1 The Object-Oriented Programming Paradigm

Object-oriented programming is a paradigm for encapsulating data and methods into co-hesive units. Key features of object-oriented programming are:

Encapsulation
> Data and methods are collected into a cohesive unit, and an application program need only focus on how to use the object, not on its implementation details.

Inheritance
> Objects are organized in a hierarchy of gradually increased specialty. This promotes a design of code that is of general use, and code reuse.

Polymorphism
> By overriding methods from a base class, derived classes define new data types while their methods still produce results compatible with the base class definitions.

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. Notice that the last steps are identical to imperative programming, **??**. 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.

· use case
· problem statement
· user story

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

A key point in object-oriented programming is that objects should to a large extent be independent and reusable. As an example, the type `int` models the concept of integer numbers. It can hold integer values from -2,147,483,648 to 2,147,483,647, and a number of standard operations and functions are defined for it. We may use integers in many different programs, and it is certain that the original designers did not foresee our use, but strived to make a general type applicable for many uses. Such a design is a useful goal when designing objects, that is, our objects should model the general concepts and be applicable in future uses.

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.

· nouns

· verbs

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

· nouns-and-verbs method

> 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.

· class diagram
· Unified Modelling Language 2
· UML

A class is drawn as shown in **??**. In UML, classes are represented as boxes with their class

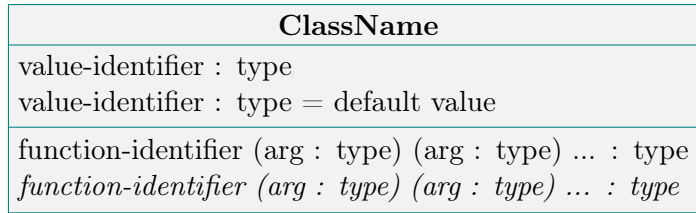| **ClassName** |
|---|
| value-identifier : type |
| value-identifier : type = default value |
| function-identifier (arg : type) (arg : type) ... : type |
| *function-identifier (arg : type) (arg : type) ... : type* |

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

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 **??**.

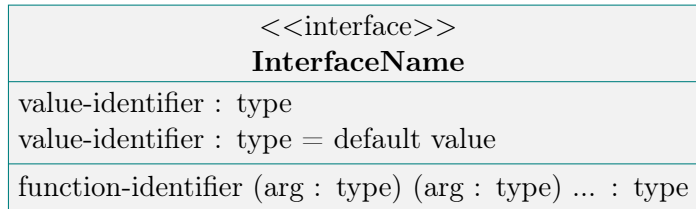| <<interface>> **InterfaceName** |
|---|
| value-identifier : type |
| value-identifier : type = default value |
| function-identifier (arg : type) (arg : type) ... : type |

Figure 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 **??**. Their meaning will be described in detail in the following.
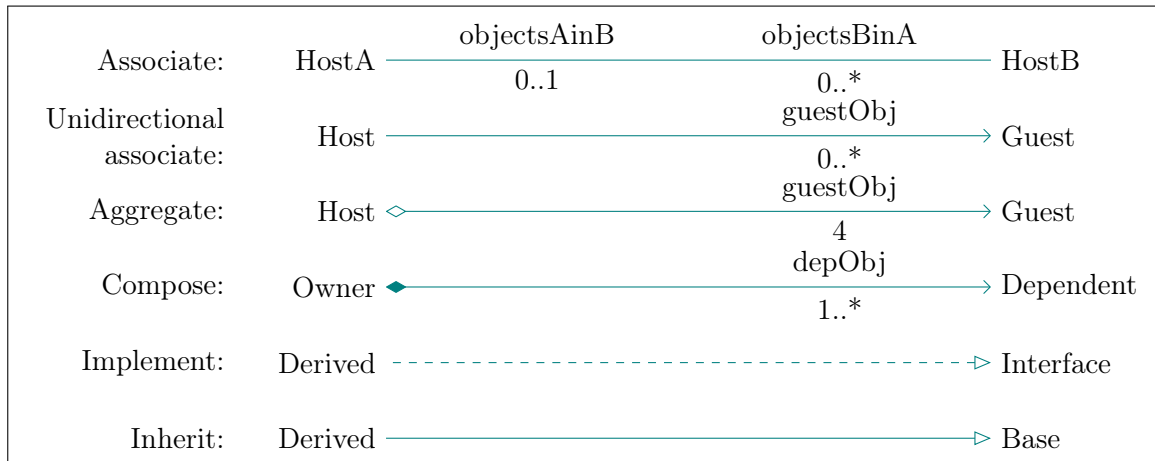

Figure 1.3: Arrows used in class diagrams to show relations between objects.

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 **??**. Aggregational and compositional are specialized types of associations that imply ownership and are often called *has-a* relations. A composition   · has-a relation
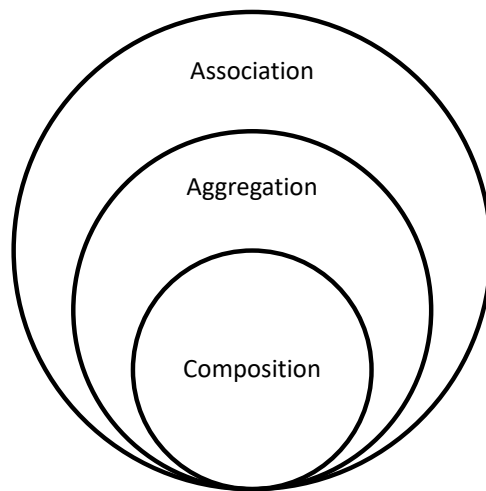
Figure 1.4: The relation between Association, Aggregation and Composition in UML.

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 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*.

· knows-about relation
· association

Association

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 **??**. Association may be annotated by an identifier and a multiplicity. In the
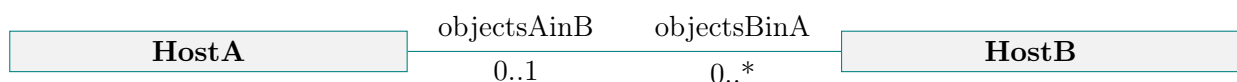

Figure 1.5: Bidirectional association is shown as a line with optional annotation.

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 **??**. If the association is unidirectional, then an arrow is added for emphasis, as shown in **??**. In this example, Host knows about Guest and has one instance of it, and Guest is oblivious about Host.

A programming example showing a unidirectional association is given in **??**.

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.

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

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

```
1  type teacher () =
2    member this.answer (q : string) = "4"
3  type student (t : teacher) =
4    member this.ask () = t.answer("What is 2+2?")
5
6  let t = teacher ()
7  let s = student (t)
8  s.ask()
```

Here, the `student` is unidirectionally associated with a `teacher` since the `student` 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: The `teacher` and `student` objects can access each other's functions, and thus they have an association relation.

· aggregation

Aggregation

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 **??**. Here the Host class has stored aliases to four different Guest objects.

An programming example of an aggregation relation is given in **??**.

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

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

```
1  type book (name : string) =
2    let mutable _name = name
3  type author () =
4    let _book = book("Learning to program")
5    member this.publish() = _book
6  type reader () =
7    let mutable _book : book option = None
8    member this.buy (b : book) = _book <- Some b
9
10 let a = author ()
11 let r = reader ()
12 let b = a.publish ()
13 r.buy (b)
```

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 **??**.
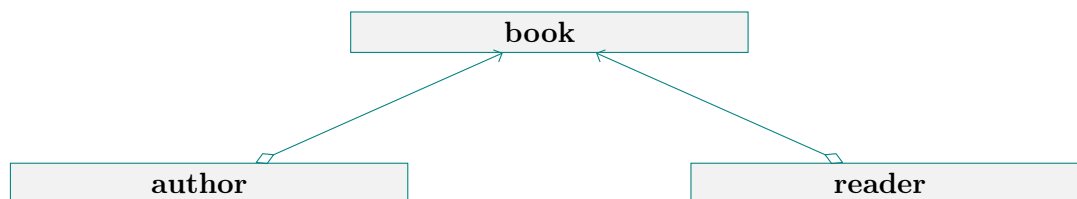

Figure 1.9: A book is an object that can be owned by both an author and a reader.

· composition

Composition

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 **??**. In this example, Owner has created 1 or more objects

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

of type Dependent, and when Owner is deleted, so are these objects.

A programming example of a composition relation is given in **??**.

<div style="border:1px solid #006666;border-radius:8px;padding:8px">

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

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

let bestFriend = dog ()
```
</div>

In **??**, 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: 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

Inheritance   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 **??**. Here two classes inherit the base class.
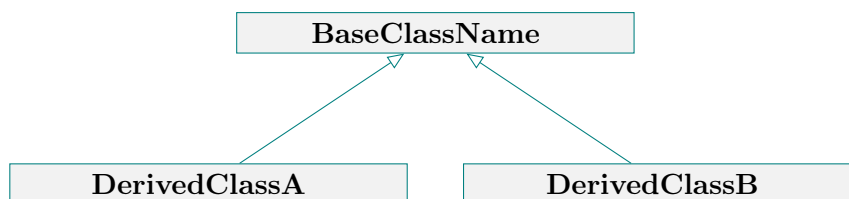

Figure 1.12: Inheritance is shown by a closed arrowhead pointing to the base.

A programming example of an inheritance is given in **??**.

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

```
1  type person (name : string) =
2     member this.name = name
3  type student (name : string, book : string) =
4     inherit person(name)
5     member this.book = book
6  type teacher (name : string, slides : string) =
7     inherit person(name)
8     member this.slides = slides
9
10 let s = student("Hans", "Learning to Program")
11 let t = teacher("Jon", "Slides of the day")
```

In **??**, 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 **??**.
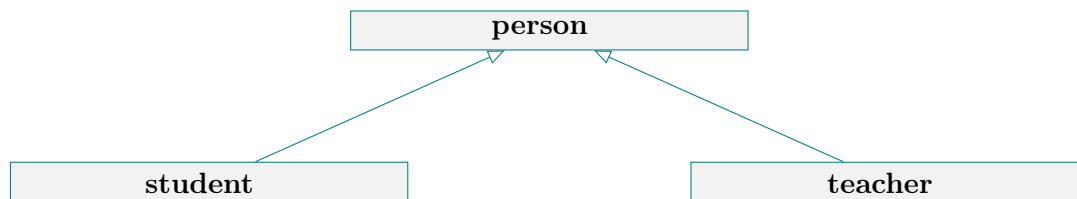

Figure 1.13: A student and a teacher inherit from a person class.

Interface 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 **??**.
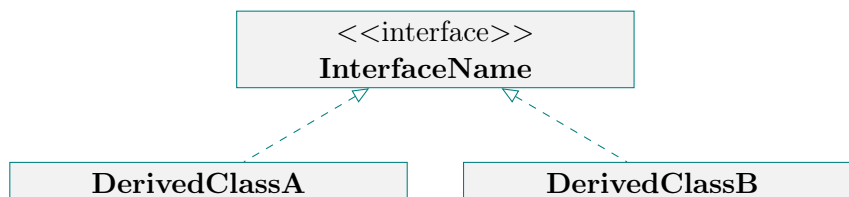

Figure 1.14: Implementations of interfaces is shown with stippled line and closed arrowhead pointing to the base.

A programming example of an interface is given in **??**.

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

```
1  type button =
2    abstract member press : unit -> string
3  type television () =
4    interface button with
5      member this.press () = "Changing channel"
6  type car () =
7    interface button with
8      member this.press () = "Activating wipers"
9  let pressIt (elm : #button) =
10   elm.press ()
11
12 let t = television ()
13 let c = car ()
14 printfn "%s" (pressIt t)
15 printfn "%s" (pressIt c)
```

In **??**, the `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 **??**.
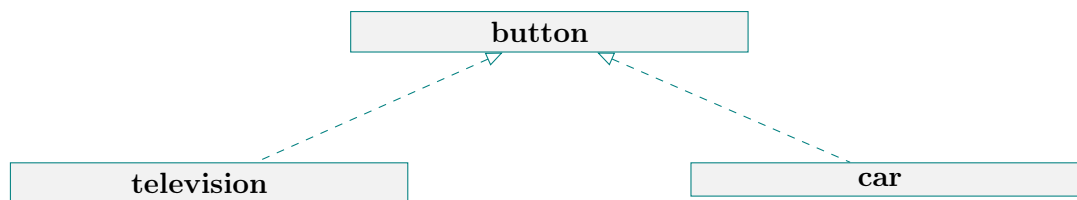

Figure 1.15: A student and a teacher inherit from a person class.

1.2.3 Packages

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

1.3 Programming Intermezzo: Designing a Racing Game

An example is the following *problem statement*:   · problem statement

Problem 1.1

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.
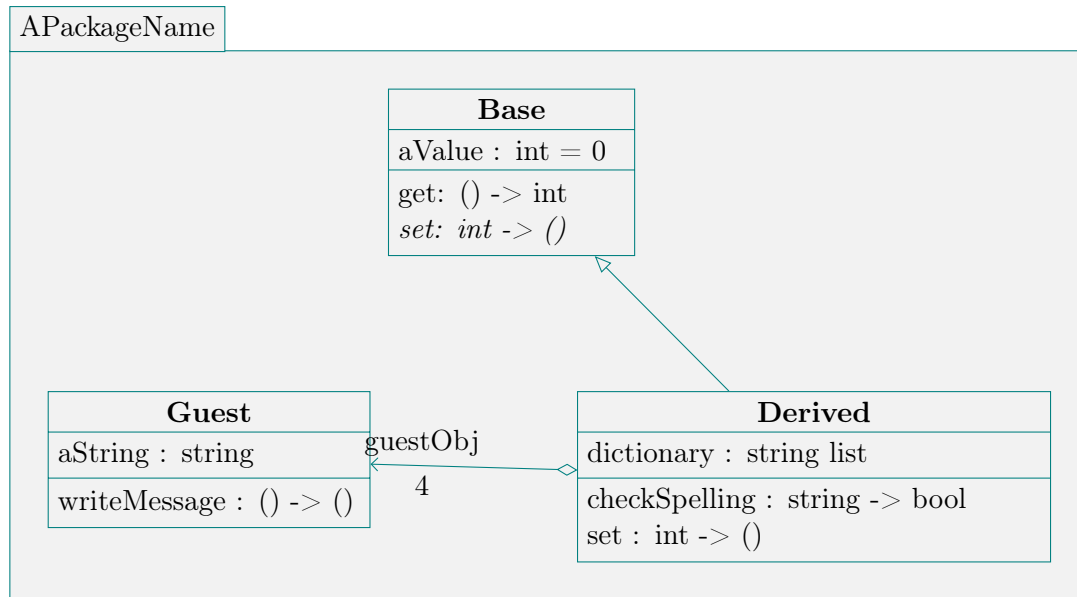
Figure 1.16: Packages are a visualizations of modules and namespaces.

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 · verbs 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 **??**.
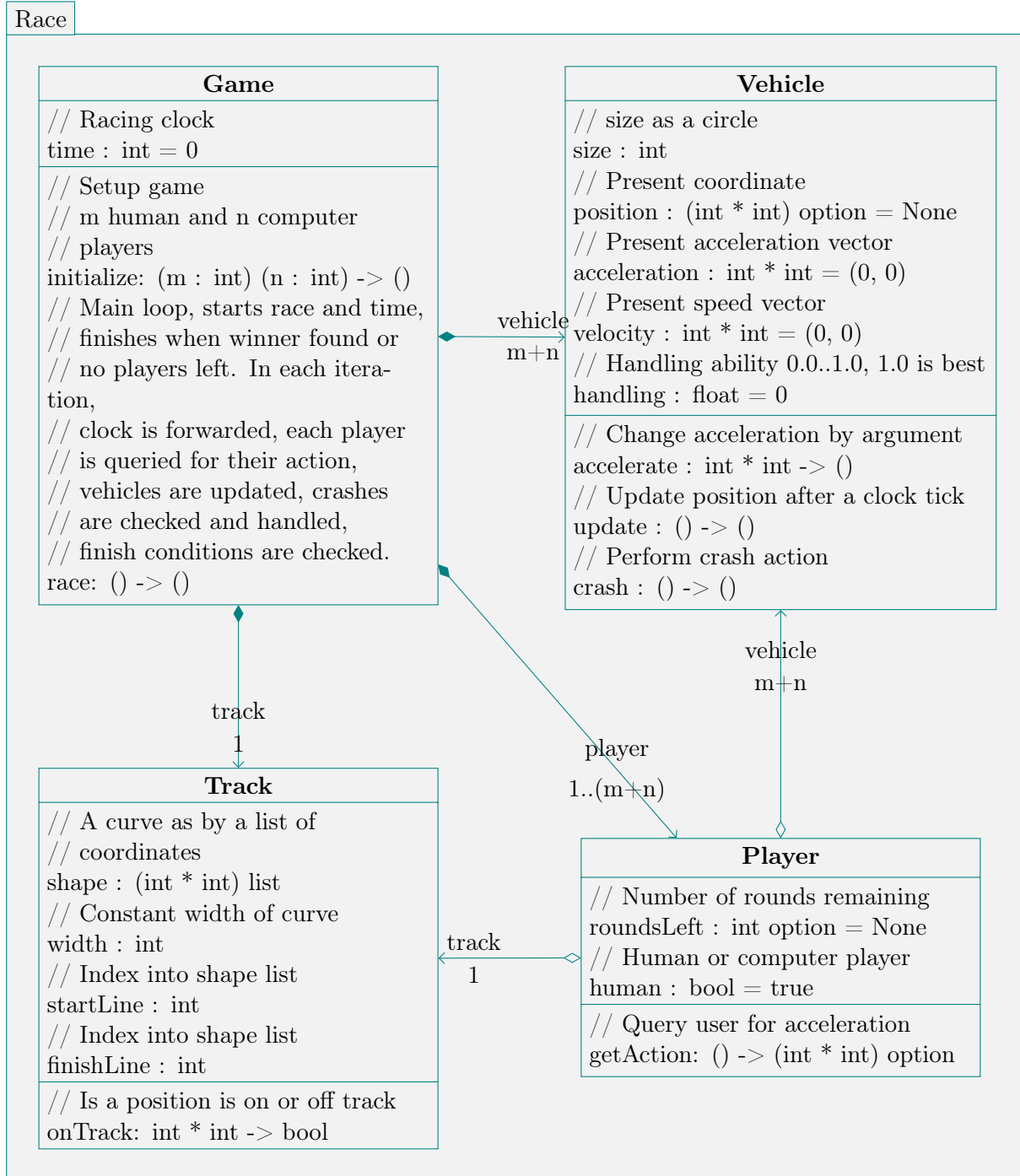

Figure 1.17: A class diagram for a racing game.

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.