

Messaging



Now that object / class properties and methods have been introduced, we can look at how objects interact or communicate with each other. This process called **Messaging** is the way objects communicate and will be covered in this section.

Messaging

Most systems require interaction or communication between objects. **Messaging** is the way objects communicate. Sending a **message** to an object causes that object to perform an operation on itself. It is the object's responsibility to look at the message and think about what is appropriate to perform on itself. Each class specifies code to be used in response to messages. The code corresponding to a particular message is known as the method for that message. Methods have been dealt with in previous sections of this chapter.

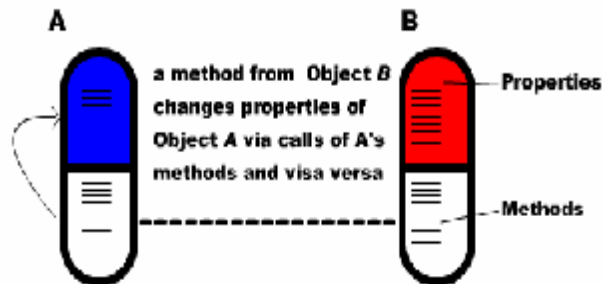


Figure 1: Messages from object B to object A

Have a look at Eddie in the picture below. Let's model this situation using two objects. One is Eddie (belonging to the *Sloopy* class) and the other one is Eddie's skateboard, belonging to a new *Skateboard* class.



Figure 2: Object *Eddie* and object *Skateboard*

If Eddie wants to ride his skateboard, a message – let's call it *ride* has to be sent from Eddie to the skateboard. As a result, the state (that is the value of a certain property) of the skateboard will change; it is likely to be at a different location.

Object	Property: Value	Message	Property: Value
Skateboard	x: 200 y: 100	→ <i>ride</i>	x: 400 y: 500

Figure 3: The value of Skateboard changes after the message has been sent

When initiating a message the outcome is almost always a change in the state of either the object that has initiated the message or the object that has received the message or both. Methods are used to represent messages. Each method / message has three parts:

- The **object** to which the message is addressed
- The name of the **method** to perform
- Any **parameters** the method needs

Lets look at the pseudo code.

```
sloopy.ride (skateboard)
  ↑   ↑   ↑
  object method parameter
```

A more realistic example is the method declaration below where *x* and *y* parameters tell the message how far the Sloopy should ride and a *speed* parameter telling the message how fast to ride.

```
sloopy.ride (skateboard, 200, 400, 5)
```

Within the <implementation> part of the *ride* method, the values for *x* and *y* will change as follows:

```
Skateboard.x := Skateboard.x + 200
Skateboard.y := Skateboard.y + 400
```

UML fully supports the representation of methods and therefore fully supports messages.

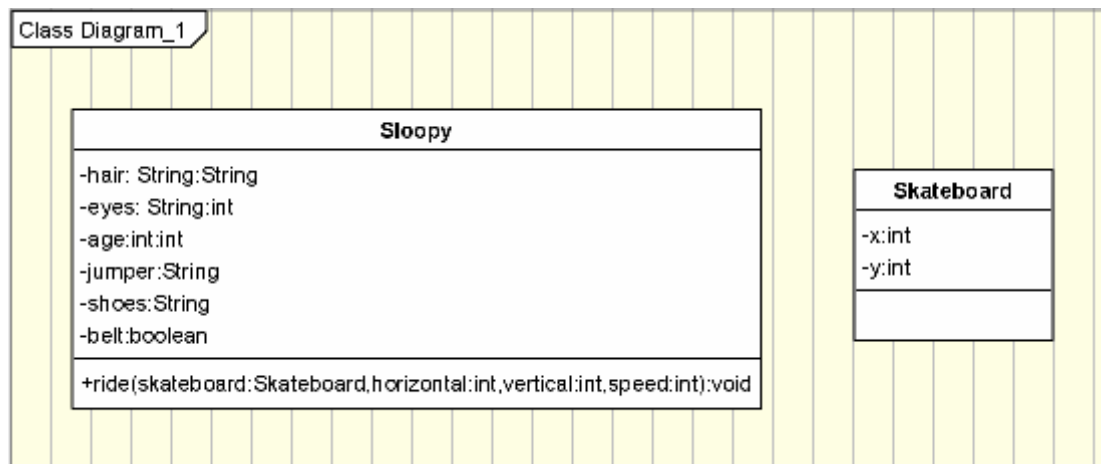


Figure 4: UML of *Sloopy* class (with *ride* message) and *Skateboard* class

The model contains two classes *Sloopy* and *Skateboard*. The *Sloopy* class includes one method / message *ride* with four parameters. *skateboard* is an instance from the *Skateboard* class, *horizontal* and *vertical* specify the how far *skateboard* has to move left/right and up/down, respectively, while *speed* tells the method how fast to move.



We have seen how **methods** and **messages** can be used to model behaviour and the interaction of objects.

Now that structure and behaviour have been introduced, key elements are available to model a complete system. The challenge from now on is how to use the constructs available in the most suitable way.