeds with partial payment (and the use case continues). The `Customer` card does not swipe properly through the scanner. After three unsuccessful attempts, the `Employee` enters the card number manually. The use case continues.
Postconditions	If the use case was successful, the payment is recorded in the system's database. Otherwise, the system's state is unchanged.

3.2.2 Activity modeling

The *activity model* can graphically represent the flow of events of a use case. Activity models fill a gap between a high-level representation of system behavior in *use case models* and much lower-level representation of behavior in *interaction models* (sequence and collaboration diagrams).

The activity diagram shows the steps of a computation. Each step is a *state* of doing something. For that reason, the execution steps for an *activity* are called *action states*. The diagram depicts which steps are executed in sequence and which can be executed concurrently. The flow of control from one action state to the next is called a *transition* (or a *flow*).

If a use case document has been completed then activities and action states can be discovered from the description of the *main and alternative flows*. However, activity models can have other uses in system development apart from providing detailed specifications for use cases (Fowler, 2004). They can be used to understand a business process at a high level of abstraction before any use cases are produced. Alternatively, they can be used at a much lower level of abstraction to design complex sequential algorithms or to design concurrency in multi-threaded applications.

3.2.2.1 Actions

If the activity modeling is used to visualize the sequencing of actions in a use case then action states can be established from the use case document. Table 3-3 lists the statements in the main and alternative flows of the use case document and identifies action states.

Table 3-3 Finding actions in main and alternative flows (Video Store)

No.	Use case statement	Action state
1	The Employee requests the system to display the rental charge together with basic customer and video details.	`Display transaction details`
2	If the `Customer` offers cash payment, the `Employee` handles the cash, confirms to the system that the payment has been received, and asks the system to record the payment as made.	`Key in cash amount;` `Confirm transaction`
3	If the `Customer` offers debit/credit card payment, the `Employee` swipes the card, and requests then the `Customer` to type the card's PIN number, select debit or credit account, and transmit the payment. Once the payment is electronically confirmed by the card provider, the system records the payment as made.	`Swipe the card;` `Accept card number;` `Select card account;` `Confirm transaction`
4	The `Customer` does not have sufficient cash and does not offer the card payment. The `Employee` asks the system to verify the `Customer's` rating (which accounts for the customer's history of payments). The `Employee` decides if to rent out the video with no or with partial payment. Depending on the decision, the `Employee` cancels the transaction (and the use case terminates) or proceeds with partial payment (and the use case continues).	`Verify customer rating;` `Refuse transaction;` `Allow rent with no payment;` `Allow rent with partial payment`
5	The `Customer` card does not swipe properly through the scanner. After three unsuccessful attempts, the `Employee` enters the card number manually.	`Enter card number manually`

An action state is represented in UML by a rounded rectangle. The activities identified in Table 3-3 are drawn in Figure 3-30.

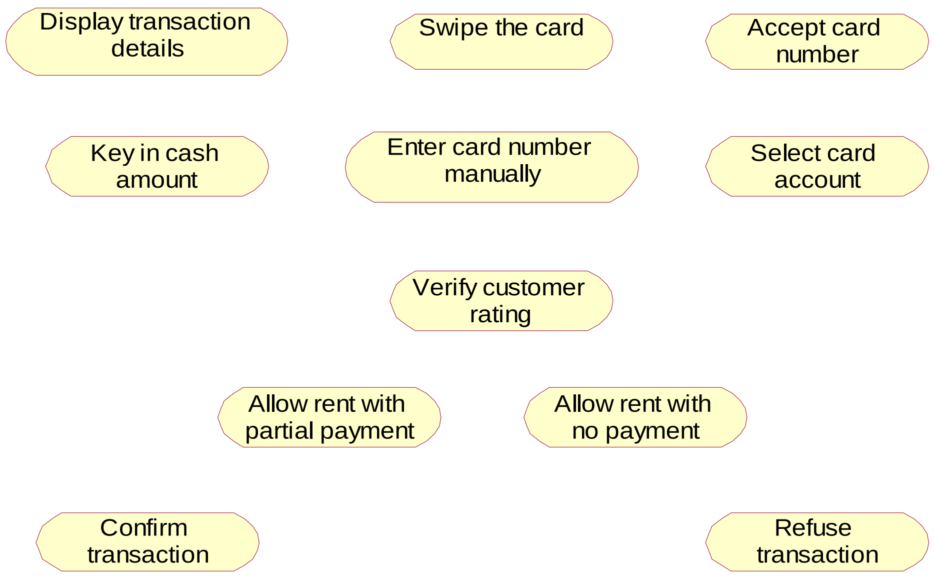


Figure 3-30 Actions for use case (Video Store)

3.2.2.2 Activity diagram

An *activity diagram* shows transitions between actions. Unless an activity diagram represents a continuous loop, the diagram will have an initial action state and one or more final action states. A solid filled circle represents the *initial state*. The *final state* is shown using a 'bull's eye' symbol.

Transitions can *branch* and *merge*. This creates *alternative* computation *threads*. A diamond box shows the branch condition. The exit from a branch condition is controlled by an event (such as `Yes`, `No`) or by a quard condition (such as `[green light]`, `[good rating]`).

Transitions can also *fork* and *rejoin*. This creates *concurrent* (parallel) computation *threads*. The fork/join of transitions is represented by a bar line. An activity diagram without concurrent processes resembles a conventional *flowchart*. (There is no example of concurrent behavior in this Section)

To draw the diagram for the Video Store example, the actions identified in Figure 3-30 have to be connected by transition lines, as demonstrated in Figure 3-31. `Display transaction details` is the initial action state. The *recursive transition* on this state recognizes the fact that the display is continuously refreshed until the next transition fires.

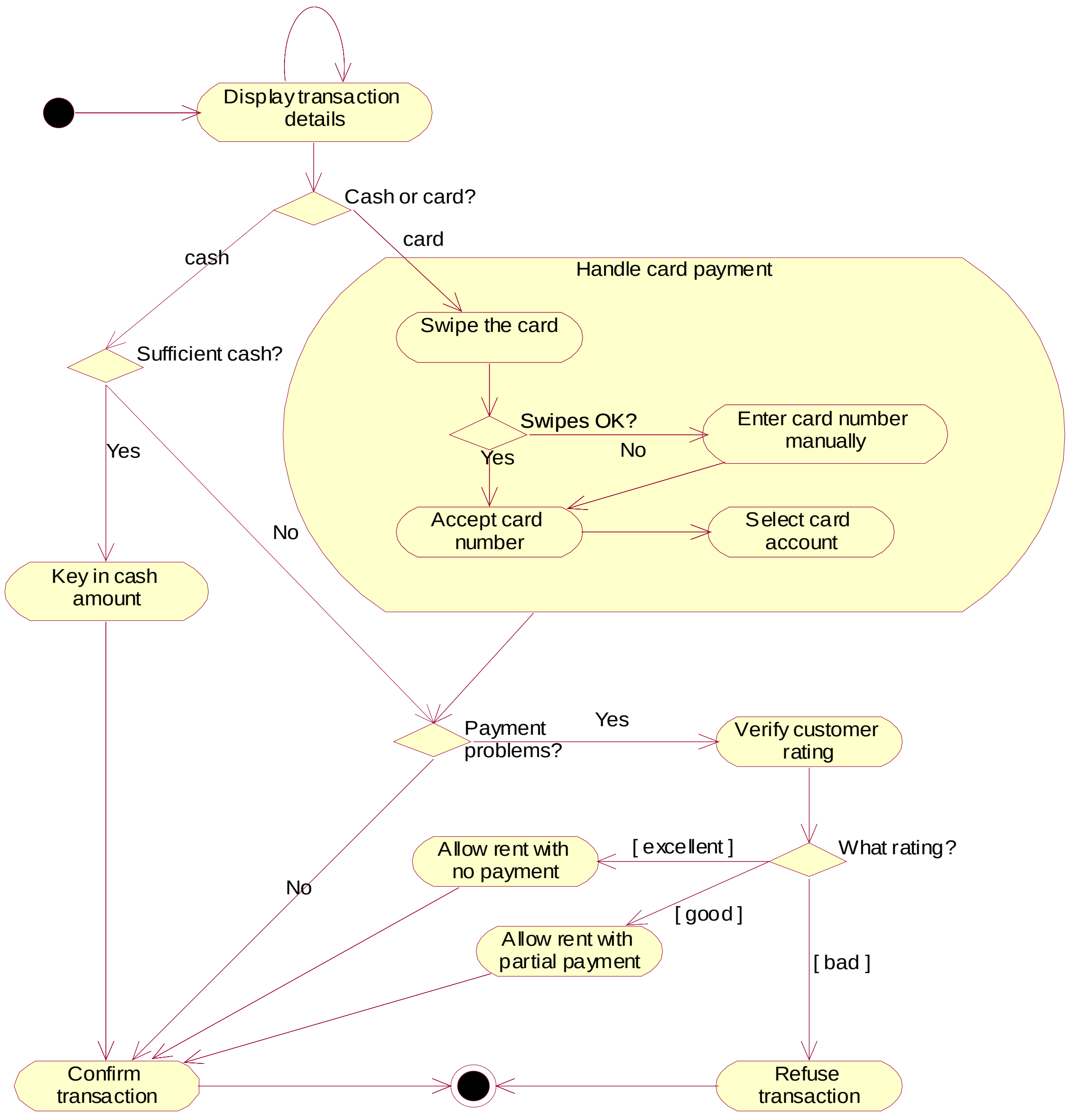


Figure 3-31 Activity diagram for use case (Video Store)

When in the state `Display Transaction Details`, the customer can offer cash or card payment, which leads to execution of one of two possible computation threads. Several actions to manage a card payment are combined in Figure 3-31 in an encompassing action called `Handle card payment`. This kind of "nesting" of actions is convenient when a transition is possible from any of the nested actions. If this is the case, the transition can be drawn from the encompassing action, as shown for the transition to the branch condition called `Payment problems?`.

Testing of the condition `Payment problems?` may come from the problems with the card payment as well as due to insufficient cash money. If there are no payment issues then the rent transaction is confirmed and the processing terminates on the final state. Otherwise, customer rating is verified.

Depending on the rating, the rent transaction is declined (if `[bad]` rating), allowed with partial payment (if `[good]` rating), or allowed with no payment (if `[excellent]` rating).

3.2.3 Class modeling

Class modeling integrates and embodies all other modeling activities. Known also as *state modeling*, class models define the static structures that capture the internal state of the system. Class models identify classes and their attributes, including relationships. These are static structures. However, class models define also operations necessary to fulfill the dynamic behavioral requirements of the system specified in use cases. When implemented in a programming language, classes represent both the static structure and the dynamic behavior of the application.

Accordingly, class models are constructed by referring to all fundamental object technology concepts discussed in Section 3.1. Understanding these concepts is the necessary but not sufficient condition for working with class models. It is the necessary condition for reading (understanding) class models. But it is not a sufficient condition for writing (developing) class models. Developing class models demands additional skills which have to do with proper use of abstraction and with ability to (iteratively) integrate variety of inputs into a single coherent solution.

The outcome of class modeling is a class diagram and related textual documentation. In this chapter, class modeling is discussed after use case modeling, but in practice these two activities are typically conducted in parallel. The two models feed off each other by providing auxiliary but complementary information. Use cases facilitate class discovery and vice versa – class models can lead to the discovery of overlooked use cases.

3.2.3.1 Classes

In the discussion so far, we have used classes to define *business objects*. Our class examples have all been long-lived (persistent) business entities, such as `Order`, `Shipment`, `Customer`, `Student`, etc. These are the classes that define the *database model* for an application domain. For that reason, such classes are frequently called *entity classes* (model classes). They represent persistent database objects.

The *entity classes* define the essence of any information system. Requirements analysis is interested predominantly in entity classes. However, for the system to function, other classes are needed as well. The system needs classes that define GUI objects (such as screen forms) – called the *presentation (boundary, view) classes*. The system also needs classes that control the program's logic and process user events – the *control classes*. Other categories of classes are needed as well, such as the classes responsible for communication with external data sources – called sometimes the *foundation classes*. Finally, the responsibility for managing the entity objects in the memory cache in order to satisfy business transactions is given to yet another category of classes – the *mediator classes*.

Depending on a particular modeling approach, the classes other than entity classes may or may not be addressed in any detail in requirements analysis The same thinking may apply to the definition of operations in early class models. Initial modeling of non-entity classes and the definition of operations may be postponed to the interaction modeling stage (Section 3.2.4). And more detailed modeling may be postponed to the system design phase.

Following the approach taken in finding actors and use cases (see Table 3-1), we can construct a table that assists in finding classes from the analysis of function requirements. Table 3-4 assigns the function requirements to the entity classes.

Table 3-4 Assignment of requirements to entity classes (Video Store)

Req#	Requirement	Entity class
1	Before a video can be rented out, the system confirms customer's identity and standing by swiping over scanner his/her Video Store membership card.	`Video, Customer, MembershipCard`
2	A video tape or disk can be swiped over scanner to obtain its description and price (fee) as part of customer's enquiry or rental request.	`VideoTape, VideoDisk, Customer, Rental`
3	Customer must pay the nominal fee before the video can be rented out. The payment may be with cash or debit/credit card.	`Customer, Video, Rental, Payment`
4	The system verifies all conditions for renting out the video, acknowledges that the transaction can go ahead, and prints the receipt for the customer.	`Rental, Receipt`

Finding classes is an iterative task and the initial list of candidate classes is likely to change. Answering a few questions may help to determine whether a concept in the requirements is a candidate class. The questions are:

- Is the concept a container for data?
- Does it have separate attributes that will take on different values?
- Would it have many instance objects?
- Is it in the scope of the application domain?

The list of classes in Table 2.4 still raises many questions. For example:

- What's the difference between `Video` and `VideoTape/VideoDisk`? Is `Video` just a generic term for `VideoTape/VideoDisk`? If so, don't we need a class to describe the video `Movie` or other content of the video medium? Perhaps a class called `Movie` is required?
- Is the meaning of `Rental` in Req# 2, 3, and 4 the same? Is it all about rental transaction?
- Perhaps `MembershipCard` is part of `Customer`?
- Is there a need to distinguish separate classes for `CashPayment` and `CardPayment`?

- Although a Video Store employee, as an actor, is not explicitly mentioned in the requirements in Table 3-4, it is clear that the system must have knowledge about which employees were involved in rental transactions. Clearly, there is a need for the class `Employee`.

Answering these and similar questions is not easy and requires an in-depth knowledge of application requirements. Figure 3-32 includes all classes identified in Table 3-4 as well as raised in the above discussion. Note that the classes `Customer` and `Employee` have already appeared as *actors* in the use case diagram – hence the annotation 'from use case view'. This duality of actors as external entities interacting with the system and as internal entities about which the system must have some knowledge is quite common in system modeling.

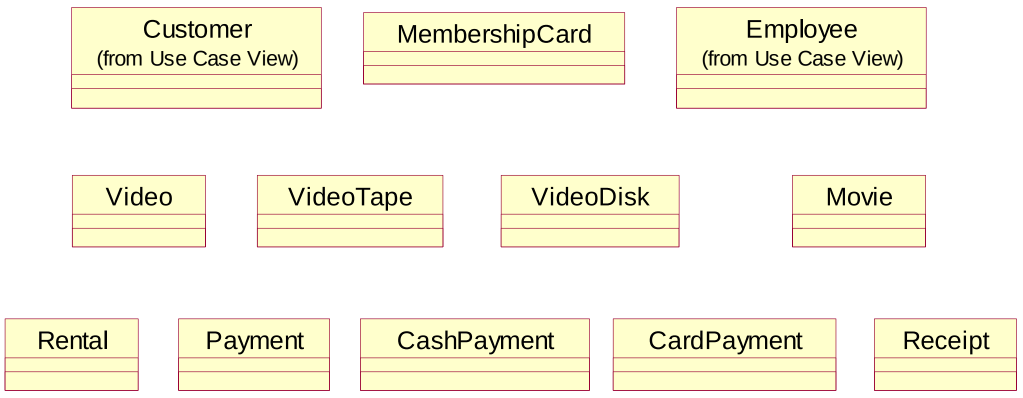


Figure 3-32 Classes (Video Store)

3.2.3.2 Attributes

The structure of a class is defined by its *attributes* (Section 3.1.2.1). The analyst must have some appreciation of the attribute structure when initially declaring a class. In practice, main attributes are usually allocated to a class immediately after the class has been added to the model.

Attributes are discovered from user requirements and from the domain knowledge. Initially, the modeler concentrates on defining *identifying attributes* for each class – one or more attributes in a class that have unique values across all instances of the class. Such attributes are frequently referred to as class *keys*. Ideally, a key should consist of one attribute. In some cases, a set of attributes constitutes a key.

Once identifying attributes are known, the modeler should define main *descriptive attributes* for each class. These are attributes that describe the main informational content of the class. There is no need, as yet, to start defining *non-primitive types* for attributes (Section 3.1.2.1). Most attributes that look like they require non-primitive types can be typed as strings of characters. The strings can be converted to non-primitive types in later modeling stages.

Figure 3-33 shows two Video Store classes with primitive attributes. Both classes have identical keys (`membershipId`). This confirms the question, raised in Section 3.2.3.1, that `MembershipCard` is some kind of part of `Customer`. This issues will definitely come back in later modeling stages.

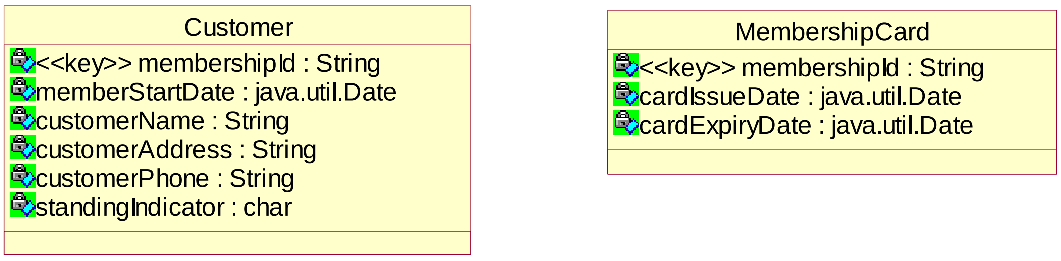

Customer
<<key>> membershipId : String
memberStartDate : java.util.Date
customerName : String
customerAddress : String
customerPhone : String
standingIndicator : char
MembershipCard
<<key>> membershipId : String
cardIssueDate : java.util.Date
cardExpiryDate : java.util.Date

Figure 3-33 Primitive attributes in classes (Video Store)

Some attributes in Figure 3-33 are typed as `java.util.Date`. This is a `Date` data type provided by a Java library and although it is a non-primitive data type, it is not *user-defined* (and, therefore, not contradicting the assumption that only primitive types are used).

In fact, from the Java viewpoint, the `String` data type is also not a primitive type. Some attributes, such as `customerAddress`, are likely to be given a user-defined non-primitive type later on (i.e. some kind of `Address` class will need to be created). For now, such attributes are typed as `String`.

The attribute `standingIndicator` is typed as `char`. This attribute captures the standing (rating) of the customer assigned to each customer based on his/her past history of payments, timely return of videos, etc. The rating can range from, say, "`A`" to "`E`", where "`A`" can mean an excellent rating and "`E`" the worst rating given to the customer (customer about to be excluded from membership).

Admittedly and understandably, there is a significant amount of arbitrary decisions in defining attributes in Figure 3-33. For example, the presence of `memberStartDate` in `Customer`, instead of `MembershipCard`, can be questioned. Similarly, the omission of `customerName` and `customerAddress` on `MembershipCard` can raise a few eyebrows.

3.2.3.3 Associations

Associations between classes establish pathways for easy object collaboration (Section 3.1.4). In the implemented system, the associations will be represented with attribute types that designate associated classes (Section 3.1.2.1.1). In the analysis model, the association lines represent associations.

Figure 3-34 demonstrates two associations between three Video Store classes – `Customer`, `Rental`, and `Payment`. Both associations are of the same one-to-many multiplicity (Section 3.1.4.2). The role names for the associations are shown. In the implemented system, the role names will be converted to attributes that designate classes (Section 3.1.2.1.1).

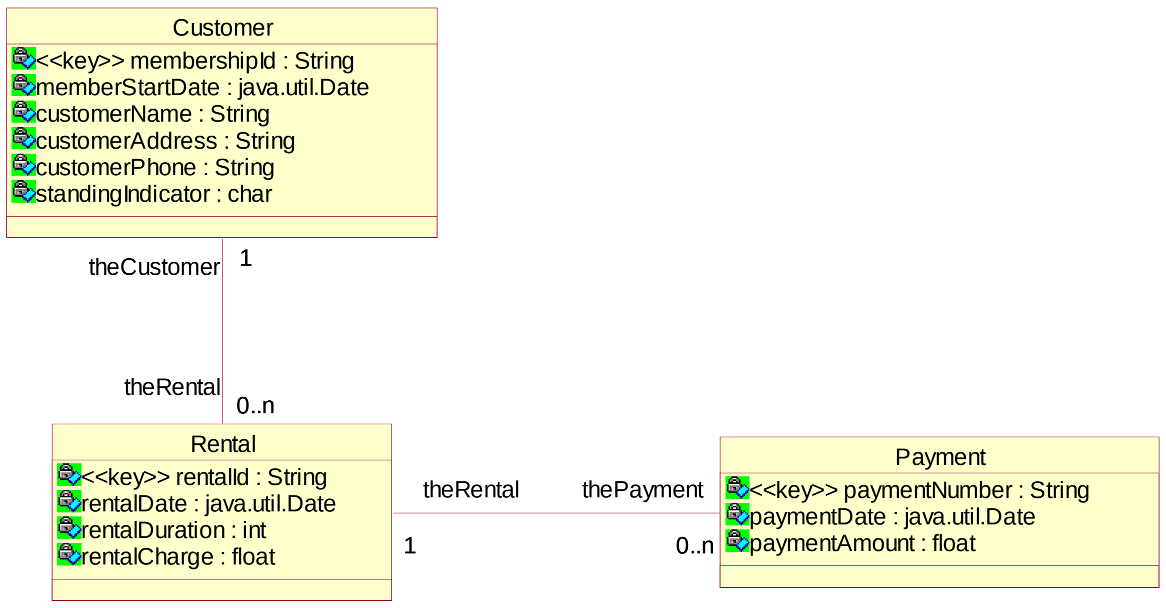


Figure 3-34 Associations (Video Store)

`Customer` can be associated with many video `Rental` transactions. Each `Rental` applies to a single `Customer`. There is no indication in the model if more than one video can be rented in a single transaction. Even if this were allowed, all videos had to be rented for the same period of time (as there is only one value of `rentalDuration` possible).

It is possible to pay for a `Rental` in more than one `Payment`. This implies that the `paymentAmount` does not have to be paid in full and can be smaller that `rentalCharge`. It is also possible to rent video without immediate payment (a `Rental` object may be associated with zero `Payment` objects). This is allowed by an alternative flow in the use case specification (Section 3.2.1.4, Table 3-2).

The model in Figure 3-34 does not include an explicit association between `Payment` and `Customer`. From the semantic point of view, such association is not necessary. The customer for a payment can be identified by "navigating" through the rental transaction. This is possible because each `Payment` object is associated with a single `Rental` object, and each `Rental` object is associated with a single `Customer`. It is likely, however, that an association between `Payment` and `Customer` will be added to the model in the design stage (this may be motivated by the considerations related to processing efficiency).

3.2.3.4 Aggregations

Aggregation and composition are stronger forms of association with ownership semantics (Section 3.1.5). In a typical commercial programming environment, aggregations and compositions are likely to be implemented like associations – with attribute types that designate associated classes (Section 2.1.2.1.1).

Figure 3-35 illustrates an aggregation relationship on classes `Customer` and `MembershipCard`. `Customer` contains zero or one `MembershipCard`. The system allows to store information about potential customers, i.e. people who do not yet have membership cards. Such potential `Customer` does not contain any `MembershipCard` and its `memberStartDate` is set to a null value (meaning that the value does not exist).

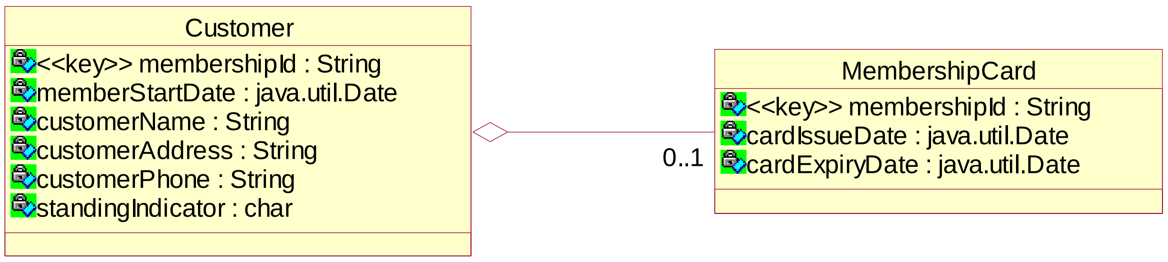

Customer
<<key>> membershipId : String
memberStartDate : java.util.Date
customerName : String
customerAddress : String
customerPhone : String
standingIndicator : char
0..1
MembershipCard
<<key>> membershipId : String
cardIssueDate : java.util.Date
cardExpiryDate : java.util.Date

Figure 3-35 Aggregation (Video Store)

The white diamond on the aggregation line does not necessarily mean that the aggregation is by reference (Section 3.1.5). It can also mean that the modeler has not decided yet about the implementation of the aggregation. If the presented diagram is an analysis model, then the implementation of the aggregation is undecided. If it is a design model, then the white diamond means indeed the aggregation by reference.

3.2.3.5 Generalizations

Generalization (Section 3.1.6) is a powerful software reuse technique, which also greatly simplifies the semantics and graphical presentation of models. The simplification is achieved in two diverse manners, depending on modeling circumstances.

By the fact that a subclass type is also the superclass type, it is possible to draw an association from any class in the model to the superclass and assume that in reality any object in the generalization hierarchy can be linked in that association. On the other hand, it is possible to draw an association to a more specific class lower in the generalization hierarchy to capture the fact that only objects of that specific subclass can be linked in the association.

Figure 3-36 is an example of generalization hierarchy rooted at the `Payment` class. Because there are only two kinds of payments allowed in Video Store (cash and card payment), the `Payment` class has become an abstract class (Section 3.1.7). `Receipt` is associated with `Payment`. In reality, objects of concrete subclasses of `Payment` will be linked to `Receipt` objects.

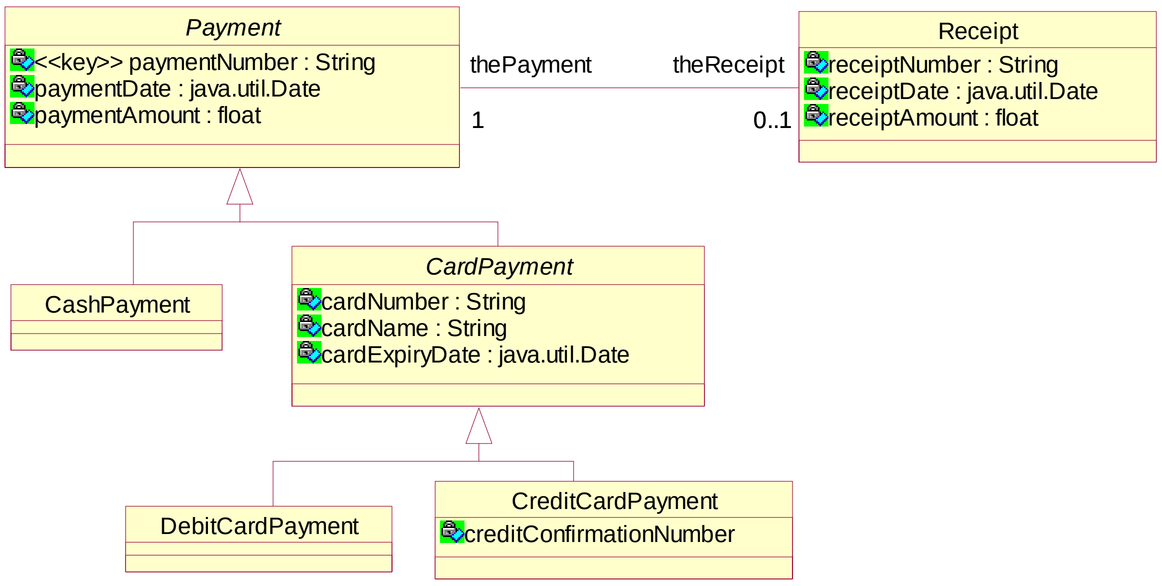


Figure 3-36 Generalization (Video Store)

The diagram in Figure 3-36 has introduced two new classes to the Video Store model. The new classes are `DebitCardPayment` and `CreditCardPayment`, which are subclasses of `CardPayment`. As a result `CardPayment` has become an abstract class.

3.2.3.6 Class diagram

The class diagram is the heart and soul of an object-oriented system. The Video Store examples so far demonstrated the *static modeling* ability of the class model. The classes contained some attributes, but no operations. The operations belong more to the design than analysis realm. When operations are eventually included in classes, the class model implicitly defines the system *behavior*.

Figure 3-37 illustrates the class diagram for Video Store. The model demonstrates all classes identified in previous examples. Apart from `Payment` and `CardPayment`, `Video` turns out to be an abstract class. All other classes are concrete.

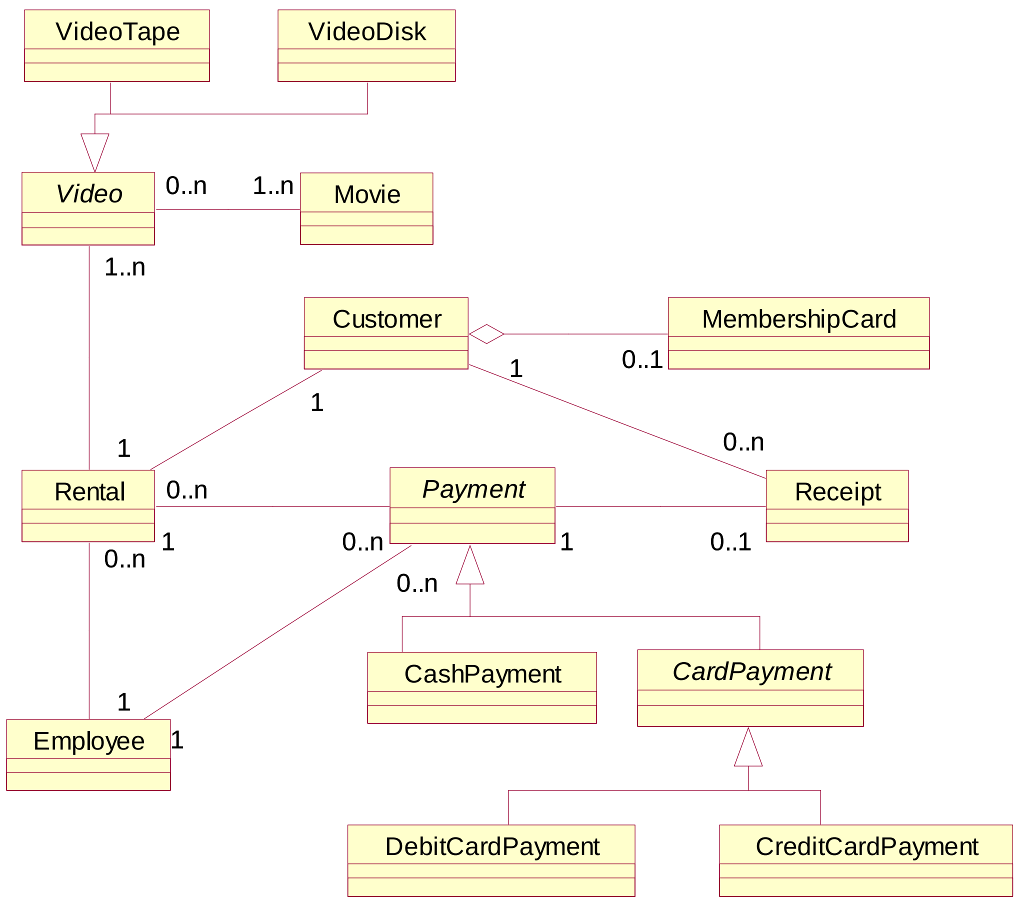


Figure 3-37 Class diagram (Video Store)

A careful analysis of the multiplicities of associations in Figure 3-37 reveals that `Rental` class refers to current rentals only. Each rented video is associated with one and only one rental transaction. Past rentals of the same video are not remembered in the `Rental` class.

A `Video` (i.e. video disk or tape) contains one or more `Movie`. A movie can be available on zero, one or more video tapes or disks.

Each rental transaction is associated with an `Employee` responsible for it. Similarly, each payment is linked to an employee. The `Customer` information for payment can be obtained by navigating from payment to customer via rental transaction or via receipt.

3.2.4 Interaction modeling

Interaction modeling captures interactions between objects needed to execute a use case or part of it. Interaction models are used in more advanced stages of requirements analysis, when a basic class model is known, so that the references to objects are backed by the class model.

The above observation underpins the main distinction between the activity modeling (see Section 3.2.2) and the interaction modeling. The *activity modeling* is frequently done at a higher level of abstraction – it shows the sequencing of events without assigning the events to objects. The *interaction modeling* shows the sequencing of events (messages) between collaborating objects.

Both activity and interaction modeling represent realization of use cases. Activity diagrams, being more abstract, frequently capture the behavior of an entire use case. Interaction diagrams, being more detailed, tend to model portions of a use case. Sometimes an interaction diagram models a single activity of the activity diagram.

There are two kinds of interaction diagrams – *sequence diagrams* and *collaboration diagrams*. They can be used interchangeably and, indeed, many CASE tools provide automatic conversion from one model to the other. The difference is in emphasis. The sequence models concentrate on time sequences and the collaboration models emphasize object relationships (cp. Rumbaugh *et al.*, 1999).

3.2.4.1 Sequence diagram

An *interaction* is a set of *messages* in some behavior that are exchanged between *objects* across *links* (persistent or transient links (Section 3.1.1.3)). The sequence diagram is a two-dimensional graph. Objects are shown along the horizontal dimension. Sequencing of messages is shown top to bottom on the vertical dimension. Each vertical line is called the object's *lifeline*. A method activated on a lifeline is called the *activation*.

Figure 3-38 shows a simple sequence diagram representing sequence of messages necessary to fulfill the activity "Verify customer" of the activity diagram in Figure 3-31 (Section 3.2.2.2). The diagram engages *objects* of four classes: `Employee`, `RentalWindow`, `CustomerVerifier`, and `Customer`. `Employee` is an actor, `RentalWindow` is a presentation class, `CustomerVerifier` is a control class, and `Customer` is an entity class (Section 3.2.3.1). Object *lifelines* are shown as vertical dotted lines. *Activations* are shown as narrow rectangles on the lifelines.

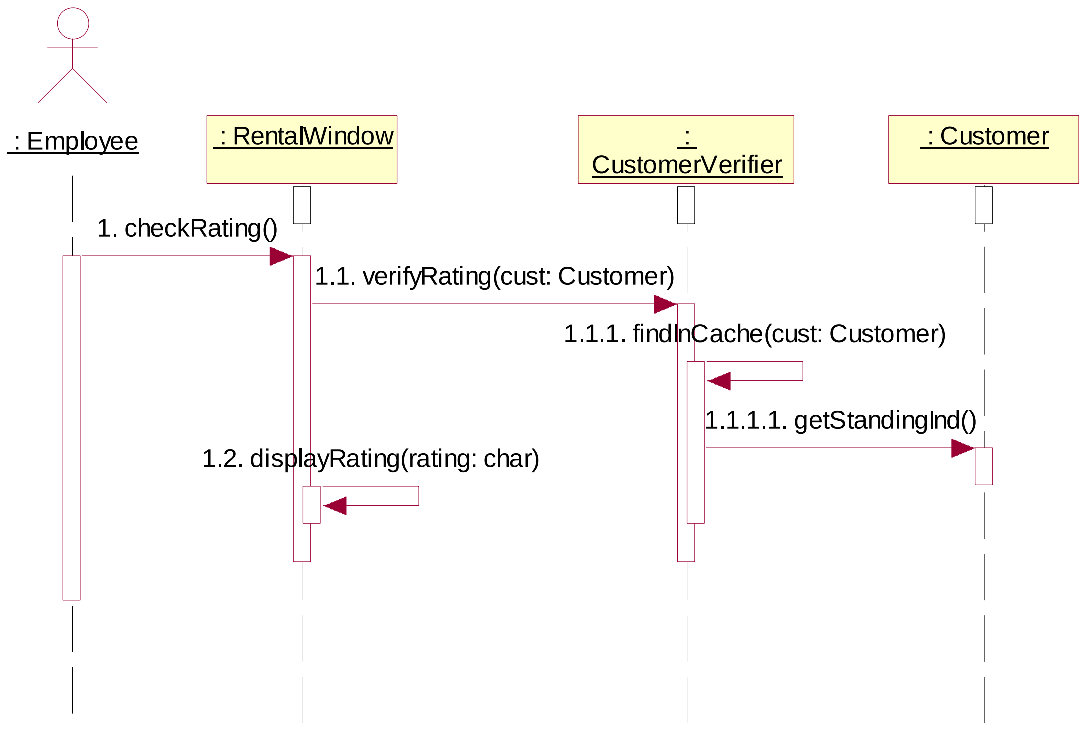

: Employee
: RentalWindow
: CustomerVerifier
: Customer
1. checkRating()
1.1. verifyRating(cust: Customer)
1.1.1. findInCache(cust: Customer)
1.1.1.1. getStandingInd()
1.2. displayRating(rating: char)

Figure 3-38 Sequence diagram for activity 'verify customer' (Video Store)

The processing begins when an `Employee` requests `RentalWindow` to `checkRating()`. When the message is received, `RentalWindow` displays information about the rental transaction being conducted for particular customer. This means that the `RentalWindow` object holds the relevant `Customer` object. Accordingly, `RentalWindow` passes the `Customer` object in `verifyRating()` message to `CustomerVerifier`.

`CustomerVerifier` is a control object responsible for the program's logic and for managing the memory cache of entity objects. Because the current rental transaction relates to a particular `Customer` object processed by `RentalWindow`, it can be assumed that the `Customer` object resides in the memory cache (i.e. it does not have to be retrieved from the database). Consequently, `CustomerVerifier` sends a *self message* (i.e. the message to its own method) to find the OID of `Customer`. This is done by the `findInCache()` message.

Once the handle (i.e. the OID) on the `Customer` object is known to `CustomerVerifier`, it requests `Customer` – in the `getStandingInd()` message – to reveal its rating. Objects returned to the original caller by the invoked methods are not explicitly shown in sequence diagrams. A *return* from a message call is implicit at the end of the object activation (i.e. when the flow of control is returned to the caller). Therefore, the value of the `Customer's standingIndicator` attribute is implicitly returned to `RentalWindow`. At this point, `RentalWindow` sends a self message to `displayRating()` for the employee's consideration.

Figure 3-38 uses hierarchical numbering of messages to show activation dependencies between messages and the corresponding methods. Note that a message to self within an activation results in a new activation. There are other important modeling facilities in sequence diagrams. They are discussed later in the book. Below is a quick recap of the main features of sequence diagrams.

An arrow represents a *message* from a calling object (*sender*) to an operation (method) in the called object (*target*). As a minimum, the message is named. *Actual arguments* of the message and other control information can also be included. The actual arguments correspond to the *formal arguments* in the method of the target object.

The actual argument can be an *input argument* (from the sender to the target) or an *output argument* (from the target back to the sender). The input argument may be identified by the keyword `in` (if there is no keyword then the input argument is assumed). The output argument is identified with the keyword `out`. The `inout` arguments are also possible but they are rare in object-oriented solutions.

As mentioned, showing the *return* of control from the target to the sender object is not necessary. The message arrow to the target object implies automatic return of control to the sender. The target knows the OID of the sender.

A message can be sent to a *collection* of objects (a collection could be a set, list, array of objects, etc.). This happens frequently when a calling object is linked to multiple receiver objects (because the multiplicity of the association is one-to-many or many-to-many). An *iteration marker* – an asterisk in front of the message label – indicates iterating over a collection.

3.2.4.2 Collaboration diagram

A collaboration diagram is an alternative representation of a sequence diagram. There is a difference in emphasis though. There are no lifelines and activities in collaboration diagrams. Both are implicit in the messages shown as arrows. Like in sequence diagrams, the hierarchical numbering of messages may help in understanding the model, but the numbering does not necessarily document the sequence of method invocations. Indeed, some models are more truthful if no numbering is used.

Figure 3-39 is a collaboration diagram corresponding to the sequence diagram in Figure 3-38. CASE tools are able to automatically convert any sequence diagram to a collaboration diagram (and vice versa).

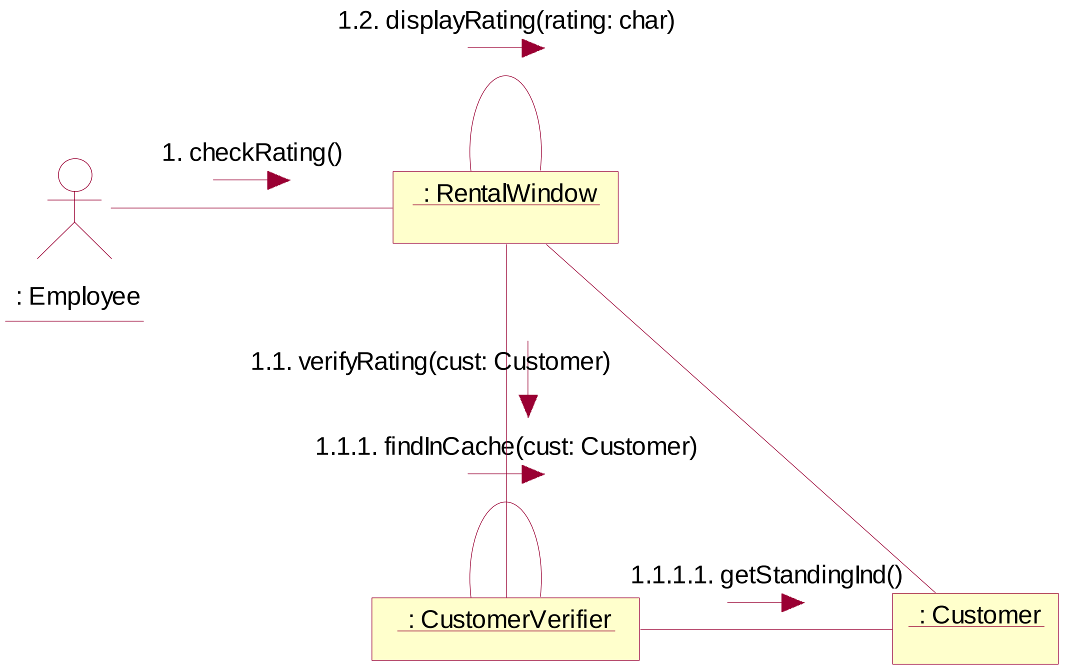

1.2. displayRating(rating: char)
1. checkRating()
: RentalWindow
: Employee
1.1. verifyRating(cust: Customer)
1.1.1. findInCache(cust: Customer)
1.1.1.1. getStandingInd()
: CustomerVerifier
: Customer

Figure 3-39 Collaboration diagram for activity 'verify customer' (Video Store)

In general, collaboration diagrams tend to be more useful graphically when representing models involving many objects. Also, and unlike sequence diagrams, the solid lines between objects may (and should) indicate the need for associations between classes of these objects. Building such associations legitimates the fact that the objects of these classes communicate.

3.2.4.3 Class methods

Examining the interactions can lead to the discovery of methods (operations) in classes. The dependency between interactions and operations is straightforward. Each message invokes a method on the called object. The operation has the same name as the message.

Of course, the one-to-one mapping between *messages* in interaction models and *methods* in implemented classes is helpful only to the point to which the interaction model constitutes a detailed technical design – something neither possible nor desirable in the analysis phase. Additional methods will be defined during detailed design and during implementation.

As an aside, note that similar one-to-one mapping exists between *messages* and *associations*, in particular for messages sent between *persistent* (*entity*) objects. Such messages should be supported by persistent links (Section 3.1.1.3.1). Similar thinking should apply to transient in-memory objects, which of course includes entity objects loaded in memory (Section 3.1.1.3.2). Therefore, the presence of a message in a sequence diagram stipulates the need for an association in the class diagram.

Figure 3-40 illustrates how interactions can be used to add operations to classes. The messages received by objects in the sequence diagram translate to methods (operations) in the classes representing these objects. The class diagram reveals also the return types and visibility of methods. These two characteristics of methods are not apparent in sequence diagrams.

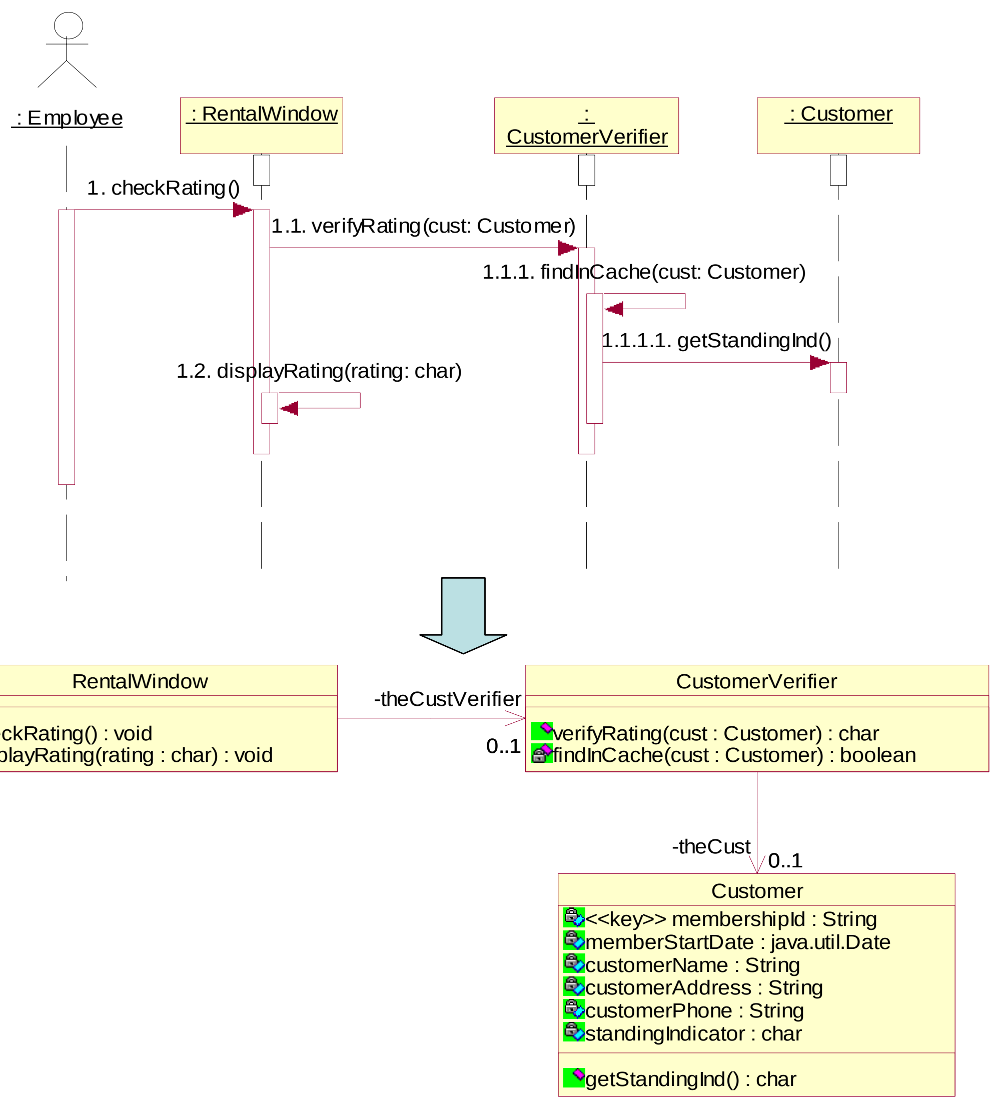

: Employee
: RentalWindow
: CustomerVerifier
: Customer
1. checkRating()
1.1. verifyRating(cust: Customer)
1.1.1. findInCache(cust: Customer)
1.1.1.1. getStandingInd()
1.2. displayRating(rating: char)
RentalWindow
checkRating() : void
displayRating(rating : char) : void
-theCustVerifier
0..1
CustomerVerifier
verifyRating(cust : Customer) : char
findInCache(cust : Customer) : boolean
-theCust
0..1
Customer
<<key>> membershipId : String
memberStartDate : java.util.Date
customerName : String
customerAddress : String
customerPhone : String
standingIndicator : char
getStandingInd() : char

Figure 3-40 Using interactions to add operations to classes (Video Store)

RentalWindow receives the request to checkRating() and delegates this request to CustomerVerifier's verifyRating(). Because RentalWindow holds the handle on the Customer object that it currently processes (displays), it passes this object in the argument of verifyRating(). The delegation itself uses the association link to CustomerVerifier. The association is conducted via the role theCustVerifier, which will be implemented as a private attribute in RentalWindow (the private visibility is indicated by the minus sign in front of the role name).

The verifyRating() method utilizes the private method findInCache() to ascertain that the Customer object is in memory and to set the theCust attribute to reference this Customer object (if the attribute has not been previously set). Consequently, CustomerVerifier asks Customer to getStandingInd() by reading its attribute standingIndicator. The char value of this attribute is returned all the way to RentalWindow's checkRating().

To display customer's rating in the GUI window under the control of `RentalWindow`, `checkRating()` sends a message to `displayRating()`, passing the rating value along. The `displayRating()` method has private visibility because it is called within `RentalWindow` by a self message.

3.2.5 Statechart modeling

An *interaction model* provides a detailed specification for a use case, a part of it, or for one or more activities. A *statechart model* specifies dynamic changes in a class. It describes various states in which objects of the class can be. These dynamic changes describe the behavior of an object across all use cases that involve the class from which the object is instantiated.

A *state* of an object is designated by the current values of the object's attributes (both primitive attributes and attributes that designate other classes). A statechart model captures the life history of the class. An object is one and the same during its lifetime – its identity never changes (Section 3.1.1.3). However, the state of an object changes. A statechart diagram is a bipartite graph of *states* and *transitions* caused by *events*.

3.2.5.1 States and transitions

Objects change values of their attributes but not all such changes cause *state transitions*. Consider a `BankAccount` object and an associated business rule that the bank fees on an account are waived when the account's `balance` exceeds $100,000. We can say that `BankAccount` then enters a `privileged` state. It is in a `normal` state otherwise. The account's `balance` changes after each withdrawal/deposit transaction, but the *state* changes only when the balance goes above or below $100,000.

The above example captures the essence of state modeling. State models are constructed for classes that have interesting state changes, not any state changes. What is 'interesting,' or not, is a business modeling decision. A statechart diagram is a model of business rules. The *business rules* are invariable over some periods of time. They are relatively independent of particular use cases. In fact, use cases must also conform to business rules.

As an example consider the class `Rental` in the Video Store case study. `Rental` has an attribute (`thePayment`) that associates it with `Payment` (Figure 3-34). Depending on the nature of this association, a `Rental` object can be in different states as far as the payments for hiring a video are concerned.

Figure 3-41 is a statechart model for the class `Rental`. The states are depicted as rounded rectangles. Events are shown as arrows. The initial state (pointed to by the arrow with black circle) of `Rental` is `Unpaid`. There are two possible transitions out of the `Unpaid` state. On the `partial payment` event, the `Rental` object goes into the `Partly Paid` state. Only one `partial payment` is allowed, according to the model. The `final payment` event, when in an `Unpaid` or `Partly Paid` state, fires a transition to `Fully Paid` state. This is the final state (indicated by a black circle with another circle around it).

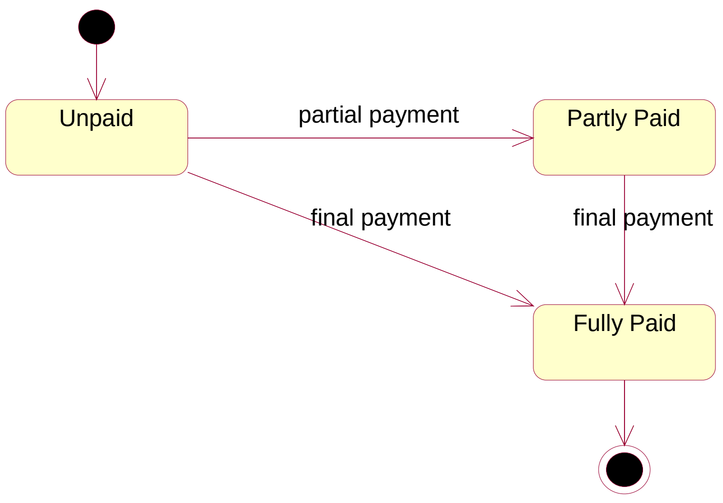


Figure 3-41 States and events for the class Rental (Video Store)

3.2.5.2 Statechart diagram

A statechart diagram is normally attached to a class, but in general, it can be attached to other modeling concepts, e.g. a use case. When attached to a class, the diagram determines how objects of that class react to events. More precisely, it determines – for each object state – what *action* the object will perform when it receives an event. The same object may perform a different action for the same event depending on the object's state. The action's execution will typically cause a state change.

The complete description of a *transition* consists of three parts:

```
event(parameters) [guard] / action
```

Each part is optional. It is possible to omit all of them, if the transition line by itself is self-explanatory.

The event is a quick occurrence that affects the object. It can have parameters, e.g. `mouse button clicked (right_button)`. The event can be guarded by a condition, e.g. `mouse button clicked (right_button) [inside the window]`. Only when the condition evaluates to 'true' does the event fire and affects the object.

The distinction between an event and a guard is not always obvious. The distinction is that the *event* 'happens' and it may be even saved before the object is ready to handle it. At the point of handling the event, the *guard* condition is evaluated to determine if a transition should fire.

The *action* is a short atomic computation that executes when the transition fires. An action can also be associated with a state. In general, an action is an object's response to a detected event. The states can additionally contain longer computations – called *activities*.

States can be composed of other states – *nested states*. The *composite state* is abstract – it is simply a generic label for all nested states. A transition taken out of a composite state's boundary means that it can fire from any of the nested states. This improves the clarity and expresiveness of the diagram. Of course, a transition out of a composite state's boundary can also be fired from a nested state.

Consider again the class `Rental` in the Video Store case study and think about all states in which `Rental` can be (not just with regard to payments, as in Figure 3-41, but with regard to all attributes in `Rental`). Figure 3-42 illustrates a state diagram for the class Rental. The diagram is purposefully drawn to take advantage of a variety of state modeling features.

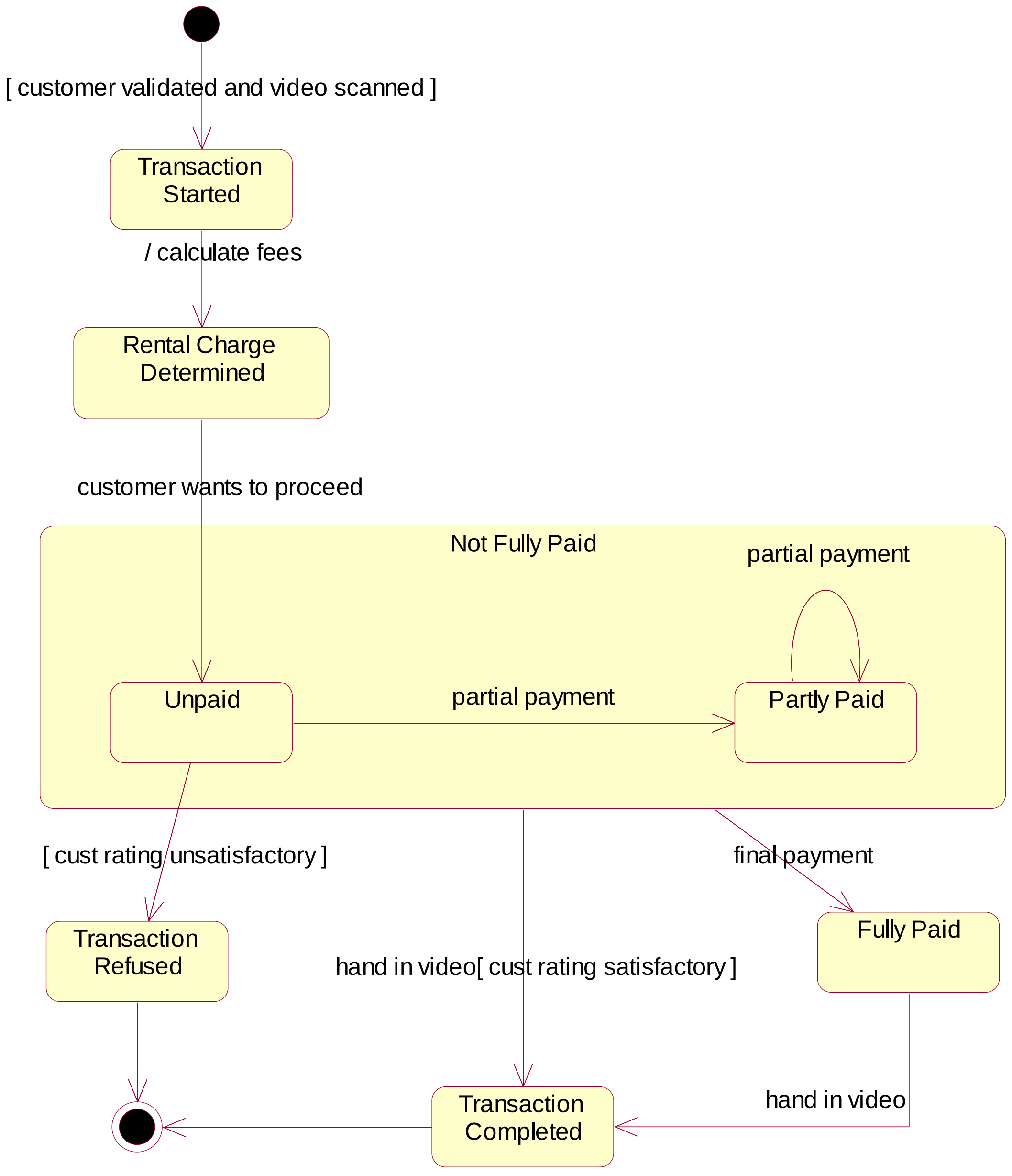


Figure 3-42 State diagram for the class Rental (Video Store)

The state model in Figure 3-42 enters the state `Transaction Started` once the *guard condition* ‘customer validated and video scanned’ is true. The transition to the next state (`Rental Charge Determined`) requires firing the *action* ‘calculate fees’. On the *event* ‘customer wants to proceed’ the `Rental` object enters the `Unpaid` state.

The `Unpaid` state is one of two nested states within the composite state `Not Fully Paid`. The other nested state is `Partly Paid`. The `Partly Paid` state accepts the *transition to self* with the event 'partial payment'. This allows multiple partial payments before the transaction is paid in full.

In the `Unpaid` state, when the guard condition 'cust rating unsatisfactory' is evaluated to true, a transition if fired into the state `Transaction Refused`. This is one of two possible final states.

The second final state is `Transaction Completed`. This state is achievable in two ways. Firstly, and preferably, the event 'final payment' from the state `Not Fully Paid` results in the state `Fully Paid`. When in this state, the event 'hand in video' places the `Rental` object in the state `Transaction Completed`. The second possibility to achieve the state `Transaction Completed` is by the transition from the state `Not Fully Paid`, which is fired by event 'hand in video' under the condition 'cust rating satisfactory'.

3.2.6 Implementation models

UML provides tools for architectural/structural modeling of physical implementation of the system. The two main tools are component diagrams and deployment diagrams (Alhir, 2003; Maciaszek and Liong, 2004).

"A *component* represents a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces." (UML, 2003, p.3-174). The *component diagram* is concerned with modeling the structure and dependencies of components in the implemented systems.

"A *node* is a physical object that represents a processing resource, generally, having at least a memory and often processing capability as well. Nodes include computing devices but also human resources or mechanical processing resources." (UML, 2003, p.3-173). The *deployment diagram* is concerned with modeling the structure and dependencies of nodes that define the implementation environment of the system.

Implementation models are in the physical modeling realm, but they must be defined with due consideration to logical structures of the system. The main logical building block is the *class* and the main logical structural model is the class diagram. Other logical structuring concepts are the notions of *subsystem* and *package*.

3.2.6.1 Subsystems and packages

The old Roman dictum, "divida et impera" (divide-and-conquer or divide-and-rule), recommends that a position of power can be achieved by isolating the adversaries and working towards causing disagreements between them. In problem solving, this dictum is frequently used in a slightly different meaning. It requires that a large problem be divided into smaller problems so that, once the solution to the smaller problem is found, a larger problem can be addressed.

The "divida et impera" principle leads to hierarchical modularization of the problem space. In system development, it results in the division of the system into subsystems and packages. The division has to be carefully planned to reduce dependencies in the hierarchy of subsystems and packages.

The notion of *subsystem* specializes (inherits from) the concept of component (Ferm, 2003). Subsystem encapsulates some part of intended system behavior. The services that a subsystem provides are the result of the services provided by its internal parts, i.e. classes. This also means that a subsystem is not instantiable (Selic, 2003).

The services of the subsystem can and should be defined using *interfaces* (Section 3.1.8). The benefits of encapsulating behavior and providing services through interfaces are many. They include insulation from change, replaceable implementation of services, extendibility and reusability.

Subsystems can be structured in architectural layers such that dependencies between layers are acyclic and minimized. Within each layer, subsystems can be nested. This means that a subsystem can contain other subsystems.

A *package* is a grouping of modeling elements under an assigned name. Like subsystem, the services that a package provides are the result of the services provided by its internal parts, i.e. classes. Unlike subsystem, the package does not reveal its behavior by exposing interfaces. As noted by Ferm (2003, p.2): "The difference between a subsystem and a package is that, for a package, a client asks *some element inside the package* to fulfill a behavior; for a subsystem, a client asks *the subsystem itself* to fulfill the behavior."

Like subsystem, a package may contain other packages. Unlike subsystem, a package can be directly mapped to a programming language construct, e.g. a Java package or a .NET namespace. Like subsystem, a package owns its members. A member (class, interface) can belong to only one direct subsystem or package.

All in all, a subsystem is a richer concept that embodies both the structural aspects of packages and behavioral aspects of classes. The behavior is provided through one or more interfaces. The clients requests subsystem's services through these interfaces.

As Figure 3-43 illustrates, in UML the graphical difference between subsystem and package is only through the use of the stereotype `«subsystem»`. Subsystem is a package stereotyped as `«subsystem»`.

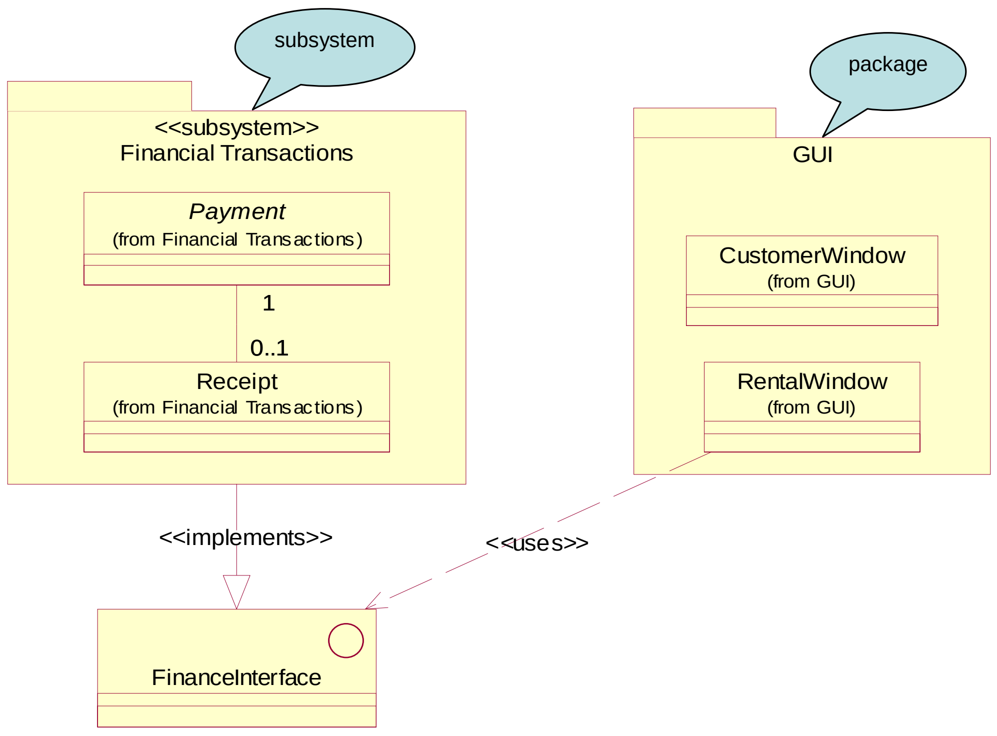


Figure 3-43 Subsystem and package

The fact that a subsystem encapsulates its behavior is represented by providing interfaces. The interface `FinanceInterface` is implemented by the subsystem `Financial Transactions` and it is used by the class `RentalWindow` in the package `GUI`.

In general, to minimize system dependencies, interfaces should be placed outside subsystems that implement them (Maciaszek and Liong, 2004). Although not shown in Figure 3-43, it is allowed to place all or most interfaces in a package that exclusively holds interfaces. Alternatively, if `GUI` classes in Figure 3-43 were the only classes using `FinanceInterface`, then the interface could be located in the `GUI` package.

3.2.6.2 Component diagram

Although a subsystem can be viewed as a specialization of the concept of component, the UML uses a separate graphical element to visualize components. The visualization is different from the stereotyped package (like in Figure 3-43 that shows a subsystem). A *component* is visualized as a rectangular box with two smaller rectangles. The rectangles provide the interface analogy (as component implements its public interface).

Figure 3-44 is an example of the component diagram. The example shows two components: `Cache Handler` and `Database Access`. `Cache Handler` uses the interface `DBInterface` implemented by `Database Access`.

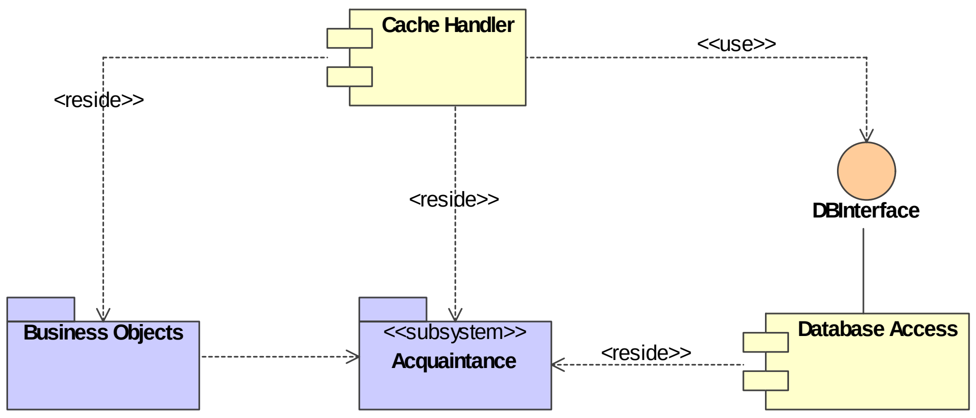


Figure 3-44 Component diagram

A component may be linked with a «`reside`» dependency to other implementation-time modeling elements. Possible elements are class, package, subsystem or other component. The meaning of the dependency is that the *client* component depends on the *supplier* element and that the supplier element resides in the component (admittedly, the direction of the arrow is somewhat counterintuitive as far as "what resides in what" is concerned). The same modeling element may reside in more than one component (Alhir, 2003), as shown for the subsystem `Acquaintance`. (Later in the book it will be revealed that the `Acquaintance` subsystem contains only interfaces, and not any classes.)

In Figure 3-44, the package `Business Objects` and the subsystem `Acquaintance` reside in `Cache Handler`. To put it another way, `Cache Handler` depends on `Business Objects` and `Acquaintance`. Additionally, `Business Objects` depends on `Acquaintance`.

3.2.6.3 Deployment diagram

Deployment diagram defines the deployment of components and other run-time processing elements on computer nodes (Alhir, 2003). The node can be any server (e.g. a printer, email, application, web, or database server) or any other computing or even human resource available to components and other processing elements.

Nodes are denoted as three-dimensional boxes. A «`deploy`» dependency from a client component to a supplier node defines the deployment. Figure 3-45 shows that `Cache Handler` is deployed on `Application Server`, and `Database Access` (together with its `DBInterface`) on `Database Server`.

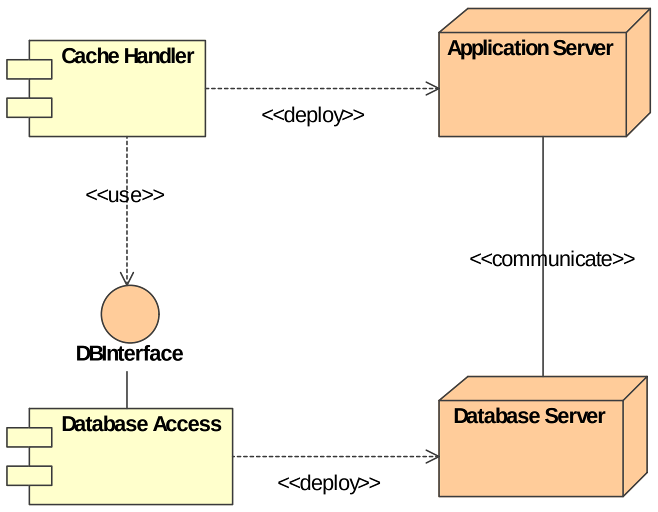


Figure 3-45 Deployment diagram

Summary

This chapter has covered quite a lot of ground. It explained the fundamental terminology and concepts of object technology. It also introduced all major UML models and diagrams and illustrated them using a single case study – Video Store. For a novice to the topic the task must have been daunting. The rewards will come in the following chapters.

An object system consists of collaborating *instance objects*. Each object has a state, behavior, and identity. The concept of identity may well be the most critical for proper understanding of object systems – it is also the most difficult to appreciate for people with some luggage of experience in conventional computer applications. Navigation along *links* is the modus operandi of object technology – something that a reader educated on the relational database technology may have difficulty to digest.

A *class* is the template for object creation. It defines the *attributes* that an object can contain and the *operations* that an object can invoke. Attributes can have primitive types or can designate other classes. The attributes that designate other classes declare the *associations*. The association is one kind of relationship between classes. Other kinds are *aggregation* and *generalization*.

A class may have attributes or operations that apply to the class itself, not to any one of its instance objects. Such attributes and operations require the notion of a *class object*. Attributes and operations defined in classes during system analysis are referred to during system design/implementation as *variables* and *methods*, respectively.

A complete method prototype includes the method name, its *signature* (the list of formal arguments), and its return type. A *constructor* is a special method that serves the purpose of instantiating objects of the class.

An *association* relationship provides a linkage between objects of given classes. Association *degree* defines the number of classes connected by the association. Association *multiplicity* defines how many objects may be represented by a *role name*. An association *link* is an instance of the association. An association that has attributes (and/or operations) of its own is modeled as an *association class*.

An *aggregation* is a whole-part relationship between a class representing an assembly of components (*superset class*) and the classes representing the components (*subset classes*). In UML, the aggregation by value is called *composition*, and the aggregation by reference is simply called *aggregation*. Aggregation/composition is frequently implemented by means of *buried references*. Java provides an alternative implementation by means of *inner classes*. Associated with aggregation/composition is the powerful technique of *delegation*.

A *generalization* relationship is a kind-of relationship between a more generic class (*superclass* or *parent*) and a more specialized kind of that class (*subclass* or *child*). Generalization provides the basis for *polymorphism* and *inheritance*. *Overriding* is the mechanism to achieve polymorphic operations. Commercial programming environments can support *multiple inheritance*, but they do not normally support *multiple* or *dynamic classification*.

Related to inheritance are the notions of abstract class and interface. An *abstract class* is a class that may have partial implementations (some operations may be declared), but which cannot be instantiated. An *interface* is a definition of a semantic type with attributes (constants only) and operations but without any implementation. A class that inherits the interface must provide the implementation.

The UML standard distinguishes three kinds of models – state, behavior and state change models. The *use case model* is the main UML representative and the focal point of behavior modeling. The model defines use cases, actors, and relationships between these modeling elements. Each use case is specified in a text document.

The *activity model* can graphically represent the flow of events of a use case. Activity models fill a gap between a high-level representation of system behavior in *use case models* and much lower-level representation of behavior in *interaction models* (sequence and collaboration diagrams).

Class modeling integrates and embodies all other modeling activities. Class models identify classes and their attributes, including relationships. Classes belong to various architectural layers. Typical groups of classes are presentation, control, entity, mediator, and foundation classes.

Interaction modeling captures interactions between objects needed to execute a use case or part of it. There are two kinds of interaction diagrams – *sequence diagrams* and *collaboration diagrams*. The sequence models concentrate on time sequences and the collaboration models emphasize object relationships. There is one-to-one mapping between *messages* in interaction models and *methods* in implemented classes.

A *statechart model* specifies dynamic changes in a class. It describes various states in which objects of the class can be. These dynamic changes describe the behavior of an object across all use cases that involve the class from which the object is instantiated. A statechart diagram is a bipartite graph of *states* and *transitions* caused by *events*.

UML provides *component diagrams* and *deployment diagrams* as two tools for architectural/structural modeling of physical implementation of the system. The notions of subsystem and package are related architectural design concepts referred to in implementation models.

Questions

Q1 Why do we need to distinguish between an instance object and a class object?

Q2 What is an object identifier? How can it be implemented?

Q3 What is the distinction between a transient object and a persistent object?

Q4 What are a transient link and a persistent link? How are they used during the program execution?

Q5 What does it mean that an attribute type designates a class? Give an example.

Q6 Why are most attributes private and most operations public in a good object model?

Q7 What is the difference between the operation visibility and scope?

Q8 What is the difference between 'public static' and 'private static' members as far as their accessibility is concerned? Give an example.

Q9 In what modeling situations must an association class be used? Give an example.

Q10 'Buried reference' and 'inner class' are two mechanisms to implement aggregation/composition. Do these mechanisms provide sufficient and complete implementation of the semantics assumed in aggregation/composition? Explain.

Q11 Explain the observation that in a typical object programming environment the inheritance applies to classes, not to objects.

Q12 What is the connection between overriding and polymorphism?

Q13 How is the multiple classification different from multiple inheritance?

Q14 What are the modeling benefits of an abstract class versus the modeling benefits of an interface? Explain by exemplification.

Q15 Explain the main characteristics and complementary properties of a state model, behavior model, and state change model.

Q16 Can an actor have attributes and operations? Explain.

Q17 Explain the role and place of activity diagrams in system modeling.

Q18 What are entity classes? What other categories of classes need to be distinguished in class modeling? Explain.

Q19 What is an actual argument and what is a formal argument?

Q20 What is the difference between an action and an activity in statechart diagrams? Exemplify.

Q21 Explain why subsystems implement interfaces and packages do not. What would be the consequences for implementation models if the subsystems, they refer to, did not implement interfaces?

Exercises

E1 Refer to Figure 3-2 (Section 3.1.1.2). Consider the following changes to the logic of object collaboration for shipping products and replenishing the stock.

The shipment and replenishment are separate processing threads. When `Order` creates new `Shipment` object and requests its shipment, `Shipment` obtains `Product` objects as in Figure 3-2. However, instead of `Stock` managing changes to product quantities, `Shipment` uses its handle on `Product` objects and directly requests `Product` to `getProdQuantity()`.

In this scenario, `Product` knows its quantity and when it has to be reordered. Consequently, when replenishment is needed, a new `Purchase` object is created to provide the `reorder()` service.

Modify the diagram in Figure 3-2 to capture the above changes. There is no need to show messages that instantiate `Shipment` and `Purchase` objects, because the topic of object creation has not been sufficiently explained yet.

E2 Refer to Figure 3-2 (Section 3.1.1.2) and to Figure 3-46 (i.e. the solution to exercise E1). Define the return types for all messages in both diagrams. Explain how the return values are used in successive calls.

E3 Refer to Figure 3-14 (Section 3.1.4.2). Suppose that a course offering includes lectures and tutorials and that it is possible that one teacher is in charge of the lecturing portion of the course offering and another teacher is in charge of the tutorials.

Modify the diagram in Figure 3-14 to capture the above fact.

E4 Refer to Figure 3-16 (Section 3.1.4.4). Provide an alternative model that does not use an association class and that does not use a ternary association (that is not recommended in the book). Describe semantic differences, if any, between the model in Figure 3-16 and your new model.

E5 Refer to Figure 3-46 below, in which `Book` contains `Chapter` objects and each `Chapter` contains `Paragraph` objects. The text of the book is stored within `Paragraph` objects. The `Finder` class is a control class. It depends on the `Book` class (this is indicated by an arrowed dotted line). Consider that `Finder` needs to display to screen all `chptNumber` and `paraNumber` that contain a particular search string within `text`.

Add operations to classes that are needed for such processing.

Draw an object collaboration diagram and explain how the processing is done, including return types of operations.

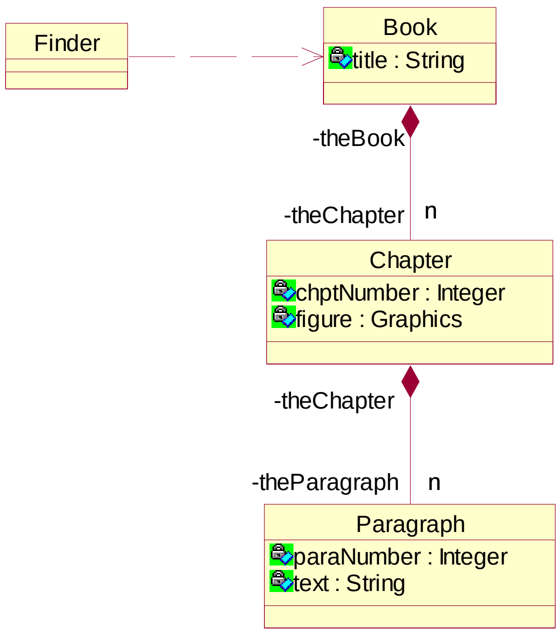


Figure 3-46 Aggregation for Book class

E6 Refer to Figure 3-46 and to the solution to exercise E5 in Figure 3-49.

Extend the class diagram by adding operations to classes that are needed for such processing.

Provide pseudo code or Java code for classes `Finder`, `Book` and `Chapter`.

E7 Refer to Figure 3-22 (Section 3.1.6.3).

Extend the example by adding attributes to the classes `Teacher`, `Student`, `PostgraduateStudent`, and `Tutor`.

E8 Refer to Figures 3-24 (Section 3.1.8.1) and 3-25 (Section 3.1.8.2).

Extend Figures 3-24 and 3-25 to consider further specialization of `VideoMedium` to differentiate between the following categories of video systems: `Beta` and `VHS` for tapes and `DVD-V` and `VCD` for disks.

Exercises (Video Store)

F1 The use case diagram in Figure 3-29 (section 3.2.1.3) does not make it explicit that the model is about renting videos (it talks only about scanning cards and video devices, accepting payments and printing receipts). In practice, a use case will be required that a rental transaction is conducted and is eventually recorded in the

Video Store database. Moreover, the use case diagram does not check the age eligibility of customers (customers must be over 18 years old to be eligible to rent movies rated R or X).

Extend the use case diagram to include the above considerations. Also, take into account that the process of renting video starts after the scanning of the customer card and of video devices took place and that it is possible (1) to rent a video without payment (in some special cases), (2) age eligibility is checked if the customer is not 18 years old and movies are rated R or X, and (3) to rent a video with or without receipt (depending on customer's request).

F2 Refer to the solution to Exercise F1 (Figure 3-51) and to the activity diagram in Figure 3-31 (Section 3.2.2.2). Develop a complementary activity diagram for the use case `Rent Video` and the subordinate use cases that extend `Rent Video` (there is no need to repeat the specifications for `Accept Payment`, already developed in Figure 3-31).

F3 Refer to Figure 3-34 (Section 3.2.3.3). Assume that not every `Payment` is linked to `Rental`. Some videos are for sale and Payment may be related to the `Sale` class. Assume also that a single payment can pay for more than one rental. How do these changes impact on the model? Modify and extend the model.

F4 Refer to Figure 3-31 (Section 3.2.2.2). Draw a sequence diagram for the activity `Handle card payment`.

F5 Refer to the class diagram Figure 3-37 (Section 3.2.3.6) and consider the class `Video`. Apart from the information that can be obtained from the class model, consider that Video Store sells videos, previously available for rent, once the movie activity threshold reaches certain level (such as when videos with that movie have not been rented out for a week). Consider also that videos are checked regularly if they are operational and may be written off if damaged.

Develop a statechart diagram for the class `Video`.

F6 Refer to the discussion on implementation models in Section 3.2.6, and in particular to the ways various dependency relationships are used in component and deployment diagrams. Consider the implementation model in Figure 3-47 with some dependency relationships missing.

Note that the placement of `VR Interface` inside the `Video Rental` subsystem box means that `Video Rental` provides (implements) this interface. An unlabeled arrow from `Inventory Control` to `Video Rental` is a generic dependency (`Inventory Control` depends on `Video Rental`).

Correct the diagram in Figure 3-47 to show all dependencies.

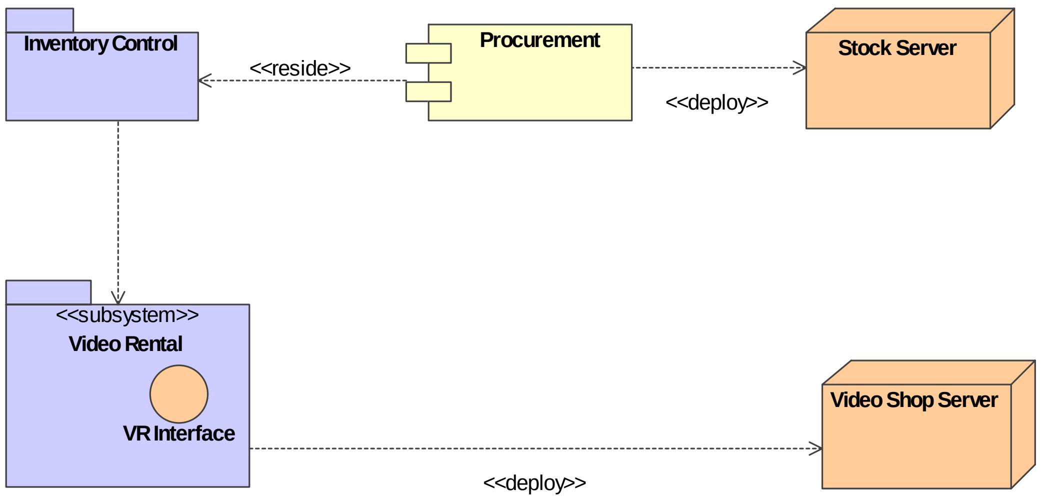


Figure 3-47 Implementation model with some dependencies missing

Answers to odd-numbered questions

Q1 - answer

Most processing in an object system is accomplished by collaboration of instance objects. *Instance objects* send messages to other instance objects to activate their methods. Hence, a `Student` instance object can send a message to an `Instructor` instance object to request consultation. There may be many `Instructor` objects but the `Student` instance is interested to talk to a specific `Instructor` instance.

At times, however, a message needs to be sent to a group of instance objects. For example, a `Student` may need to get the list of all `Instructor` instance objects before deciding on instance from which the consultation will be requested. An `Instructor` *class object* is the only object that knows about its `Instructor` instance objects. The `Student` instance object should therefore send the message to the `Instructor` class object to obtain the list of `Instructor` instance objects.

In general, a class object contains services (methods) that implement efficient access to all instance objects of that class. Such services are essential in order to obtain the current list of instance objects and to perform statistical calculations on instance objects (such as sums, counts, averages). Equally importantly, a class object is responsible for *creation* of new instance objects.

Q3 - answer

A *transient object* is created and destroyed within a single execution of the program. A *persistent object* outlives the execution of the program. A persistent object is stored in a persistent storage, typically in a database held on a magnetic disk.

A persistent object is read to program's memory for processing and may be stored back to persistent storage before the program terminates. Of course, the program may choose to destroy persistent object, thus removing it from the database.

Q5 - answer

An attribute of a class can take values of a built-in type or a user-defined type. The set of *built-in types* supported by an object programming environment is known a priori. The built-in types can be *atomic* (such as `int` or `boolean`) or *structured* (such as `Date` or `Time`).

A *user-defined type* designates a new *object type* required by the application. That type can be used to implement a *class* so that *objects* of that class can be instantiated.

An *attribute* can be assigned a user-defined object type. A value of such attribute is an OID to an object of that user-defined type. An attribute is linked to an object of another (or the same) class. We say that the attribute type designates a (user-defined) class.

An example is an attribute called `theCust` in the class called `Invoice`. The type of that attribute may be a class called `Customer`.

Q7 - answer

The *operation visibility* declares the ability of an outside object to reference the operation. The visibility can be `public`, `private`, `package` (the default visibility in Java), or `protected` (related to the notion of inheritance).

The *operation scope* declares if the owner of the operation is an instance object (*object scope*) or a class object (*class scope*). Constructor operations, which instantiate new objects, have necessarily class scope. The class scope implies centralized global information about instance objects and should be carefully controlled, in particular in a distributed object system.

Q9 - answer

An *association class* must be used when the association itself has properties (attributes and/or operations). This frequently occurs on many-to-many associations, and sometimes on one-to-one associations. An association class on a one-to-many association is unusual.

A many-to-many association between classes `Employee` and `Skill` may demand an association class if we want to store such information as `dateSkillAcquired`. The one-to-one association between `Husband` and `Wife` may result in an association class to store `marriageDate` and `marriagePlace`.

Q11 - answer

In a typical object programming environment, *inheritance* is an incremental definition of a class. It is a mechanism by which more specific classes incorporate definitional elements (attributes and operations) defined by more general classes. As such, inheritance applies to *types* (classes), not to objects.

Objects are instantiated after inherited elements have been added to the class definition. The values of inherited elements are instantiated the same way as the values of non-inherited elements.

In general, it is possible to imagine that inheritance can be extended to allow the *inheritance of values* (i.e. *inheritance of objects*). Value inheritance may be useful for setting default values of attributes.

For example, a `Sedan` object inheriting from a `Car` object may inherit the default number of wheels (four), the default transmission (automatic), etc. These values may be modified (overridden), if necessary. Some knowledge base tools support inheritance of values.

Q13 - answer

Multiple classification is a program execution regime in which an object is an instance of (and it, therefore, directly belongs to) more than one class. *Multiple inheritance* is a semantic variation of generalization in which a class can have more than one parent class (and it, therefore, inherits from all parent classes).

Multiple classification is not supported by popular object programming languages. This creates problems whenever an object can play multiple roles, such as a `Person` who is a `Woman` and a `Student`. Without multiple classification we may have to use multiple inheritance to create a specialized class `FemaleStudent`.

Defining classes for each legal combination of parent classes is not necessarily a desirable option, in particular if the objects are likely to change roles over time. Multiple classification can be combined with *dynamic classification* to allow objects to change classes during run time.

Q15 - answer

The *state model* describes the static structure of a system – classes, their internal structure, and their relationships to each other. The *behavior model* describes the actions of the objects in a system in support of business functions – class methods and object collaborations. The *state change model* describes the dynamic changes in object states over their lifetimes.

The three models offer different but complementary viewpoints on frequently the same modeling elements. The state view shows the kinds of elements that exist in a system. The behavior view ensures that the elements are capable of executing the required system functionality. However, a good state model should be able to gracefully accommodate new or extended system functionality. The state change view defines the framework for object evolution – the constraints on objects states that both the behavior and state model must conform to.

Q17 - answer

In older versions of UML, *activity diagram* were considered a special case of a *state machine* and could be even used in lieu of statechart diagram. To this aim, activity diagrams were extending the statechart notation and distinguished between *object states* (as in statecharts) and *activity states* (not modeled directly in statecharts). In the current UML, activity diagrams define only behavior by using the control and data flow model reminiscent of Petri nets (Ghezzi *et al.*, 2003).

Unlike most other UML modeling techniques, the role and place of activity diagrams in system development process is not clear-cut. The semantics of activity diagrams makes them usable at various levels of abstraction and in different phases of the lifecycle. They can be used in early analysis to depict overall behavior of the system. They can be used to model the behavior of a use case or any part of it. They can also be used in design to give a flowchart-like specification of a specific method or even individual algorithms contained in a method.

All in all, activity diagrams can be seen as "gap-fillers" in the system model. They are used as complementary technique to provide graphical visualization for a flow of events, data, and processes within various other modeling elements.

Q19 - answer

Objects communicate by sending a message from one object to call a method (operation) in another (or the same) object. A message signature includes a list of *actual arguments*. The method being invoked includes in its signature a corresponding list of *formal arguments*.

In UML, the actual arguments of a message are called simply *arguments*, but the formal arguments of a method are called *parameters*.

Q21 - answer

A *package* is just grouping of modeling elements and it is not concerned with how these modeling elements will be used by the clients. To address such concern, a special version of package is provided in UML and it is called *subsystem*.

Graphically, a subsystem is a package stereotyped as `«subsystem»`. Semantically, a subsystem hides its modeling elements and reveals only its service to the clients. The clients must request subsystem's services through its provided interfaces.

Modeling access to subsystems via provided interfaces has important benefits for implementation models. If interfaces were not available for subsystems, other components would become dependent on the implementation of classes within subsystems. Such dependencies would create difficulties for the overall system's supportability (understandability, maintainability, and scalability)

Solutions to odd-numbered exercises

E1 - solution

Figure 3-48 (Book) is the modified object collaboration. The two threads are separately numbered as 1 and 2. The shipment thread has two dependent messages, numbered hierarchically as 1.1 and 1.2.

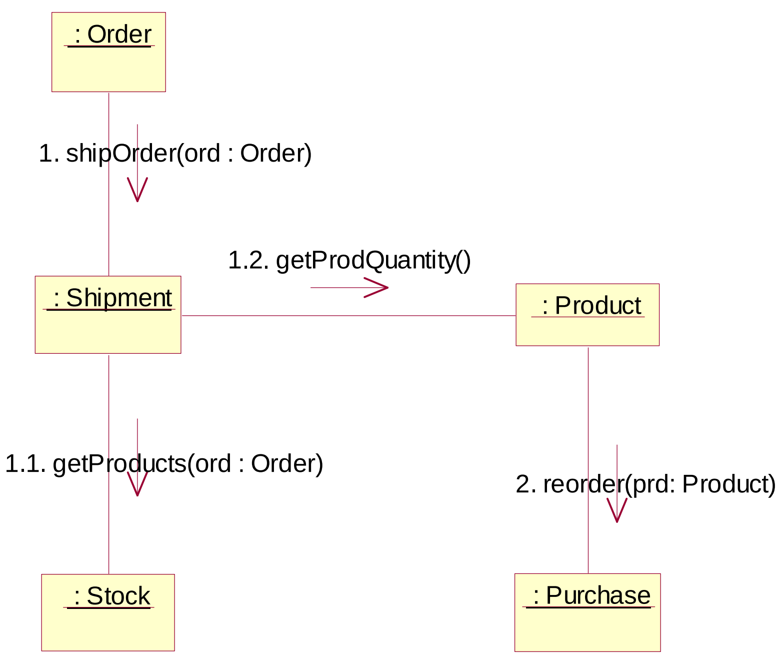


Figure 3-48 Object collaboration for shipment and replenishment

Apart from the lack of explanation how `Shipment` and `Purchase` objects are created, there are a few other details that remain unexplained. For example, the model does not explain how `getProdQuantity()` iterates over possibly many `Product` objects or what exactly makes `Product` to send the `reorder()` messages.

E3 - solution

In the modified class diagram (Figure 3-49), two new classes are identified (`Lecture` and `Tutorial`) as subset classes of `CourseOffering`. The subset classes have their own associations to `Teacher`.

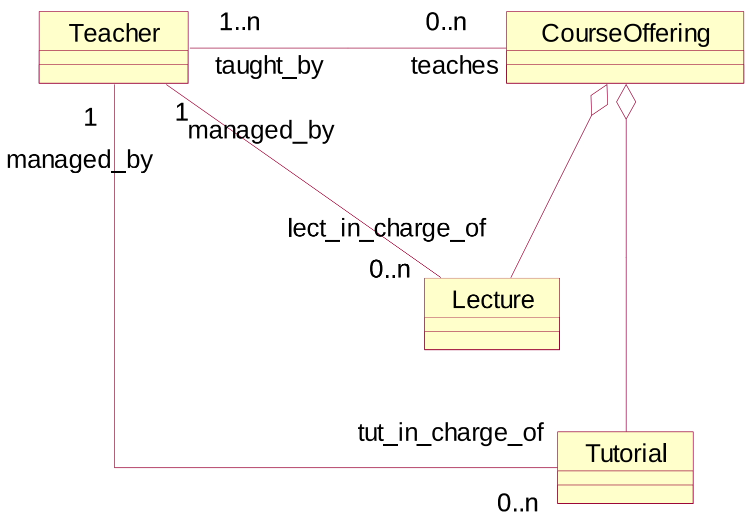


Figure 3-49 Modified class diagram to replace association class

Because a Teacher can be in charge of lectures and tutorials, the role names have different names: lect_in_charge_of and tut_in_charge_of. This is essential because two attributes will be needed in the implemented classes to capture these two roles (and the role names can then be used as attribute names).

E5 - solution

For explanatory reasons, the solution in Figure 3-50 shows return types, although they are not normally presented in collaboration diagrams. The messages iterateChapters and iterateParagraphs are just high levels descriptions for the requirement that Book has to iterate over many chapters and Chapter needs to iterate over many paragraphs. The details of such loop-based operations are not explained. Because iterateChapters and iterateParagraphs are just abstractions, rather than messages to methods, there are no brackets after their names that would signify a list of arguments.

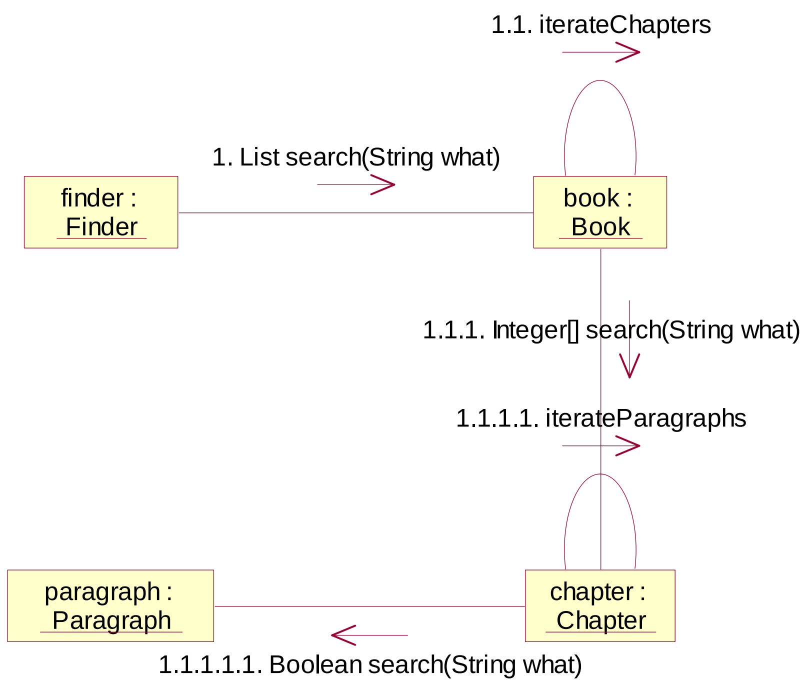

1.1. iterateChapters
1. List search(String what)
finder : Finder
book : Book
1.1.1. Integer[] search(String what)
1.1.1.1. iterateParagraphs
paragraph : Paragraph
chapter : Chapter
1.1.1.1.1. Boolean search(String what)

Figure 3-50 Object collaboration diagram
[With contribution of Bruc Lee Liong]

Finder invokes the search() operation on Book. It is eventually returned a List of chapter numbers and paragraph numbers containing the string value passed in the argument what. Before Book can return anything, it iterates over its Chapter objects to search for the string value.

However, the text of the book is contained in Paragraph objects. Accordingly, Chapter iterates over its Paragraph objects and it invokes the search() operation on each object. This operation returns just true/false, but for each true outcome, Chapter constructs an array of paragraph numbers. This array is returned to Book. Book can now build a list of chapter numbers and paragraph numbers, and return it to Finder.

E7 - solution

There are different (and arbitrary) solutions possible. Figure 3-51 is an example. Note that the example is not Java-based, as Java does not support multiple inheritance.

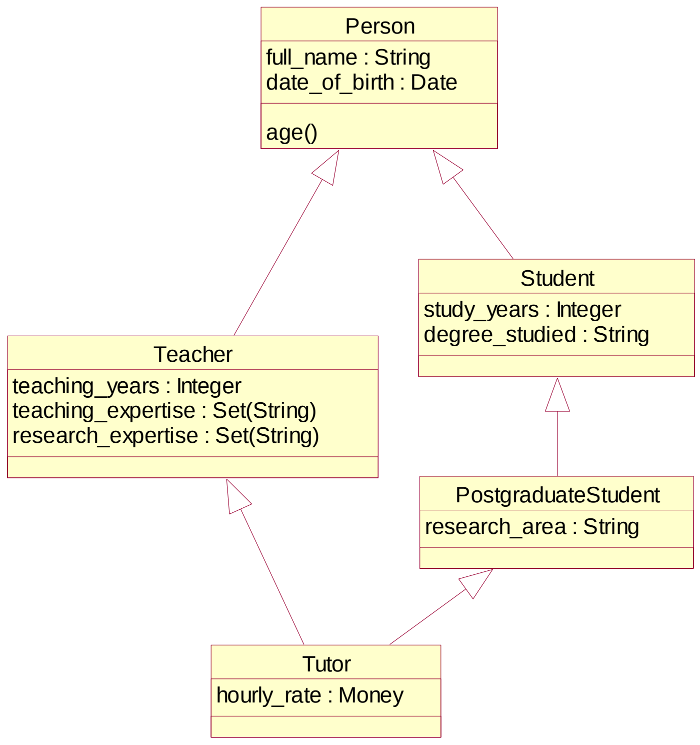


Figure 3-51 Multiple inheritance

The types of attributes `teaching_expertise` and `research_expertise` in `Teacher` are known as *parameterized types*. Both `Set` and `String` are classes. Hence, `String` is a parameter of `Set`. The values of these two attributes are sets of strings.

Solutions to odd-numbered exercises (VS)

F1 - solution

Figure 3-52 presents an extended use case diagram. `Rent Video` is extended by `Print Receipt`, `Check Age Eligibility`, and `Accept Payment`. There are no direct relationships made to the two `Scan...` use cases. The model assumes that `Rent Video` has the information from scanning when the rental transaction is started.

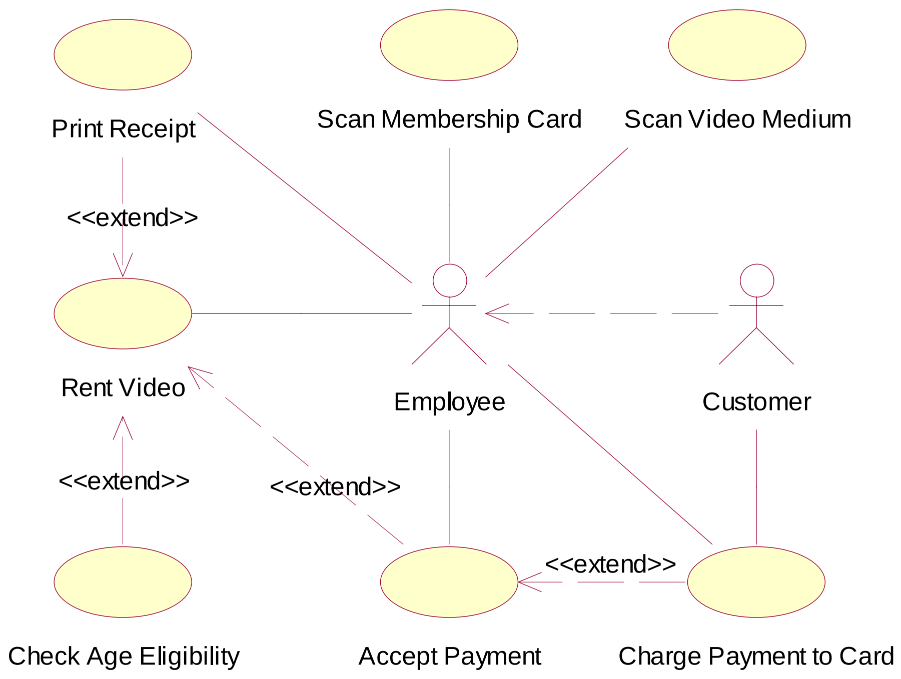


Figure 3-52 Extended use case diagram (VS)

F3 - solution

Figure 3-53 illustrates that the introduction of the `Sale` class leads to new associations, but the "tricky' issue is elsewhere. There is a need in the new model to introduce an association between `Customer` and `Payment`. A payment can be made for more than one rental and it is possible (as this is not forbidden) that these rentals can be related to different customers. Consequently, without a `Payment-Customer` association it may be not possible to identify a customer of the payment.

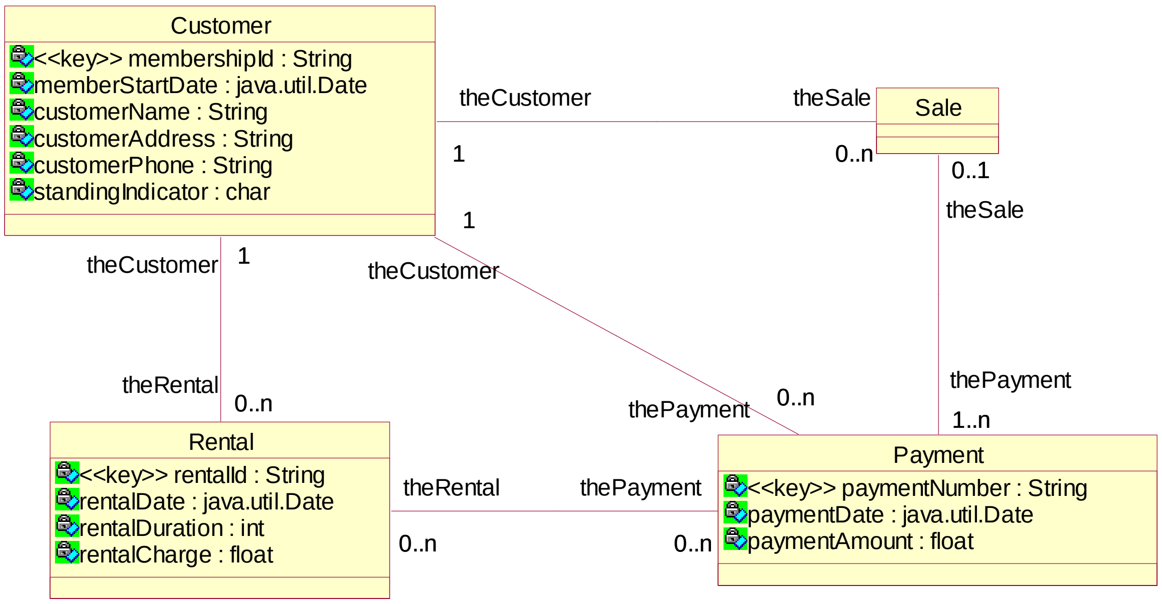


Figure 3-53 Modified and extended class model

F5 - solution

Figure 3-54 is a statechart model for the class `Video`. Note the use of event followed by a guard condition on transition to the state `Damaged`. Note also the use of a guard followed by an action on transition to state `For Sale`. All other transitions are marked with events.

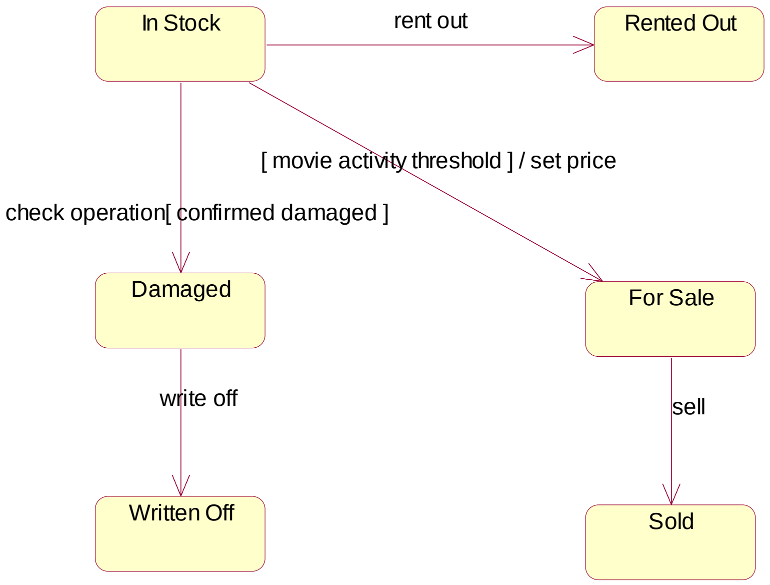

In Stock
rent out
Rented Out
[movie activity threshold] / set price
check operation[confirmed damaged]
Damaged
For Sale
write off
sell
Written Off
Sold

Figure 3-54 Statechart diagram for the class Video

The model does not identify a start or end state. It has been decided that such identifications are not essential for the modeling purpose.