

The Construction Phase- Design

Lecture 7

Design - Introduction

By now, we have a full grasp of the problem we are trying to solve (for this iteration). We have developed the Use Cases for the first iteration to a deep level of detail, and we are now ready to design the solution to the problem.

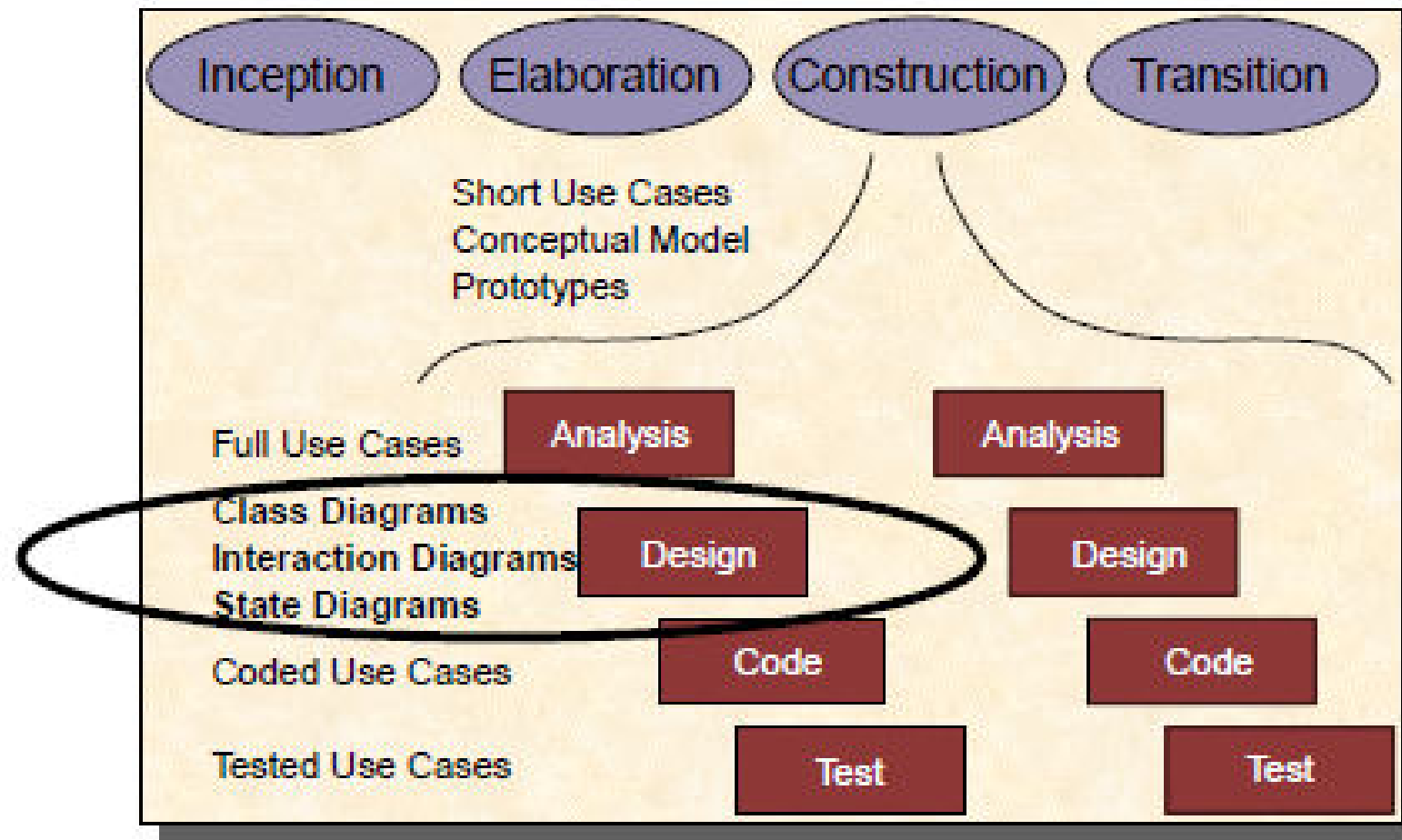
Use Cases are satisfied by objects interacting. So in this stage, we need to decide on what objects we need, what the objects are responsible for doing, and when the objects need to interact.

The UML provides two diagrams to allow us to express the interaction of objects, namely the **Sequence Diagram** and the **Collaboration Diagram**. These two diagrams are very closely related (some tools can generate one diagram from the other one!) Collectively, the Sequence and Collaboration Diagrams are called the **Interaction Diagrams**.

When we decide on the objects we need, we must document the classes of objects we have, and how the classes are related. The UML **Class Diagram** allows us to capture this information. In fact, much of the work of producing the Class Diagram is already done – we'll use the Conceptual Model we produced earlier as a starting point.

Finally, a useful model to build at the design stage is the **State Model**. More details on this later.

So, in design, we are going to produce three types of model – the **Interaction Diagram**, the **Class Diagram** and the **State Diagram**.



The deliverables from the design stage

Collaboration of Objects in Real Life

So, our Use Cases are going to be satisfied by the collaboration of different objects. This is actually what happens in real life. Consider a library. The library is run by a librarian who runs the front desk. The librarian is responsible for handling queries from customers, and is responsible for managing the library index. The librarian is also in charge of several library assistants. The assistants are responsible for managing the library shelves (the librarian couldn't do this – or she wouldn't be able run the front desk efficiently).

The objects in this library system are:

Customer

Librarian

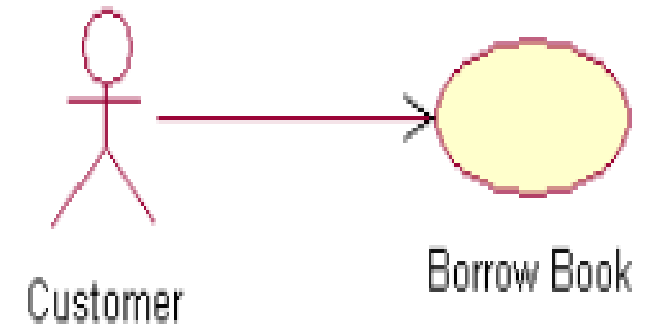
Library Assistant

Library Index

Bookshelf

Let's consider the obvious Use Case – Borrow Book

How would this Use Case be satisfied? Let's assume that the customer does not know where the book is located and needs help. This could be the chain of events:



1. The Customer goes to the librarian and asks for "Applied UML" by Ariadne Training.

2. The Librarian looks in the index for the name of the book. She finds the index card for the book, which says that the book is located at shelf 4F.

3. The Librarian asks the assistant to retrieve the book from shelf 4F.

4. The Librarian gets the requested book and returns it to the librarian.

5. The librarian checks out the book and hands it to the customer.

Asks
Librarian for
book



Sends
assistant to
get the book



Librarian looks up
book in index;
finds location



Man goes to library
to borrow a book



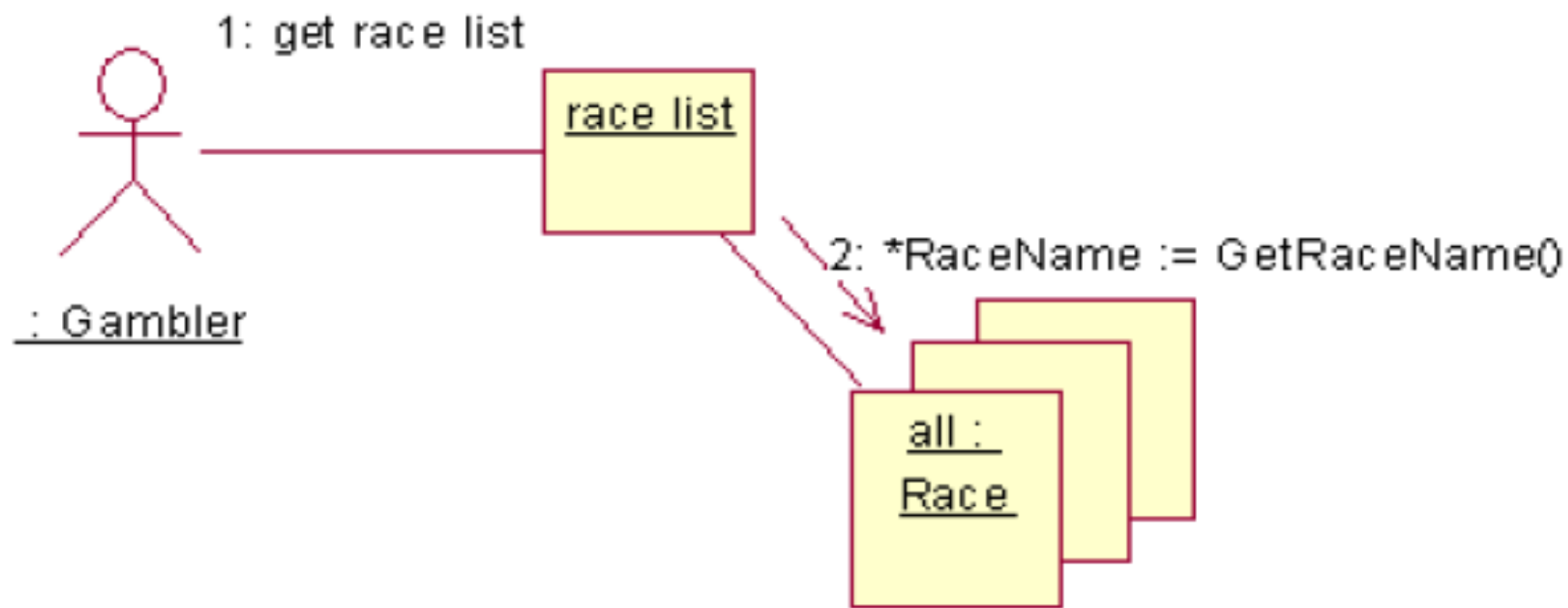
Chain of events for "Borrow Book"

Even though this example is very simple, it was still not particularly easy to consider the responsibilities of each object. This is one of the key activities of Object Oriented Design – getting the responsibilities of each object correct. For example, if I had decided to let the librarian physically retrieve the book, I'd have designed a very inefficient system. Later on in this chapter, we'll present some guidelines for allocating responsibilities to objects.

Collaboration Diagrams

In this section, we will look at the syntax of the UML Collaboration diagram. We will look at how to use the diagram in the next section.

A collaboration diagram allows us to show the interactions between objects over time. Here is an example of a completed collaboration diagram:



Collaboration Diagram

Collaboration Syntax : The Basics

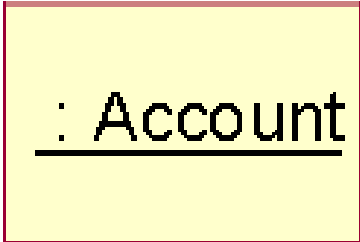
A class on the collaboration diagram is notated as follows:



Account

A yellow rectangular box with a red border, representing a class in a UML Collaboration Diagram.

An instance of a class (in other words, an object) is notated as follows:



: Account

A yellow rectangular box with a red border, representing an object in a UML Collaboration Diagram. The text inside is underlined.

Sometimes, we will find it useful to name an instance of a class. In the following example, I want an object from the Account class, and I want to call it “first”:

first :
Account

If we want one object to communicate with another object, we notate this by connecting the two objects together, with a line. In the following example, I want a “bet” object to communicate with an “account” object:



Once we have notated that one object can communicate with another, we can send a named message from one object to the other. Here, the “bet” object sends a message to the “account” object, telling it to debit itself:



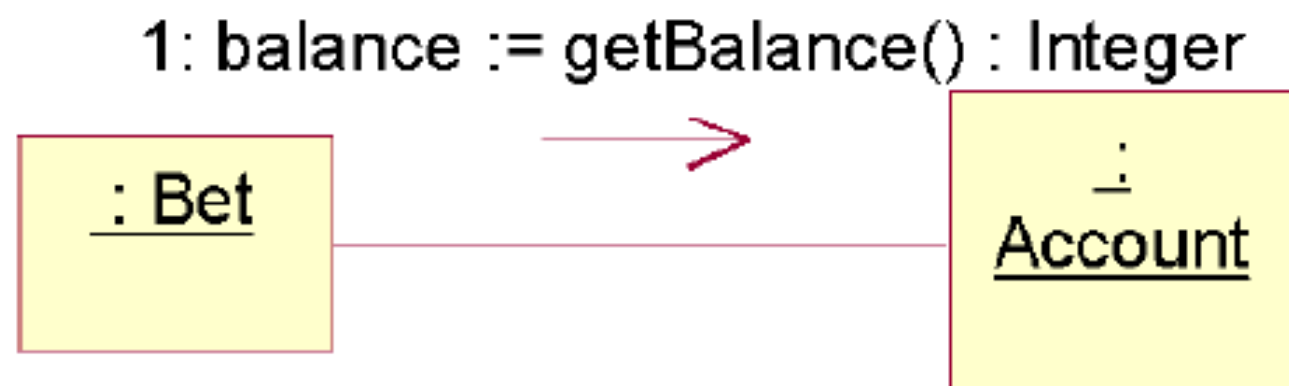
If we want to pass parameters with the message, then we can include the parameter in the brackets, as follows. The datatype of the parameter (in this case, a class called “Money”) can be shown, optionally.



A message can return a value (analogous to a function call at the programming stage). The following syntax is recommended in the UML standard if you are aiming for a language neutral design. However, if you have a target language in mind, you can tailor this syntax to match your preferred language.

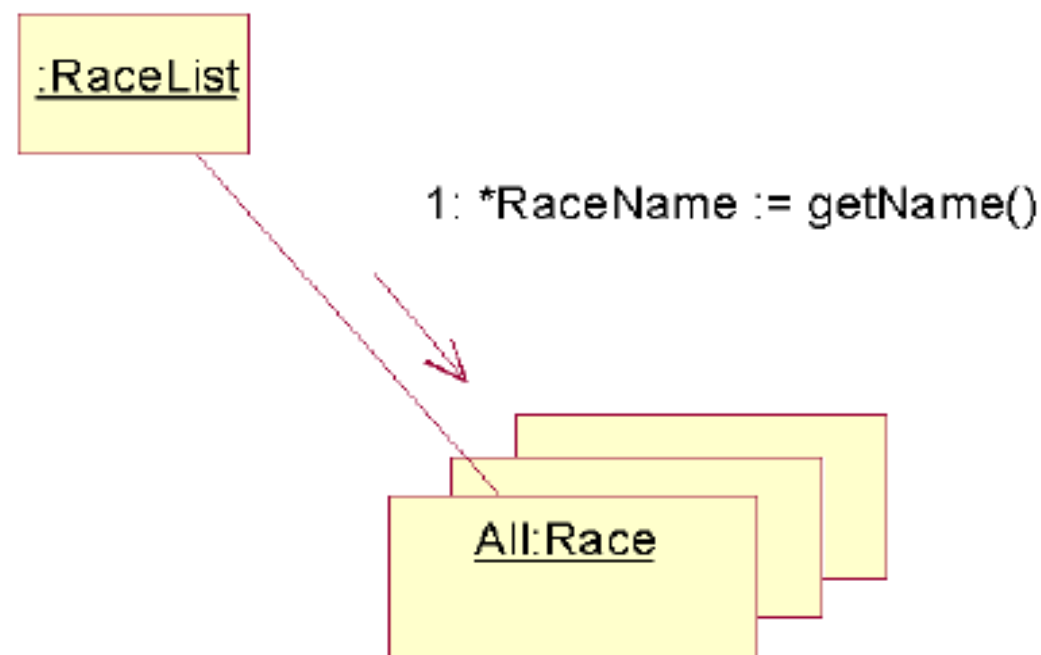
```
return := message (parameter : parameterType) : returnType
```

In the following example, the bet object needs to know the balance of a particular account. The message “getBalance” is sent, and the account object returns an integer:



Collaboration Diagrams : Looping

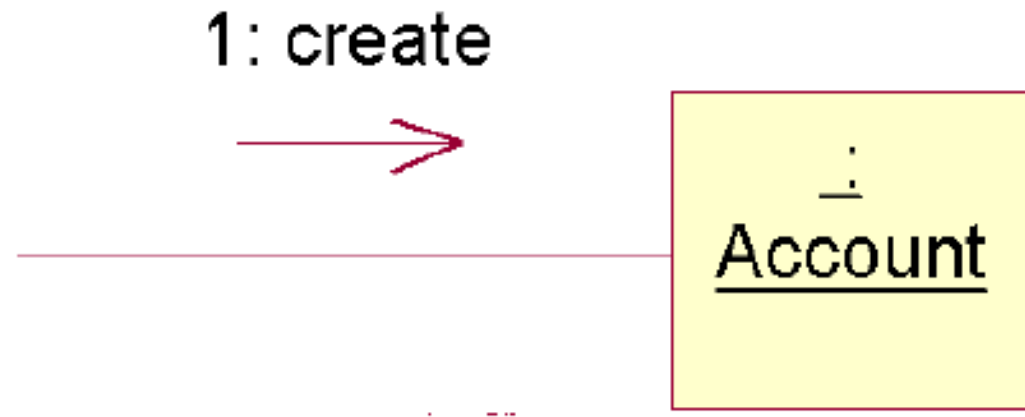
If we need to introduce a loop into a collaboration diagram, we use the following syntax. In this example, an object from the “Race List” class needs to assemble itself. To do this, it asks every member of the “Race” class to return its name.



The asterisk denotes that the message is to be repeated. Rather than specifying an individual object name, we have used the name “All” to denote that we are going to iterate across all of the objects. Finally, we have used the UML notation for a collection of objects with the “stacking” of the object boxes.

Collaboration Diagrams : Creating new objects

Sometimes, one object will want to create a new instance of another object. The method for doing this varies between languages, so the UML standardises creation through the following syntax:



The syntax is rather strange, really – you are sending a message called “Create” to an object that doesn’t yet exist!

!! In reality, you are sending a message to the class, and in most languages you are also calling the constructor.

Message Numbering

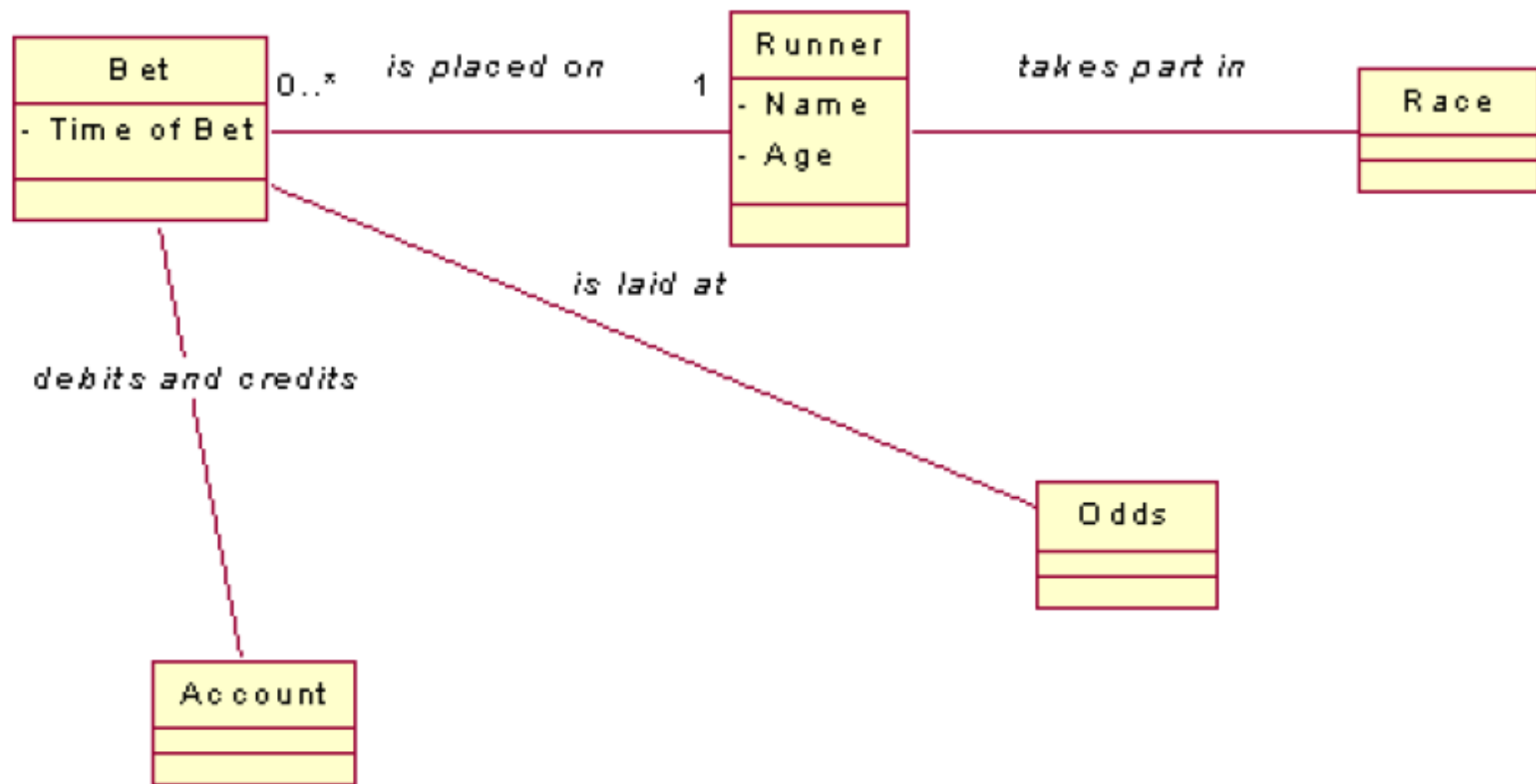
Notice that all of the messages we have included so far have a mysterious “1” alongside them? This indicates the order in which the message is executed in order to satisfy the Use Case. As we add more messages , we increment the message number sequentially.

Collaboration Diagrams : Worked Example

Let's pull all of that theory together and see how the notation works in practise. Let's build the "place bet" Use Case using a collaboration diagram.

This example is far from perfect, and leaves a lot of questions unanswered (we'll list the unresolved issues at the end of this chapter). However, as a first cut, the example should illustrate how collaboration diagrams are built. '

To build this diagram, we need some objects. Where do we get the objects from? Well, we will certainly have to invent some new objects as we go along, but many of the candidate objects should come directly from our old friend, the conceptual model we built at the elaboration phase. Here is the conceptual model for the betting system:



Betting System Conceptual Model

Where there are associations, such as “is placed on”, we will probably use these associations to pass messages on the collaboration diagram. We could possibly decide that we need to (for example) pass a message between “account” and “race”. This is perfectly valid, but as the association was not discovered at the conceptual stage, we might have possibly broke some of the customer’s requirements. If this happens, *check with the customer!*

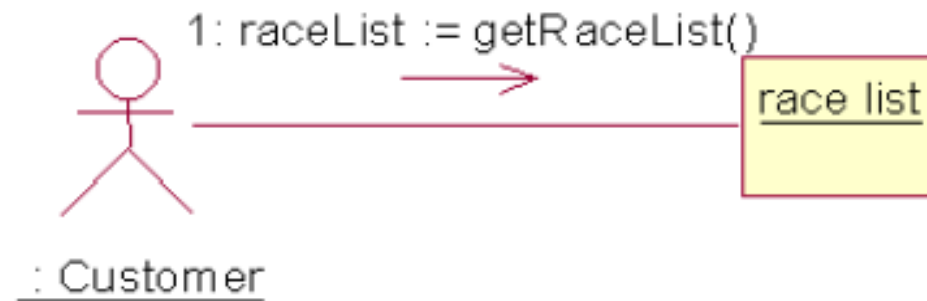
With the Use Case description and the conceptual model in mind, let’s build the collaboration for “Place Bet”.

1. First of all, we start with the initiating actor, the customer. The actor symbol is not strictly part of the UML collaboration diagram, but it is extremely useful to include it on the diagram anyway.



Use Case :
Place Bet

2. Now, according the Use Case description, when the customer selects the “place bet” option, a selection of races are presented. So we’ll need an object that contains a full list of the races for today, so we create an object called “Race List”¹³ This is an object which was not represented on the conceptual model. This is called a *design class*.

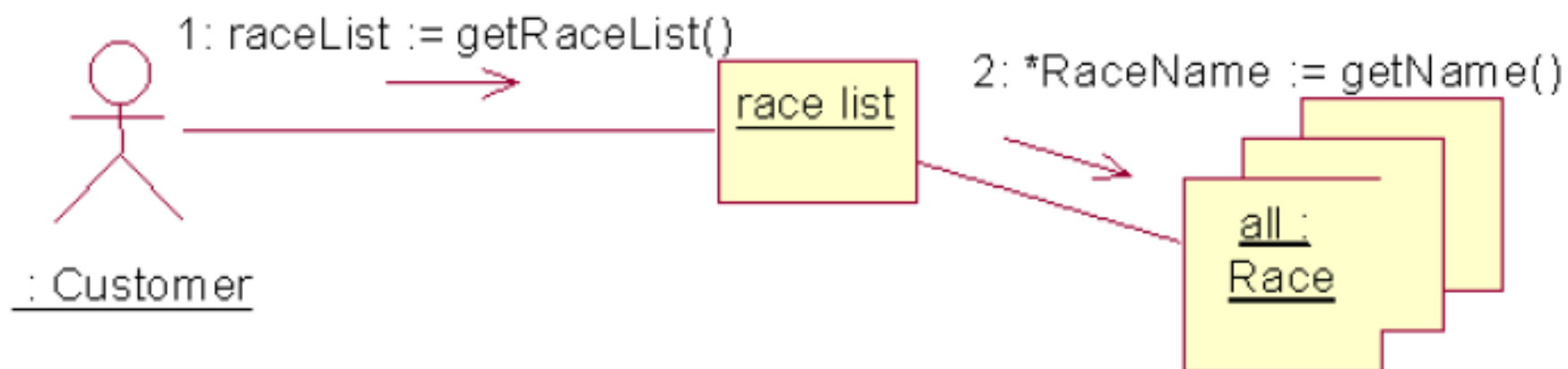


Use Case :
Place Bet

¹³ This will be a container, or array, or something similar, depending on the target language

3. So, the actor sends a message to the new “Race List” object called “getRaceList”. Now, the next job is for the race list to assemble itself. It does this by looping around all of the Race objects, and asking them what their name is.

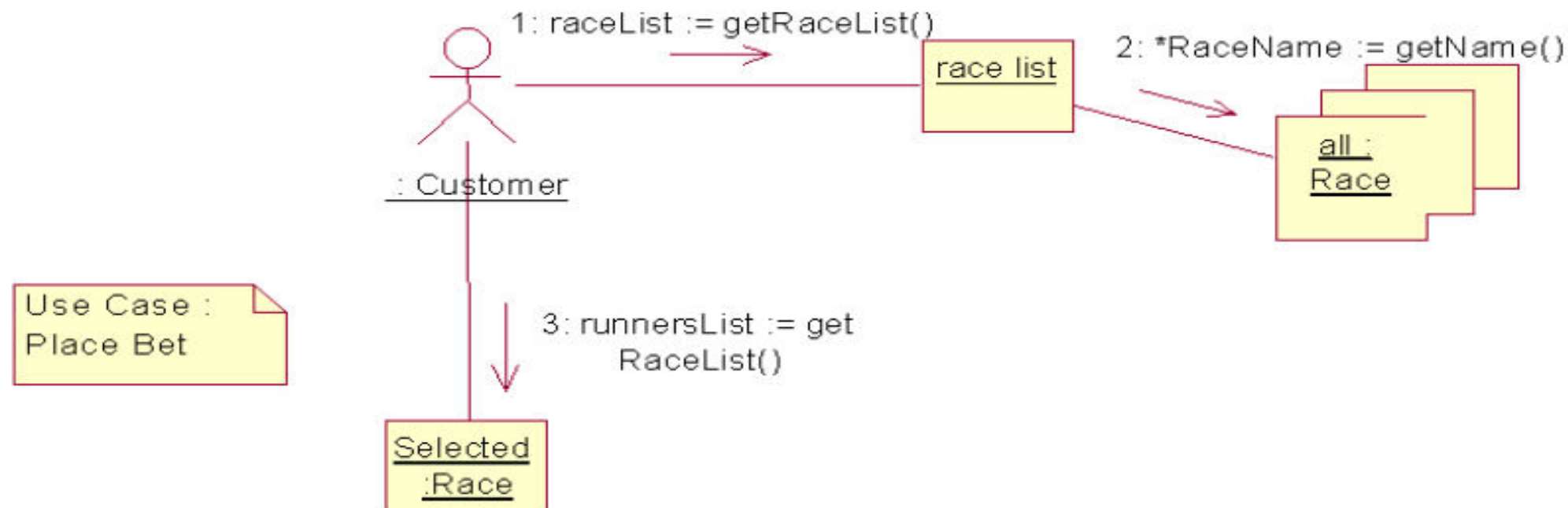
The race object has been taken from the conceptual model.



Use Case :
Place Bet

4. Next, we assume the race list is now returned back to the customer. The ball is now in their court, and according to the Use Case description , the user now selects a race from the list.

We may now assume that a race has been selected. We now need to get a list of runners for the selected race, and to find that out I have decided to make the race object responsible for keeping a list of its runners. We'll see in future chapters why and how this kind of decision is made.

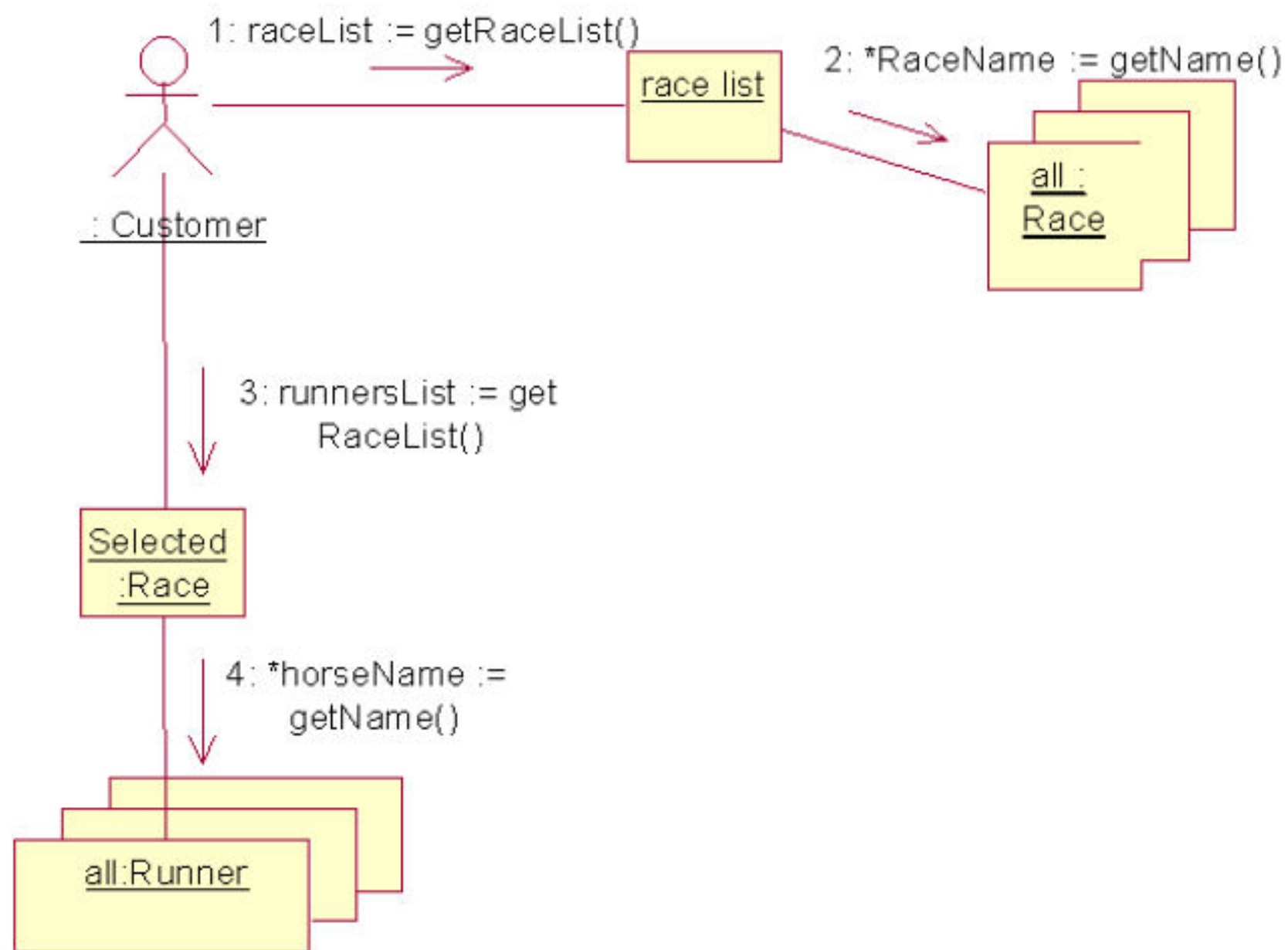


So, message number 3 is sent to the selected race, and the message is asking for a list of runners for that race.

5. How does the race object know what runners are part of that race? Once again, we'll do this using a loop, and we'll get the race object to assemble a list of runners.

How this is actually achieved in the real system is not trivial. Clearly, we are going to store these runners on a database of some kind, so part of the physical design process is going to be to construct a mechanism for retrieving the horse records from the database. For now, however, it is enough to say that the selected Race object is responsible for assembling a list of the runners for that race.

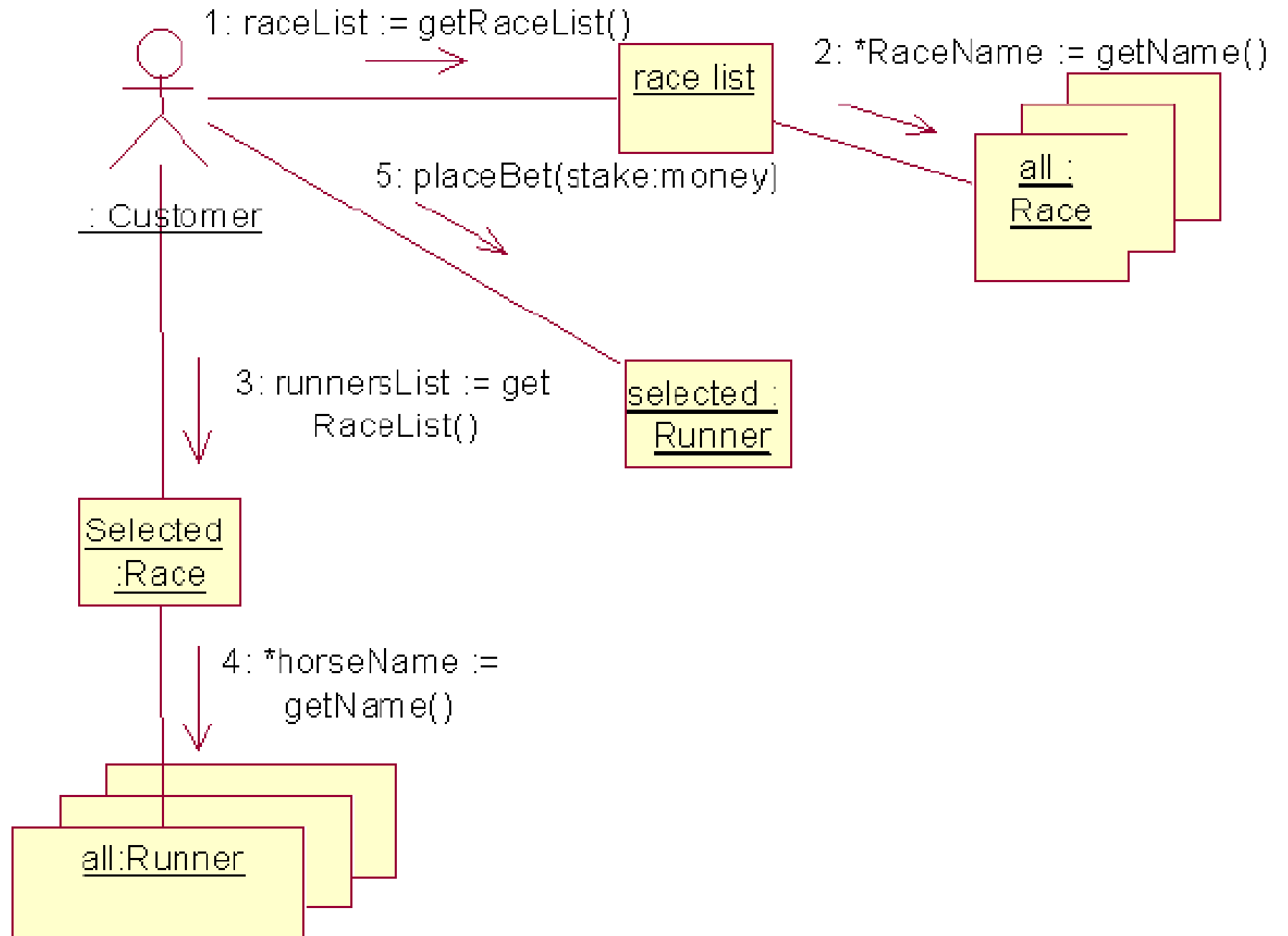
Use Case :
Place Bet



6. The list of runners is now returned back to the customer. The ball is again in their court, and according to the Use Case description, they must now select a runner, and then select their stake money.

Now that we know the runner and the stake, we can send a message to the selected runner, and tell it that a bet has been placed upon it.

Use Case :
Place Bet



By building this collaboration diagram, we have NOT mapped out exactly what the code will look like. The following issues have been left unresolved:

1. We have not mentioned anything on the diagram about how the user inputs data into the system, and how the data (such as the list of runners) is output on the screen. Somehow all of this happens as if by magic “inside” the actor.

We'll see later that this is good design. We want to make our design as flexible as possible, and by including detail about the User Interface at this stage, we are tying ourselves down to one specific solution.

2. How does the “Race” object find out what runners are part of that race? Clearly, there is some kind of database (or even network) operation going on here. Again, we do not want to tie our design down at this stage, so we defer these details until later.

3. Why have we made the “runner” object responsible for tracking which bets have been placed on it? Why didn’t we create another class, perhaps called “bet handler” or “betting system”? This issue will be explored in the following chapter.

What we **have** done is decide on the responsibilities of each class. What we are doing, in effect, is building upon the conceptual model we produced at the elaboration stage.

Some Guidelines For Collaboration Diagrams

As we progress through this course, we'll expand on how to produce good diagrams. For now, keep the following guidelines in mind:

1. **Keep the diagram simple!!!** It seems that in our industry, unless a diagram covers hundreds of pages of A4 and looks very complicated, the diagram is trivial! The best rule to apply to the collaboration diagram (and the other UML diagrams too) is to keep them as simple as possible. If the collaboration for a use case gets complicated, break it up. Perhaps produce a separate diagram for each user/system interaction.
2. **Don't try to capture every scenario.** Every use case comprises a number of different scenarios (the main flow, several alternatives and several exceptions). Usually, the alternatives are fairly trivial and not really worth the bother of including.

A common mistake is to cram every scenario on one diagram, making the diagram complex and difficult to code.

3. **Avoid creating classes whose name contains “controller”, “handler”, “manager” or “driver”.** Or at least, be suspicious if you do come up with an object with such a name. Why? Well, these classes tend to suggest that your design is not object oriented. For example, in the “Place Bet” use case, I could have created a class called “BetHandler” that deals with all of the betting functionality. But this would be an action oriented rather than an object oriented solution. We already have the “Bet” object from the collaboration diagram, so why not use it, and give it the responsibility of handling bets?
4. **Avoid God classes.** Similarly, if you end up with a single object that does a lot of work and does not collaborate much with other objects, you have probably built an action oriented solution. Good OO solutions consist of small objects who don't do too much work themselves, but work with other objects to achieve their goal. We'll go into much more detail about this later.

Chapter Summary

In this chapter, we began to construct a software solution to our Use Cases. The collaboration diagram enabled us to allocate responsibilities to the classes we derived during elaboration.

We touched on a few issues we should be aware of when allocating responsibilities, even though we need to learn more about this topic later. An example for “Place Bet” was studied.

In the next section, we’ll see how we can expand the Conceptual Model and progress it towards a true Class Diagram.