Chapter 1

Object-Oriented Design

Abstract Designing large programs is a challenging task. A key element of success is the encapsulation and definition of semantic meaningful units and interfaces. The object-oriented programming paradigm is supported by well-developed design paradigms. Here, we will examine both the *Universal Modelling Language* (*UML*), which is a graphical language for structuring object-oriented programs, and the *nouns and verbs* method for analyzing problem descriptions for candidates for classes as their interactions. In this chapter, you will learn how to

- use UML to visualize classes, their hierarchies, and their objects
- use the nouns and verbs method to analyze a problem statement or a use-story to identify potential candidates for classes and their interactions

Object-oriented programming has a well-developed methodology for analysis and design. The analysis serves as input to the design phase, where the analysis reveals *what* a program is supposed to do, and the design *how* it is supposed to be doing it. The analysis should be expressed in general terms irrespective of the technologic constraints, while the design should include technological constraints such as defined by the targeted language and hardware.

The primary steps for *object-oriented analysis and design* are:

1. identify objects,
2. describe object behavior,
3. describe object interactions,
4. describe some details of the object's inner workings,
5. write a precise description for classes, properties and methods using, e.g., F#'s XML documentation standard,
6. write mockup code,
7. write unit tests and test the basic framework using the mockup code,
8. replace the mockup with real code while testing to keep track of your progress. Extend the unit test as needed,
9. evaluate code in relation to the desired goal,
10. complete your documentation both in-code and outside.

Steps 1–4 are the analysis phase which gradually stops in step 4, while the design phase gradually starts at step 4 and gradually stops when actual code is written in step 7. Programming is never a linear experience, and you will often need to go back to previous steps to update or change decisions. You should not refrain from improving your program design and implementation, but you should always be mindful of the goal. Often less than the perfect solution will suffice.

An object-oriented analysis can be a daunting process. A good starting point is a *use case*, *problem statement*, or a *user story*, which in human language describes a number of possibly hypothetical interactions between a user and a system with the purpose of solving some task. Two useful methodologies for performing an object-oriented analysis is the method of nouns-and-verbs and the unified modeling language, described in the following sections.

1.1 Identification of Objects, Behaviors, and Interactions by Nouns-and-Verbs

Analyzing a specific use-case, good candidates for objects are persons, places, things, events, concept etc., which are almost always characterized by being *nouns* in the text. Interactions between objects are actions that bind objects together, and actions are often associated with *verbs*. When choosing methods, it is important to maintain an object-centered perspective, i.e., for a general-purpose object, we should limit the need for including information about other objects. E.g., a value of type `int` need not know anything about the program in which it is being used.

Said briefly, the *nouns-and-verbs method* is:

> Nouns are object candidates, and verbs are candidate methods that describe interactions between objects.

1.2 Class Diagrams in the Unified Modelling Language

Having found an initial list of candidate objects and interactions, it is often useful to make a drawing of these relations with an increased focus on the object's inner workings. A *class diagram* is a schematic drawing of the program, highlighting its object-oriented structure, and we will use the *Unified Modelling Language 2* (*UML*) [**?**] standard. The standard is very broad, and here we will discuss structure diagrams for use in describing objects.

A class is drawn as shown in Figure 1.1. In UML, classes are represented as boxes

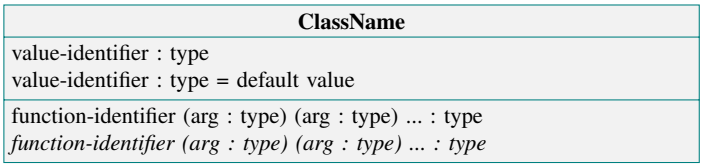

Fig. 1.1 A UML diagram for a class consists of it's name, zero or more attributes, and zero or more methods.

with their class name. Depending on the desired level of details, zero or more properties and methods are described. These describe the basic interface to the class and objects of its type. Abstract members that require an implementation are shown in cursive. Here we have used F# syntax to conform with this book theme, but typically C# syntax is used. Interfaces are a special type of class that require an implementation. To highlight this, UML uses the notation shown in Figure 1.2.

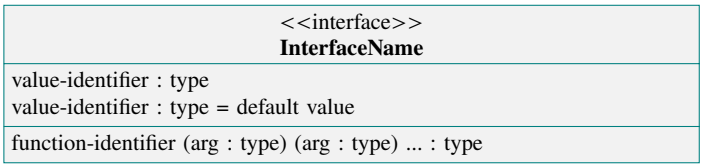

Fig. 1.2 An interface is a class that requires an implementation.

Relations between classes and objects are indicated by lines and arrows. The most common ones are summarized in Figure 1.3. Their meaning will be described in

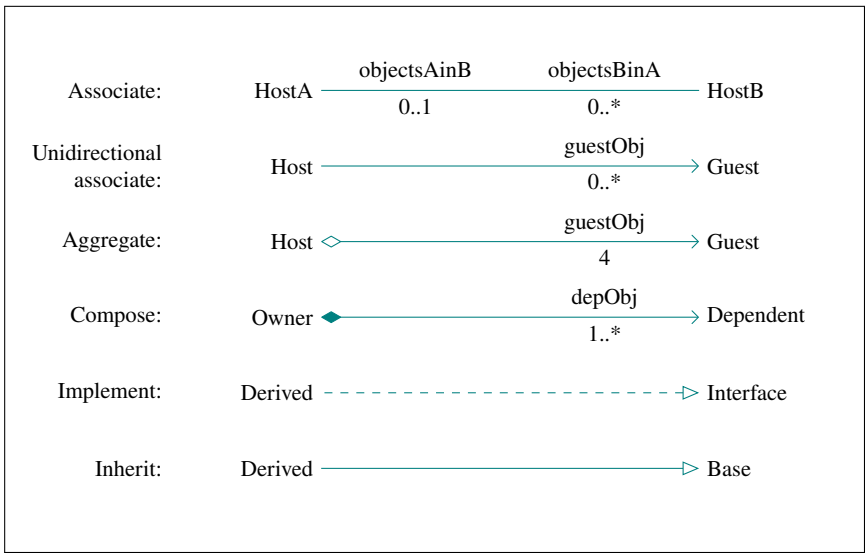

Fig. 1.3 Arrows used in class diagrams to show relations between objects.

detail in the following.

1.2.1 Associations

A family of relations is *association*, *aggregation*, and *composition*, and these are distinguished by how they handle the objects they are in relation with. The relation between the three relations is shown in Figure 1.4. Aggregational and compositional are specialized types of associations that imply ownership and are often called *has-a* relations. A composition is a collection of parts that makes up a whole. In object-oriented design, a compositional relation is a strong relation, where a guest object makes little sense without the host, as a room cannot exist without a house. An aggregation is a collection of assorted items, and in object-oriented design, an aggregational relation is a loose relation, like how a battery can meaningfully

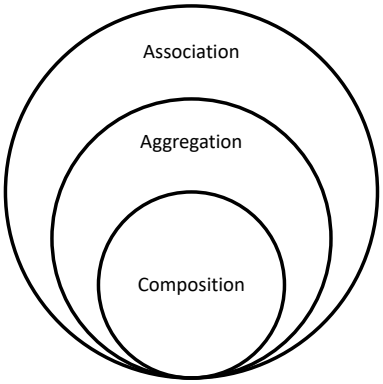

Fig. 1.4 The relation between Association, Aggregation and Composition in UML.

be separated from a torchlight. Some associations are neither aggregational nor compositional, and commonly just called an association. An association is a group of people or things linked for some common purpose a cooccurrence. In object-oriented design, associations between objects are the loosest possible relations, like how a student may be associated with the local coffee shop. Sometimes associational relations are called a *knows-about*.

The most general type of association, which is just called an association, is the possibility for objects to send messages to each other. This implies that one class knows about the other, e.g., uses it as arguments of a function or similar. A host is associated with a guest if the host has a reference to the guest. Objects are reference types, and therefore, any object which is not created by the host, but where a name is bound to a guest object but not explicitly copied, then this is an association relation.

Bidirectional association means that classes know about each other. The UML notation is shown in Figure 1.5. Association may be annotated by an identifier and

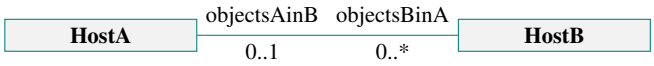

Fig. 1.5 Bidirectional association is shown as a line with optional annotation.

a multiplicity. In the figure, HostA has 0 or more variables of type HostB named objectsBinA, while HostB has 0 or 1 variables of HostA named objectsAinB. The multiplicity notation is very similar to F#'s slicing notation. Typical values are shown in Table 1.1. If the association is unidirectional, then an arrow is added for emphasis,

n	exactly n instances
*	zero or more instances
n..m	n to m instances
n..*	from n to infinite instances

Table 1.1 Notation for association multiplicities is similar to F#'s slicing notation.

as shown in Figure 1.6. In this example, Host knows about Guest and has one instance of it, and Guest is oblivious about Host.

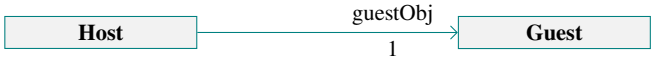

Fig. 1.6 Unidirectional association shows a one-side *has-a* relation.

A programming example showing a unidirectional association is given in Listing 1.1. Here, the `student` is unidirectionally associated with a `teacher` since the `student`

Listing 1.1 umlAssociation.fsx:
The `student` is associated with a `teacher`.

```
type teacher () =
  member this.answer (q : string) = "4"
type student (t : teacher) =
  member this.ask () = t.answer("What is 2+2?")

let t = teacher ()
let s = student (t)
s.ask()
```

can send and receive messages to and from the `teacher`. The `teacher`, on the other hand, does not know anything about the `student`. In UML this is depicted as shown in Figure 1.7.

Fig. 1.7 The `teacher` and `student` objects can access each other's functions, and thus they have an association relation.

Aggregated relationships are a specialization of associations. As an example, an author may have written a book, but once created, the book gets a life independent of the author and may, for example, be given to a reader, and the book continues to exist even when the author dies. That is, In aggregated relations, the host object has a reference to a guest object and may have created the guest, but the guest will be shared with other objects, and when the host is deleted, the guest is not.

Aggregation is illustrated using a diamond tail and an open arrow, as shown in Figure 1.8. Here the Host class has stored aliases to four different Guest objects.

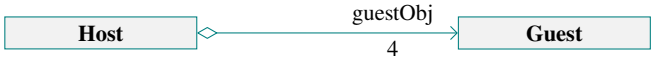

Fig. 1.8 Aggregation relations are a subset of associations where local aliases are stored for later use.

An programming example of an aggregation relation is given in Listing 1.2. In aggregated relations, there is a sense of ownership, and in the example, the `author` object creates a `book` object which is published and bought by a reader. Hence the `book` change ownership during the execution of the program. In UML this is to be depicted as shown in Figure 1.9.

Listing 1.2 umlAggregation.fsx:
The book has an aggregated relation to `author` and `reader`.

```
type book (name : string) =
  let mutable _name = name
type author () =
  let _book = book("Learning to program")
  member this.publish() = _book
type reader () =
  let mutable _book : book option = None
  member this.buy (b : book) = _book <- Some b

let a = author ()
let r = reader ()
let b = a.publish ()
r.buy (b)
```

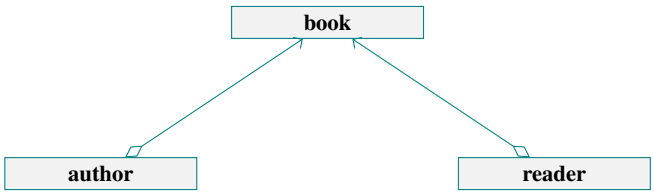

Fig. 1.9 A book is an object that can be owned by both an author and a reader.

A compositional relationship is a specialization of aggregations. As an example, a dog has legs, and dog legs can not very sensibly be given to other animals. That is, in compositional relations, the host creates the guest, and when the host is deleted, so is the guest. A composition is a stronger relation than aggregation and is illustated using a filled filled diamond tail, as illustrated in Figure 1.10. In this example, Owner

Fig. 1.10 Composition relations are a subset of aggregation where the host controls the lifetime of the guest objects.

has created 1 or more objects of type Dependent, and when Owner is deleted, so are these objects.

A programming example of a composition relation is given in Listing 1.3. In

Listing 1.3 umlComposition.fsx:
The dog object is a composition of four `leg` objects.

```
type leg () =
  member this.move = "moving"
type dog () =
  let _leg = List.init 4 (fun e -> leg ())

let bestFriend = dog ()
```

Listing 1.3, a `dog` object creates four `leg` objects, and it makes less sense to be able to turn over the ownership of each `leg` to other objects. Thus, a `dog` is a composition of `leg` objects. Using UML, this should be depicted as shown in Figure 1.11.

Fig. 1.11 A dog is a composition of legs.

1.2.2 Inheritance-type relations

Classes may inherit other classes where the parent is called the base class and the children its derived classes. Such a relation is often called an *is-a* relation, since the derived class *is a* kind of base class.

Inheritance is a relation between properties of classes. As an example, a student and a teacher is a type of person. All persons have names, while a student also has a reading list, and a teacher also has a a set of slides. Thus, both students and teacher may inherit from a person to gain the common property, name. In UML this is illustated with an non-filled, closed arrow as shown in Figure 1.12. Here two classes

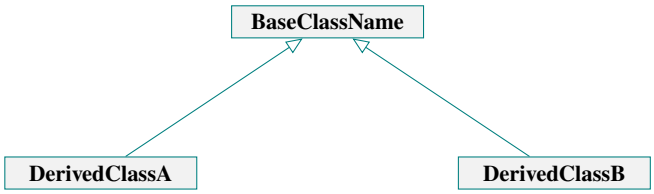

Fig. 1.12 Inheritance is shown by a closed arrowhead pointing to the base.

inherit the base class.

A programming example of an inheritance is given in Listing 1.4. In Listing 1.4, the `student` and the `teacher` classes are derived from the same `person` class. Thus, they all three have the `name` property. Using UML, this should be depicted as shown in Figure 1.13.

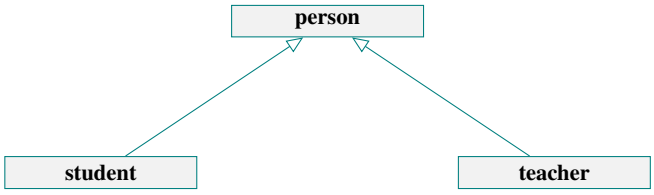

Fig. 1.13 A student and a teacher inherit from a person class.

Listing 1.4 umlInheritance.fsx:
The `student` and the `teacher` class inherits from the `person` class.

```
type person (name : string) =
  member this.name = name
type student (name : string, book : string) =
  inherit person(name)
  member this.book = book
type teacher (name : string, slides : string) =
  inherit person(name)
  member this.slides = slides

let s = student("Hans", "Learning to Program")
let t = teacher("Jon", "Slides of the day")
```

An interface is a relation between the properties of an abstract class and a regular class. As an example, a television and a car both have buttons, that you can press, although their effect will be quite different. Thus, a television and a car may both implement the same interface. In UML, interfaces are shown similarly to inheritance, but using a stippled line, as shown in Figure 1.14.

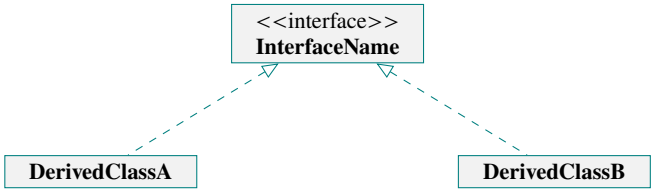

Fig. 1.14 Implementations of interfaces is shown with stippled line and closed arrowhead pointing to the base.

A programming example of an interface is given in Listing 1.5. In Listing 1.5, the

Listing 1.5 umlInterface.fsx:
The `television` and the `car` class both implement the `button` interface.

```
type button =
  abstract member press : unit -> string
type television () =
  interface button with
    member this.press () = "Changing channel"
type car () =
  interface button with
    member this.press () = "Activating wipers"
let pressIt (elm : #button) =
  elm.press()

let t = television()
let c = car()
printfn "%s" (pressIt t)
printfn "%s" (pressIt c)
```

`television` and the `car` classes implement the `button` interface. Hence, although they are different classes, they both have the `press ()` method and, e.g., can be given as a function requiring only the existence of the `press ()` method. Using UML, this should be depicted as shown in Figure 1.15.

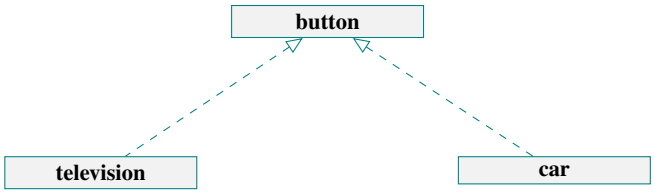

Fig. 1.15 A student and a teacher inherit from a person class.

1.2.3 Packages

For visual flair, modules and namespaces are often visualized as *packages*, as shown in Figure 1.16. A package is like a module in F#.

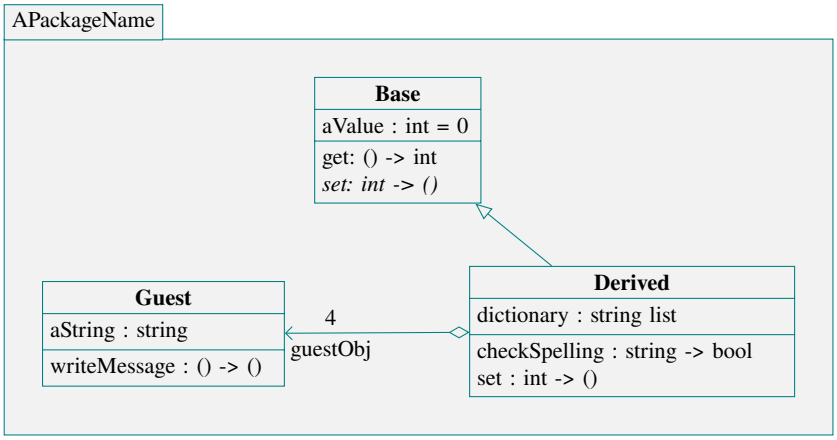

Fig. 1.16 Packages are a visualizations of modules and namespaces.

1.3 Programming Intermezzo: Designing a Racing Game

An example is the following *problem statement*:

Problem 1.1

rite a racing game, where each player controls his or her vehicle on a track. Each vehicle must have individual features such as top acceleration, speed, and handling. The player must be able to turn the vehicle left and right, and to accelerate up and down. At the beginning of the game, each vehicle is placed behind the starting line. Once the start signal is given, then the players may start to operate their vehicles. The player who first completes 3 rounds wins.

To seek a solution, we will use the **nouns**-and-verbs method. Below, the problem statement is repeated with **nouns** and verbs highlighted.

> Write a **racing game**, where each **player** controls his or her **vehicle** on a **track**. Each **vehicle** must have individual **features** such as **top acceleration**, **speed**, and **handling**. The **player** must be able to turn the vehicle left and right, and to accelerate up and down. At the **beginning** of the **game**, each **vehicle** is placed behind the **starting line**. Once the **start signal** is given, then the **players** may start to operate their **vehicles**. The **player** who first completes 3 **rounds** wins.

The above nouns and verbs are candidates for objects, their behaviour, and their interaction. A deeper analysis is:

Identification of objects by nouns (Step 1):

Identified unique nouns are: **racing game (game)**, **player**, **vehicle**, **track**, **feature**, **top acceleration**, **speed**, **handling**, **beginning**, **starting line**, **start signal**, **rounds**. From this list we seek cohesive units that are independent and reusable. The nouns

> **game**, **player**, **vehicle**, and **track**

seem to fulfill these requirements, while all the rest seems to be features of the former and thus not independent concepts. E.g., **top acceleration** is a feature of a **vehicle**, and **starting line** is a feature of a **track**.

Object behavior and interactions by verbs (Steps 2 and 3):

To continue our object-oriented analysis, we will consider the object candidates identified above, and verbalize how they would act as models of general concepts useful in our game.

player The **player** is associated with the following verbs:

- A **player** controls/operates a **vehicle**.
- A **player** turns and accelerates a **vehicle**.
- A **player** completes **rounds**.

- A **player** wins.

Verbalizing a **player**, we say that a **player** in general must be able to control the **vehicle**. In order to do this, the **player** must receive information about the **track** and all **vehicles**, or at least some information about the nearby **vehicles** and **track**. Furthermore, the **player** must receive information about the state of the **game**, i.e., when the race starts and stops.

vehicle A **vehicle** is controlled by a **player** and further associated with the following verbs:

- A **vehicle** has **features top acceleration**, **speed**, and **handling**.
- A **vehicle** is placed on the **track**.

To further describe a **vehicle**, we say that a **vehicle** is a model of a physical object which moves around on the **track** under the influence of a **player**. A **vehicle** must have a number of attributes such as top acceleration, speed, and handling, and must be able to receive information about when to turn and accelerate. A **vehicle** must be able to determine its location in particular if it is on or off **track** and, and it must be able to determine if it has crashed into an obstacle such as another **vehicle**.

track A **track** is the place where vehicles operate and is further associated with the following verbs:

- A **track** has a **starting line**.
- A **track** has rounds.

Thus, a **track** is a fixed entity on which the **vehicles** race. It has a size and a shape, a starting and a finishing line, which may be the same, and **vehicles** may be placed on the **track** and can move on and possibly off the **track**.

game Finally, a **game** is associated with the following verbs:

- A **game** has a **beginning** and a **start signal**.
- A **game** can be **won**.

A **game** is the total sum of all the **players**, the **vehicles**, the **tracks**, and their interactions. A **game** controls events, including inviting **players** to race, sending the **start signal**, and monitoring when a **game** is finished and who **won**.

From the above we see that the object candidates **features** seems to be a natural part of the description of the **vehicle**'s attributes, and similarly, a **starting**

line may be an intricate part of a **track**. Also, many of the *verbs* used in the problem statement and in our extended verbalization of the general concepts indicate methods that are used to interact with the object. The object-centered perspective tells us that for a general-purpose **vehicle** object, we need not include information about the **player**, analogous to how a value of type `int` need not know anything about the program, in which it is being used. In contrast, the candidate **game** is not as easily dismissed and could be used as a class which contains all the above.

With this description, we see that 'start signal' can be included as a natural part of the game object. Being confident in our working hypothesis of the essential objects for the solution, we continue our investigation into further details about the objects and their interactions.

Analysis details (Step 4):

A class diagram of our design for the proposed classes and their relations is shown in Figure 1.17.

In the present description, there will be a single Game object that initializes the other objects, executes a loop updating the clock, queries the players for actions, and informs the vehicles that they should move and under what circumstances. The track has been chosen to be dumb and does not participate much in the action. Player's method getAction will be an input from a user by keyboard, joystick or similar, but the complexity of the code for a computer player will be large, since it needs to take a sensible decision based on the track and the location of the other vehicles. What at present is less clear, is whether it is the responsibility of Game or Vehicle to detect an off track or a crash event. If a vehicle is to do this, then each vehicle must have aggregated association to all other vehicles and obstacles. So, on the one hand, it would seem an elegant delegation of responsibilities that a vehicle knows whether it has crashed into an obstacle or not, but on the other hand, it seems wasteful of memory resources to have duplicated references of all obstacles in every vehicle. The final choice is thus one of elegance versus resource management, and in the above, we have favored resource management. Thus, the main loop in Game must check all vehicles for a crash event after the vehicle's positions have been updated, and in case of a crash, informs the relevant vehicles.

Having created a design for a racing game, we are now ready to start coding (Step 6–). It is not uncommon that transforming our design into code will reveal new structures and problems that possibly require our design to be updated. Nevertheless, a good design phase is almost always a sure course to avoid many problems once coding, since the design phase allows the programmer to think about the problem from a helicopter perspective before tackling details of specific sub-problems.

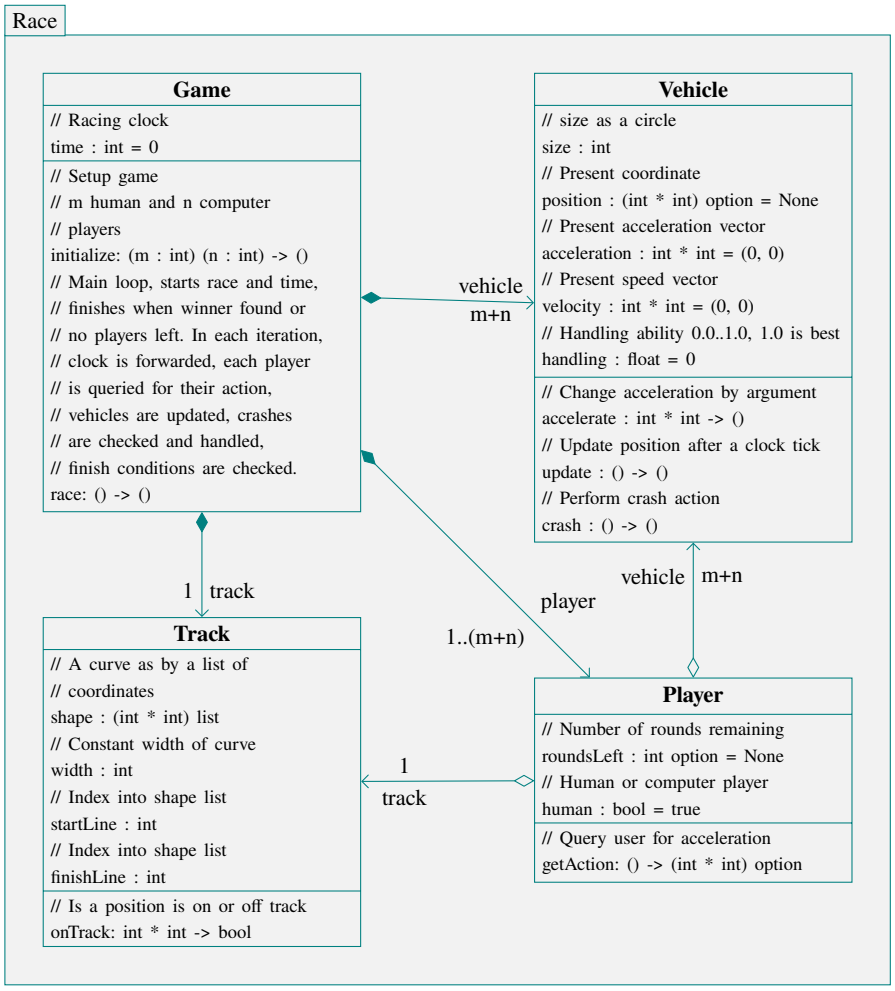

Fig. 1.17 A class diagram for a racing game.

1.4 Key Concepts and Terms in This Chapter

In this chapter, we have looked at object-oriented design. Two key elements have been discussed:

- how to organize and reason about classes and objects and their interactions using **UML** diagrams
- how in UML classes and objects are drawn as boxes and their relations are shown as lines with varying arrow-heads and -tails to denote the relation kind

- that class relations can be described as a **composition**, an **aggregation**, and an **association**
- that classes in an inheritance relation can be called an **is-a** relation, while aggregational and compositional relations can be called an **has-a** relation