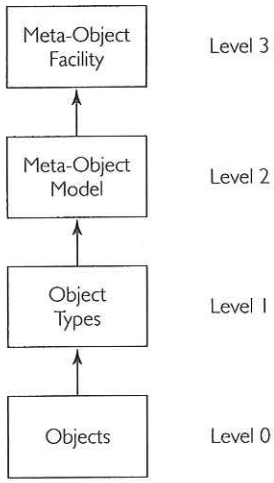
used whenever two concurrent threads proceed independently. Without parallel composition the independent states would have to be combined and this generally leads to an explosion in the number of states. Hence the aim of parallel composition is to reduce the number of states. A parallel composition is shown in Example 2.7.

2.3 A Meta-Model for Distributed Objects

In this section, we discuss the concepts that are needed for defining distributed objects. In order to avoid confusing the different levels of abstraction that are involved, we show the reference model of the OMG Meta-Object Facility [OMG, 1997a] in Figure 2.2. At the lowest level (Level 0), there are individual distributed objects. These are capable of performing services upon requests from other objects. They are instances of object types defined at Level 1. Object types specify the common properties of these similar objects.

Figure 2.2
Objects, Types, Meta-Object Models and Meta-Object Facilities



Object types are instances of a meta-object model (Level 2). This is the level where most of the concepts we define in this section are situated. We note that there are many meta-object models. Examples include the meta-model of the Unified Modeling Language or the object models that are used in object-oriented programming languages. They all differ slightly in the way they treat inheritance, polymorphism, static or dynamic typing and failure handling. Thus a need arises to be able to formalize meta-object models and a meta-object facility (Level 3) is used for that purpose. The reader should be aware of the existence of meta-object facilities, but we are not going to use such a formal meta-object facility because it does not aid in comprehending the concepts. In this section, we use UML to show how these concepts are applied. Thus we use concepts of Levels 0 and 1 to exemplify the meta-model concepts that we introduce in this section.

2.3.1 Objects

Objects have attributes that represent their states

An *object* has a set of attributes that together represent the *state* of the object. Attributes need not be accessible from other objects; they are then private or hidden. Hiding attributes

is encouraged because it achieves data abstraction as demanded by Parnas. Data stored in hidden attributes can only be accessed and modified by operations; other objects cannot become dependent on the internal data structures of an object and they can be changed without affecting other objects. This is particularly important if objects are designed and maintained in a distributed setting, possibly by different organizations. Objects may export a set of operations that reveal the state of attributes or modify their values. Other objects may request execution of an exported operation.

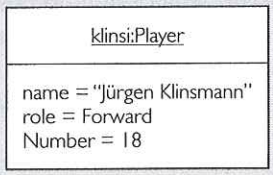
and operations that may modify the states.

An attribute may be seen as a pair of operations. The first operation is used to store data in the attribute and the second operation returns data from the attribute. Each attribute has a *name*. Names should be chosen to reveal the semantics of the data stored in the attribute. A name is used to identify an attribute within the context of an object. To be useful for this purpose, attribute names should be unique within an object. Attributes also have a type. The type determines the domain of the attribute. Attributes can only store and return values that are from the domain induced by the attribute's type.

An attribute is a mapping between an object and a value; it is characterized by a name and a type.

Object-oriented meta-models often provide concepts that support the hiding of attributes. A *public attribute* is visible outside the object and may be accessed and modified by other objects. This defeats data abstraction and public attributes should be used very carefully. A *private attribute* is not visible outside the object and can only be accessed in operations of the object. Some object-oriented meta-models also support *read-only attributes* whose visibility lies between public and private attributes; the value can be accessed from outside the object but only be modified by operations of the object. Attributes may also be *constant*. This means the value is determined during creation of the object and it is never changed.

Example 2.8
Soccer Player Object



A soccer player object may have certain attributes, such as a name, a role and a number worn on the back of a shirt. Some of these attributes may not be visible to the outside.

Some object-oriented programming languages support the concept of class variables, which are sometimes referred to as static member variables. These variables retain state information that is common to all instances of a type. We note that this concept is not appropriate for distributed objects, which are generally distributed across different hosts; class variables or static member variables would lead to the need to distribute updates of this state to all those hosts where instances reside.

Distributed objects do not have class or static variables.

Objects have a unique *object identifier* that is assigned to the object when it is created. Objects generally retain their identity throughout their lifetime. An *object reference* is a handle that a client has on an object. Clients use object references to reference the object, for instance when they want to request an operation. An object has just one identity but there may be many references to the object.

Objects have one unique identifier but may have multiple references.

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.

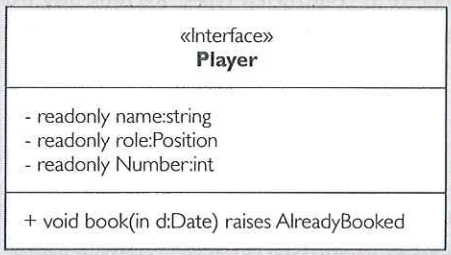
Objects are instances of object types.

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`.