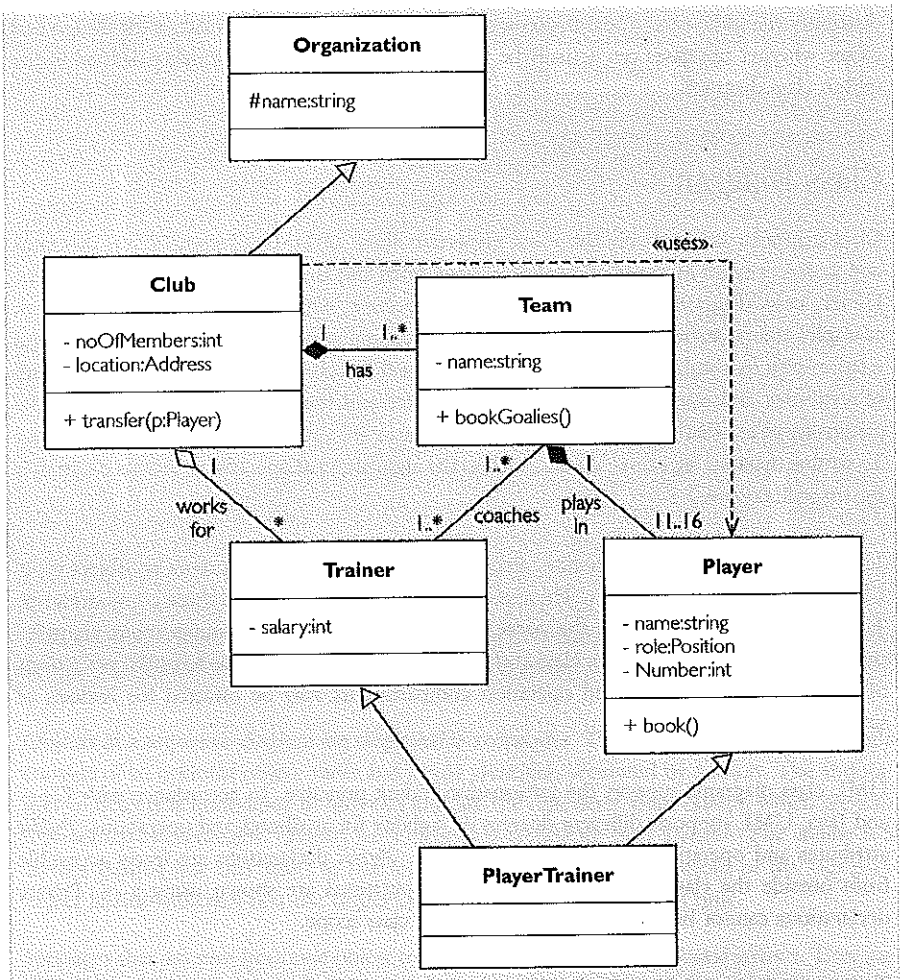
Example 2.3

Class Diagram with Object Types for Soccer Team Management



Class `Organization` generalizes class `Club`. The class diagram shows several associations, for instance to indicate that a `Player` `plays in` a `Team` and that a `Club` has several `Teams`. Association `works for` is an aggregation, `has` is a composition modelling that a `Team` is entirely controlled by a `Club`. A soccer team needs at least `11` and up to `16` players. This is modelled using the cardinality `11..16` at the `Player` end of association `plays in`. The cardinality of `1` at the `Team` end of `plays in` means that players play in exactly one `Team`. An example of a dependency is that `Club` depends on `Player` because `Player` is used as a parameter type in operation `transfer`.

Dependencies indicate that one object type depends on another type.

Dependencies are another form of association. A dependency indicates that a class depends on another class. This dependency is often created by the use of object types in attributes, operation results or parameters. Dependency associations are represented as dashed arrows. They start at the dependent class.

Associations may designate different *cardinalities*. The cardinality determines in how many instances of a particular association an object can participate. If upper bounds are not known, a * is used. A cardinality of zero means that participation in the association is optional.

Associations may have different cardinalities.

We note that class diagrams are used in several circumstances during the engineering of distributed objects. Class diagrams are used during the analysis stage when they determine the relevant properties of objects. In these class diagrams, the object types are modelled from the perspective of what the supported system should do. Class diagrams are also used during the design stage to define the interfaces of distributed objects. As distributed objects are often composed of many local objects, class diagrams may also be used for the design of server object implementations.

Class diagrams are used to capture type-level abstractions in both the analysis and design of distributed objects.

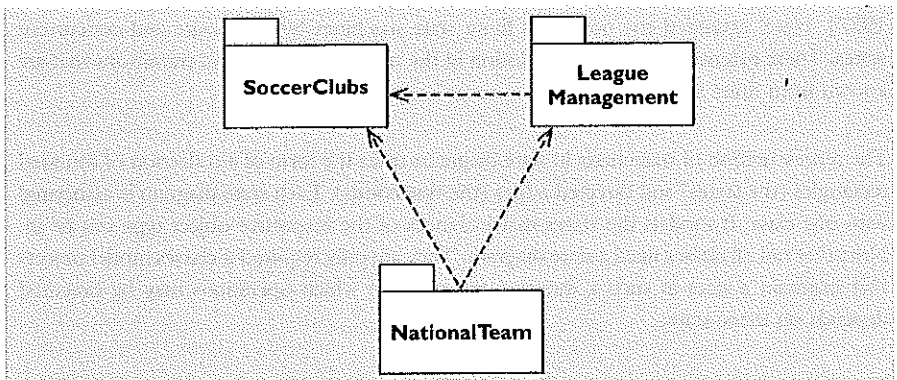
In practice, a class design of a distributed system may have to define several hundred object types. In order to manage the resulting complexity, some structuring mechanism is needed for class diagrams. *Packages* provide such a structuring mechanism. Packages can be used to organize class diagrams in a hierarchical way (see Example 2.4). A package is represented in a class diagram as a folder. Each package is represented by a class diagram at the level below.

Packages are used to structure analysis and design information in a hierarchical manner.

2.2.4 Object Diagrams

Object diagrams are used to show examples of instances of classes that were shown in class diagrams and how they are interconnected. They express the static part of an interaction diagram and we will occasionally use them for this purpose in this book. An object diagram shows objects, which are instances of classes. Objects are represented as rectangles. In order to highlight that these rectangles represent objects, rather than classes, we underline the content of the first compartment, which shows the name and the type of an object. These two are separated by a colon, but if it is not important, the name can be omitted.

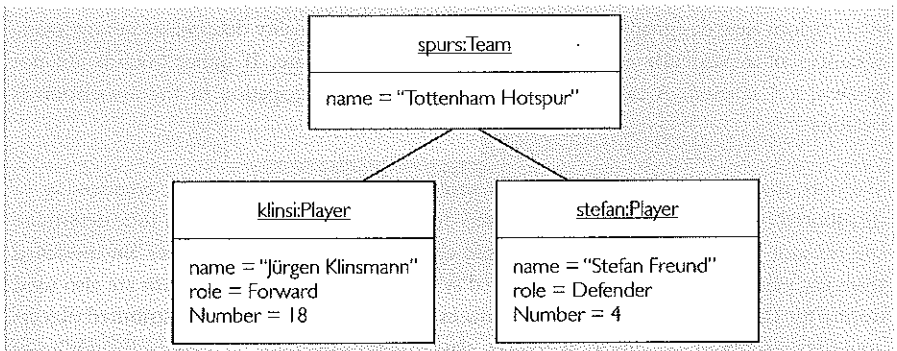
Object diagrams show examples of instances of classes.



Example 2.4
Use of Packages

Package `SoccerClubs` might be refined by the class diagram shown in Example 2.3. The packages have dependency associations between them. These dependency associations mean that a class in a dependent package uses one or more classes from the package on which it is dependent.

Example 2.5
Object Diagram



The above diagram shows a team and player objects that are part of a soccer club.

The second compartment in an object representation shows the attributes. Unlike in a class diagram, where the name, type and initialization are shown, the attribute representation in an object diagram only shows the name and the current value of the attributes. Because it is not necessary to specify the operations of objects (they are all specified in the classes of the object), the rectangles in object diagrams generally only have two compartments. Finally, object diagrams may show links that interconnect objects. A link is an instance of an association. As such, it needs to meet the constraints of the association, such as the cardinality.

2.2.5 State Diagrams

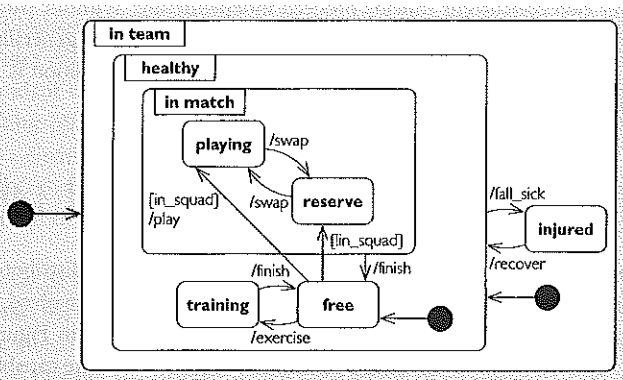
State diagrams model the dynamic behaviour of objects from a type level of abstraction.

Class diagrams model objects statically. UML *state diagrams* model the dynamic behaviour of objects. State diagrams provide support for describing the temporal evolutions of objects in response to interactions with other objects. This is achieved by modelling the various states of an object and identifying possible transitions between these states. State diagrams in UML are essentially equal in power to statecharts, which were first introduced by [Harel, 1987]. Harel extended the notion of finite state machines by composite and concurrent states. These extensions avoid the explosion in the number of states and transitions that are known from finite state machines.

The representation of statecharts in state diagrams takes the form of a collection of rounded rectangles (the states) and directed edges (the transitions). Each state diagram is associated with one class. It models the states that instances of the respective object type can be in. Note that not all classes need a state diagram. They are only required if classes model objects that maintain a state in such a way that the order in which operations may be executed depends on those states.

The transitions between states may be annotated. By convention, these annotations have two optional parts that are separated by a slash. The first part is a condition, which is given in square brackets. If present, the condition must hold for the transition to be enabled. The second part of a transition annotation denotes the operation that is executed during that transition.

Example 2.6
State Diagram with Composite States



The transition between `free` and `playing` is annotated. The condition `[in_squad]` is true if the trainer has designated the player to play when the game kicks off. State `in team` is a composite state. Transition `fall_sick` is allowed from every substate of `healthy`: a player can acquire an injury when he is playing in a game, during a training session or during spare time. He may even acquire a cold when sitting on the reserve bench. After having recovered from an injury, a player's state is the default `free`.

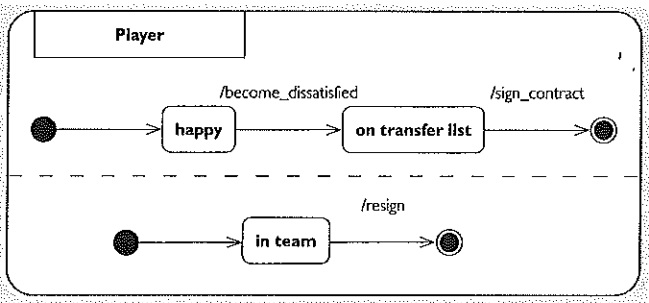
States may be composed of other states.

A *composite state* is represented as a rounded rectangle with embedded states. State diagrams do not impose any restrictions on the number of nested component states. Transitions leading from one component state to another mean that that transition is allowed from every substate of the composite state.

Default states determine the initial states of a composite state. Default states are designated by the unlabelled transition that originates at a filled circle.

State diagrams can be composed by nesting a state within another state. This essentially reduces the number of transitions as a transition leading from a state subsumes transitions from all substates that it entails. State diagrams also support *parallel states*. Parallel states are

Example 2.7
State Diagram with Parallel States



Two threads of computation proceed independently within a `Player` object. Initially, a player is happily playing in a team. The player may then at some time in point become dissatisfied and be put on the transfer list, while still playing in the team. When the player finds a new team, the player will sign a contract with the new team and resign from the current contract. Again, there is not necessarily any particular order in which these two actions have to be carried out.