Object-oriented meta-models distinguish between object identity and object equality. Two objects are *equal* if they comply to some equivalence rule. This can be as simple as 'store the same data' or more complicated and application-dependent, such as 'equality of complex numbers expressed in cartesian and polar coordinates'. They may still have a different identity. Objects addressed by two different object references are *identical* if they have the same object identity. Two identical objects are also equal but the reverse may not hold.

There are usually many different perspectives on the same object. Stakeholders consider different properties of an object as important. A player is interested in his or her schedule, while the soccer association is more interested in the player's license number. During object-oriented analysis, analysts try to define the overall functionality of an object; they are concerned with *what* an object can do. During object-oriented design, it is determined *how* an object works. Designers incorporate decisions about the environment in which the object resides. Yet, designers are not interested in how operations are implemented. They are merely interested in the interface the object has with other objects. Programmers implementing an object in an object-oriented programming language, in turn, focus on the way operations are implemented. Most of the object-oriented concepts that we discuss now apply to objects in all the different perspectives. This conceptual continuity is, in fact, one of the strongest advantages of the object-oriented approach to engineering distributed systems.

2.3.2 Types

Object Types

Object types specify the common characteristics of similar objects.

It would be inappropriate to declare operations and attributes individually for each object. Many objects are similar in structure and behaviour. They have the same attributes and operations. There may be many soccer player objects, for example, and all of them have the same attributes and operations; only their object identity and the values of their attributes differ. *Object types* specify those characteristics that are shared by similar objects.

Object types define a contract that binds the interaction between client and server objects.

[Meyer, 1988b] considers object types as *contracts* between clients and servers. This notion is particularly useful in distributed systems where client and server objects are often developed autonomously. The object type of a server object declares the attributes and operations that it offers. Automatic validation can determine whether or not client and server implementations obey the contract that is embedded in the type definition. The concept of object types therefore contributes greatly to reducing the development times of distributed objects.

Objects are instances of object types.

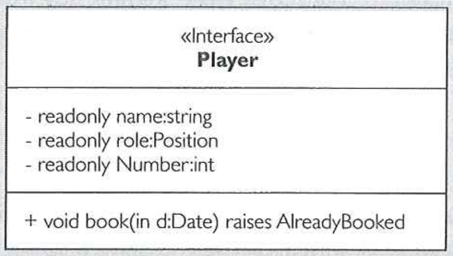
Objects are *instances* of types. By means of the instance relationship, we can always get information about the attributes and operations that an object supports. Object-oriented meta-models differ in whether each object has one or several types. They may also differ in whether the instance relationship can change during the lifetime of an object.

Object types are specified through interfaces that determine the operations that clients can request.

The type of a distributed object is specified through an *interface*. The interface defines all aspects of an object type that are visible to client objects. These include the operations that the object supports and the attributes of the object. The interface of an object type defines a set of *operations* by means of which the data stored in attributes of the object are modified.

Operations do not have to modify an object's state; they may just compute a function. The execution of an operation may rely on operations provided by other objects.

Example 2.9
An Object Type Specification Through an Interface



The above UML diagram shows the interface of `Player` objects. The diagram uses the stereotype `<<Interface>>` to indicate that the definition is an interface rather than a class. It defines the attributes and operations that are important for soccer players.

An operation has a *signature*, which determines the operation's name, the list of formal parameters and the result that the operation returns. Typed object-oriented meta-models will assign formal types to parameters and results. These formal types are used to check that the parameters that are passed to an operation during invocation actually correspond to the parameters that the operation expects. The same ideas apply to the return value of an operation. It may be typed in order to restrict the domain from which operations can return values.

The signature specifies the name, parameters, return types and exceptions of operations.

Operations also have a visibility. *Public operations* are visible to other objects; they may request their execution. *Private operations* are hidden from client objects. Execution of a private operation may only be requested by other operations of the same object.

Not all operations supported by an object type need to be included in an interface.

Non-Object Types

The management of distributed objects imposes a significant overhead. Unlike programming languages, which can treat object references as memory addresses for objects, object references in distributed systems also have to include location information, security information and the like. It would, therefore, be largely inappropriate to treat simple data, such as boolean values, characters and numbers as distributed objects. Likewise, it is advantageous if complex data types can be constructed from simpler data types without incurring the overhead imposed by objects. Yet, the idea of having a contract, which defines how these data can be used, applies as well. These considerations underlie the concept of *non-object types*, which can be atomic types or constructed types. Instances of these non-object types are *values* rather than objects. Values do not have an identity and cannot have references.

Meta-models for distributed objects also include types that are not objects.

Object-oriented meta-models will include a small number of *atomic types*. Although different types are supported by different meta-models, most will include types such as `boolean`, `char`, `int`, `float` and `string`. Atomic types determine a set of operations that can be applied to atomic values. Examples of these are the boolean operators `AND`, `OR` and `NOT`.

Constructed types are built from atomic types or object types by applying a *type constructor*.[1] The type constructors provided by object-oriented meta-models will again vary. They are likely to include records, arrays and sequences of simpler types. Again, a fixed set of operations may be applied to constructed types. Records may have a selection operation so that an individual element of the record can be selected. Likewise, sequences may have an iteration operation that visits each element of the sequence.

2.3.3 Requests

An object request is made by a client object in order to request execution of an operation from a server object.

Conceptually, an *object request* is a triple consisting of an object reference, the name of an operation and a list of actual parameters. An object request is similar to a method invocation in an object-oriented programming language. Both eventually execute an operation and pass actual parameters to the formal parameters of the operation. Requests, however, involve a more significant overhead: they have to resolve data heterogeneity; they need to synchronize client and server; they may cause network communication; and a server object may not be active and may have to be started before it can execute the request. As all this takes considerably longer, we want to distinguish between method invocations and object requests.

Like method invocations, requests may be parameterized. The request passes actual parameters from the client object to the server object. If client and server object reside on different hosts, parameters have to be transmitted via the network. If client and server are hosted by different hardware platforms, the data representation of request parameters may have to be translated. Request parameters might also be used to pass the results of the operation execution from the server to the client. Again, network communication and data representation translation might be required.

The number of parameters a client has to pass and their types are determined by the signature of the requested operation. For each formal parameter that is declared in the operation signature, the request has to include a compatible actual parameter. The engineering of distributed systems is considerably simplified if the compiler used for developing clients validates that the types of actual request parameters match the formal parameters declared in the signature of the requested operation declaration.

Example 2.10
Object Request

Let us reconsider the trainer who wants to book a training appointment with a soccer player. To make such a request, the trainer object has to have an object reference to the soccer player object. The trainer will then request a booking by providing the name of operation `book` as well as a value of type `Date` as an actual parameter. If the player and trainer objects are located on different machines, the request is transferred to the server host and the actual parameters are transmitted via the network to the server object. The server object, in turn executes the operation and transmits the result (whether the booking operation was successful) back to the client object.

[1] Type constructors should not be confused with constructors that create objects.

2.3.4 Exceptions

Exceptions notify clients about failures that occur during the execution of an object request.

Requests may not always be executed successfully. Requests may fail for reasons that we discussed in the previous chapter. When we discussed failure transparency in that chapter, we demanded that failures should be concealed from application programmers and users alike. While it is highly desirable to hide failures from application programmers, it might not always be possible. Failures, however, must be hidden from users. Client programmers, therefore, have to react to those failures that the middleware cannot handle itself. Thus, the meta-model should include a mechanism for notifying clients about the details of a fault. *Exceptions* are such a mechanism.

Exceptions are data structures for details about failures. An exception is *raised* by a server object or by the distribution middleware. Exceptions are transferred via the network from the server or middleware to the client. When a client checks for the occurrence of an exception we say that the client *catches* the exception. When an exception occurs, the client can use the exception's data structures in order to find out what went wrong.

We can distinguish *system* from *type-specific* exceptions. System exceptions are raised by the middleware. They inform about system failures, such as an unreachable server object. A server object may raise type-specific exceptions when the execution of a request would violate the object's integrity. The server object would then not complete the operation execution and inform the client about this failure by means of an exception. In theory, these situations could be avoided if the client always enquired whether the server would complete an operation. In practice, this is too slow because two requests are needed, one for the enquiry and another one for the request itself. Remember that requests involve a significant overhead and the number of requests has to be minimized. Hence, clients request an operation and are prepared for the failure of the operation.

Example 2.11
Exception Definition

Let us reconsider the operation to book a soccer player. That operation might raise an exception `AlreadyBooked` if the player has already been booked at the requested date and time. The data structures associated with `AlreadyBooked` might include vacant time slots that are adjacent to the ones requested. By using exceptions rather than an explicit enquiry about free time slots and a successive booking, the trainer object uses one rather than two requests.

The designer of a server object should be able to signal to designers of client objects that they should catch certain exceptions. Hence, meta-models often extend signatures for operations with an exception clause. That clause enumerates all the exceptions that the operation is allowed to raise.

Exceptions are part of the contract between client and server objects.

In fact, exceptions are part of the contract between server and client object. By using an operation whose signature includes exceptions, the client object is obliged to catch the declared exceptions. Unless it catches them, the client does not know if the operation has been executed successfully. As exceptions highlight these obligations, designers of server objects should use exceptions to indicate failures. Explicit error parameters or operation results do not convey this obligation to check and therefore the use of exceptions is preferred.