2.2 UML Representations for Distributed Object Design

A software engineering method encompasses a notation as well as development heuristics and procedures that suggest how to use the notation. A notation is defined in terms of syntax and semantics. The syntactic part of the notation can be determined in terms of a grammar, be it for a textual or a graphical language. The semantics of a language can be distinguished in static and dynamic respects. The static semantics typically identify scope and typing rules, such as that declarations of identifiers must be unique within a certain scope and that applied occurrences must match particular declarations. The dynamic semantics defines the meaning for different concepts of the notation.

In this section, we discuss the syntax and semantics of the Unified Modeling Language (UML). UML is a notation for object-oriented analysis and design. The Object Management Group standardized UML in 1997. UML consists of a meta-model, the definition of the semantics of concepts identified in the meta-model, and a notation guide that identifies different diagram types that utilize the concepts of the meta-model. The purpose of this section is to introduce a standard notation for communication about objects. We do so by focussing on the diagram types and their use in the process of engineering distributed objects. For a discussion of the UML meta-model and the precise semantics of UML concepts, we refer the reader to the UML literature [Booch et al., 1999, Rumbaugh et al., 1999, Jacobson et al., 1999].

Different forms of UML diagram are used during the design of distributed objects.

The UML diagram types that we present now are likely to be used during the engineering of distributed objects. For each of them, we discuss its purpose, indicate the main concepts that are available and briefly and informally indicate the semantics of these concepts.

2.2.1 Use Case Diagrams

Use case diagrams capture requirements from a functional perspective.

Use case diagrams are produced during the requirements analysis stage of the system development. A use case diagram (see Example 2.1) has a graphical notation that is used to analyze the system and the different ways in which it is used.

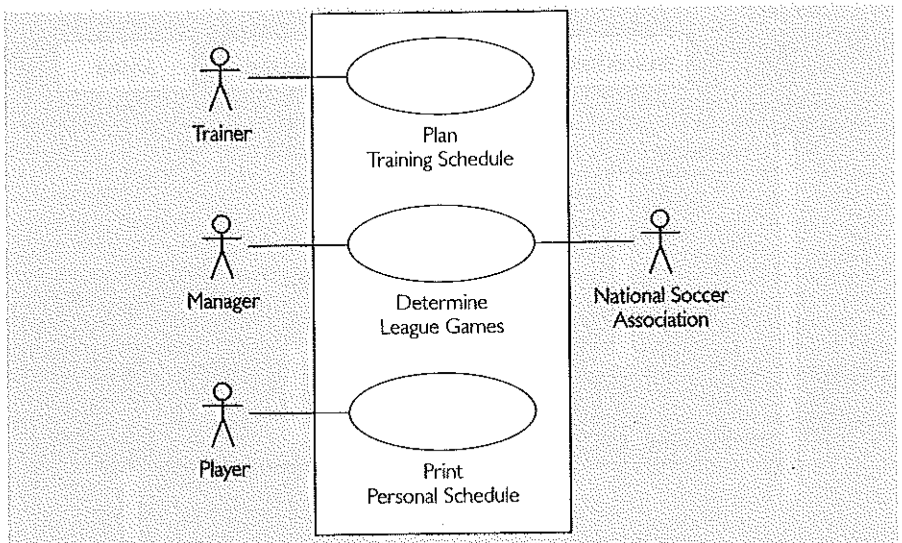
Actors are types of human user or external system.

Analysis begins with the identification of *actors* external to the system. An actor is a generic way of describing the potential users of the system. Actors are represented as stick persons in use case diagrams. In identifying actors, we will need to consider scenarios or situations typical to the system and its use. Note that the users are not necessarily human; they may also be external systems that use an application through its system interfaces. A further important distinction is between actors and users. Actors are types while users denote particular instances of these types.

A use case is a generic description of an entire course of events involving the system and actors external to the system.

A *use case* is depicted as an oval in a use case diagram. The description of the use case is associated with that diagram. Together the use cases in a diagram represent all the defined ways of using the system and the behaviour it exhibits whilst doing so. Again, we separate types and instances for use cases. Each use case is a specific type for how the system is used.

Example 2.1
Use Case Diagram for Soccer Management



The National Soccer Association is an external system actor. A particular system user, such as the trainer Ottmar, may take on the roles of different actors at different times, for instance Player or Trainer.

A *scenario* denotes an instance of a use case. When a user (an actor instance) inputs a stimulus, the scenario executes and starts a function belonging to the use case.

Actors interact with certain use cases. These *interactions* are recorded in the use case diagram in order to show which users of which types interact with a particular use case. Interactions also identify which types of user have a stake in the functionality that is embedded in a use case. Interactions are depicted as lines.

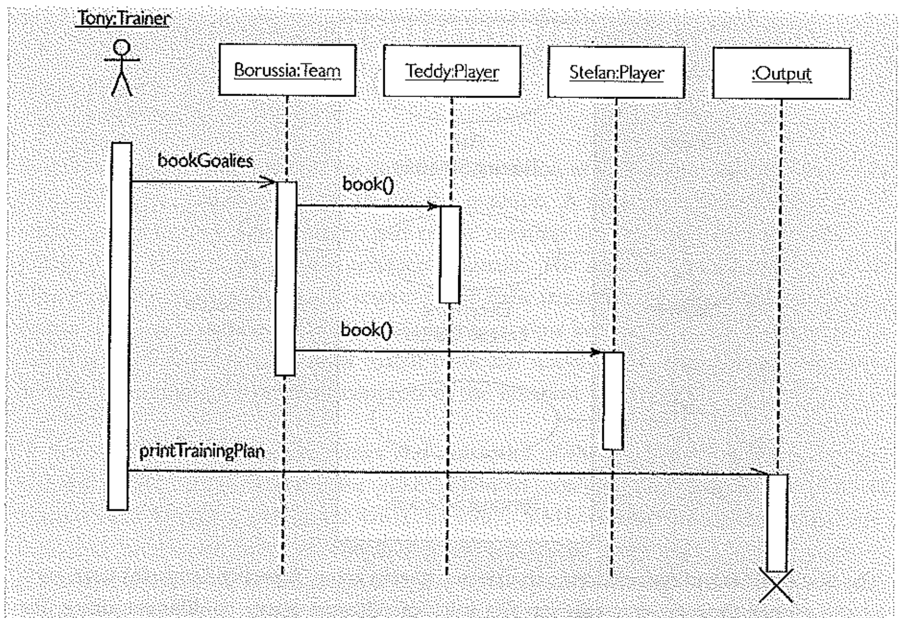
The identification of interaction between actors and use cases is an important step for the engineering of a distributed system. Actors will need an interface to the system. If actors represent humans they will need a user interface. If they represent external systems, they will need a system interface. In both cases, the interface may need distributed access to a use case. The user interface might be deployed on the web, for instance, and then the user should be able to interact with the system from a browser running on a remote machine. Likewise, the external system might be running on a remote host and should be able to access the use case remotely.

2.2.2 Sequence Diagrams

Sequence diagrams model interaction scenarios between objects in a sequential fashion.

Use cases are rather abstract and difficult to use for eliciting detailed requirements. A scenario can be seen as particular example of a use case and as such it is more concrete. *Sequence diagrams* are a graphical notation that can be used for modelling scenarios. Example 2.2 shows a sequence diagram that refines a scenario for the Plan Training Schedule use case of Example 2.1. A sequence diagram depicts objects that participate in the scenario and the sequence of messages by means of which the objects communicate.

Example 2.2
Sequence Diagram for a Plan Training Schedule Scenario



This diagram refines a scenario for the Plan Training Schedule use case of Example 2.1. It shows four objects: `Borussia`, `Teddy` and `Stefam` and an anonymous `Output` object.

Objects in a sequence diagram are displayed as rectangles. The annotation in that rectangle consists of an optional object name and a type identifier, separated by a colon. The annotation of objects is underlined in order to indicate the difference from classes, which are also represented as rectangles. Objects have lifelines that indicate their lifetime. The lifeline is a vertical dashed line that starts at the rectangle. The cross at the bottom of the `Output` object lifeline indicates that the object ceases to exist at that point in time.

Messages are sent from a client object to a server object to request operation executions.

Messages that are sent from one object to another are shown as horizontal arrows. They are given names that usually correspond to operations of the target object. An arrow with a complete arrowhead shows a synchronous message. In the example, `bookGoalies` is a synchronous message that the trainer of the goalkeepers would send to the team object. `printTrainingPlan` is an asynchronous message because the trainer wants to continue working while the output is processed on a printer. Filled arrowheads indicate local messages that are implemented as procedure calls. The `book` messages that the team object sends to the two players who are goalkeepers are examples of those. Messages are ordered in time. Earlier messages are shown above later messages. A rectangular stripe along a lifeline represents the activation of the object. Activation corresponds to a period when the object performs an action, either directly or by sending a message to another object. The top of the stripe indicates the beginning of activation and the bottom shows when the object is deactivated.

Sequence diagrams can be used during analysis and design.

The use of sequence diagrams is not confined to requirements analysis. Sequence diagrams are also extremely useful to show design scenarios. During analysis, sequence diagrams show what happens in a scenario. In design, sequence diagrams show how a scenario is implemented using objects that can be directly mapped to the target programming language and distribution middleware. These implementation scenarios are used to validate the design and to see whether use cases can actually be implemented with the functionality defined in the design class diagram.

2.2.3 Class Diagrams

Class diagrams are probably the most important form of UML diagram. Class diagrams define object types in terms of attributes, operations and the associations between objects. Hence, class diagrams provide a type-level perspective, while sequence diagrams present examples at an instance-level of abstraction. Class diagrams model attributes, operations and relationships in a static way. The concerns addressed by class diagrams include the visibility of attributes and operations, the type of attributes and the signatures of operations, the cardinality of associations and the like. Class diagrams are a static modelling tool and do not determine dynamic aspects of classes, such as the algorithms of operations. A class diagram is shown in Example 2.3.

Class diagrams define the type of objects in terms of attributes, operations and associations.

The main building blocks of a class diagram are *classes*. A class represents an object type. Classes are shown as rectangles. These rectangles may be divided into three compartments. The top compartment shows the name of the class, the middle compartment identifies the attributes supported by the class and the bottom compartment declares the operations of the class. The attribute and operation compartments may be omitted for reasons of simplicity.

UML class diagrams support the information hiding principle. In accordance with C++ and Java, UML supports different degrees of visibility for attributes and operations. *Public* attributes and operations are displayed with a +. *Public* declarations are freely accessible from outside the class. *Private* attributes and operations are shown using a −. Private declarations cannot be accessed from outside the class scope.

The visibility of attributes can be public or private.

Type and subtype relationships between classes are modelled in UML class diagrams using *generalizations*. They are represented as an arrow with a white arrowhead. The class to which the arrowhead points generalizes the class at the other end of the arrow. The existence of generalizations motivates a third category of visibility. *Protected* operations and attributes are only accessible from within the class and any specialized class. Protected declarations are preceded by a #. In the example, this means that operations inside the scope of class `Club` can access the inherited `name` attribute, as its visibility is declared protected in class `Organization`.

A type may generalize another type.

Associations are shown using lines and they may be given a name. Associations model the fact that one class is related to another class. Most associations are *binary*, which means that they connect two classes. Associations are the most general form of relationship. An *aggregation* is an association that indicates that objects are aggregates of other objects. Aggregation relationships have a diamond at the object that forms the complex object from its aggregates. *Composition* associations are stronger form of aggregations and they determine that the component fully belongs to the composite. This means that there can be only one composite for every component. A filled diamond indicates such a composition.

Associations connect objects and they may have aggregation or composition semantics.