Department of Computer Science

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Tackling Design Patterns
Chapter 21: UML Communication Diagrams

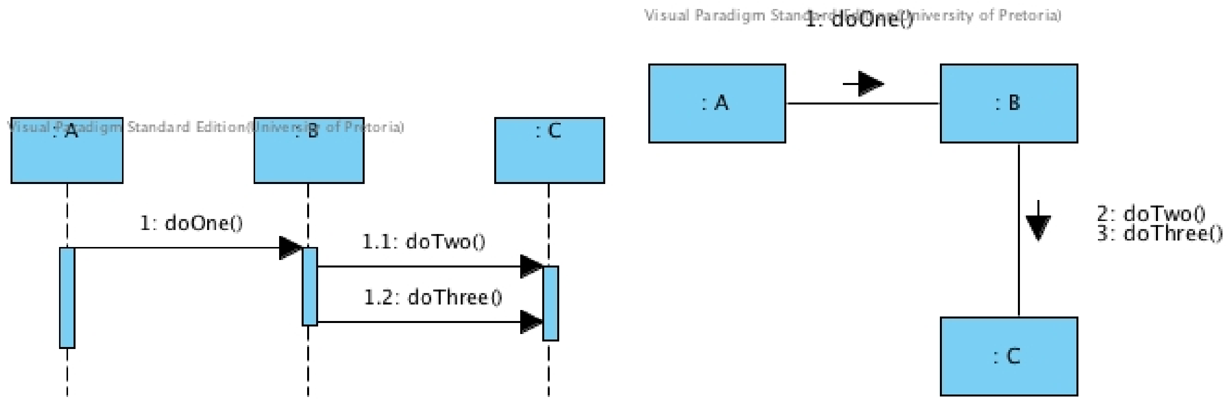
Contents

21.1 Introduction

UML 2.0 includes a number of interaction diagrams. These are sequence diagrams, interaction overview diagrams, timing diagrams and communication diagrams. In this lecture we look specifically at communication diagrams. They are used to model which objects interact with one another in terms of the messages they pass to one another. While all interaction diagrams model these interactions, communication diagrams emphasise relations between the originators and the receivers of messages.

The way in which object oriented programs systems produce useful results is mainly through passing messages between objects. These messages appear in the form of method calls. A communication diagram is used to visualise which objects are involved in the execution of a reaction to some event. In UML 1 these diagrams were called Collaboration diagrams.

Communication diagrams and sequence diagrams are very similar. Figure 1 illustrate the differences and similarities between these types of diagrams. As can be seen in the figure they both model method calls between objects. They use different layouts to show the same information.


Visual Paradigm Standard Edition(University of Pretoria)
: A
: B
: C
1: doOne()
1.1: doTwo()
1.2: doThree()
Visual Paradigm Standard Edition(University of Pretoria)
1: doOne()
: A
: B
2: doTwo()
3: doThree()
: C

Sequence diagram Communication diagram

Figure 1: Comparison of UML Sequence diagrams and UML Communication diagrams

The difference between sequence diagrams and communication diagrams can be summarised as follows:

- sequence diagrams
  - Easier to read call-flow sequence
  - More notation options allows for higher expressiveness
- communication diagrams
  - Easier to observe which objects are involved in the communication
  - Free format spacing allows for easier maintainance

21.2 Basic Notational Elements

The basic elements in a communication diagram are objects. A communication diagram shows that there exist communication between objects by connecting them with a link line. The detail of the communication between objects is visualised in terms of the messages that are passed between these objects. Often the order in which messages are passed are indicated by using sequence numbers.

21.2.1 Objects and Actors

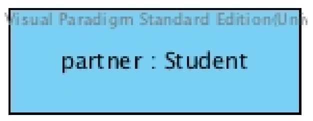


Figure 2: An object

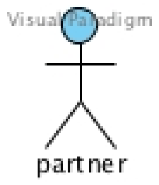


Figure 3: An actor

Figure 2 shows an object. The same notation that is used in UML Object diagrams and UML Sequence diagrams to show objects, are used in UML Communication diagrams. An object is shown as a rectangle containing the object name and the class name of which it is an instance. The object name is separated from the class name using a colon. An object my be anonymous. For example when an object is created but not assigned to a variable. When an object is anonymous it is shown as a rectangle containing only the class name of the class of which it is an instance. In this case the class name is preceded by a colon. The objects shown in Figure 4 are anonymous objects.

Figure 3 shows the same object shown in Figure 2 as an actor. Any object may be shown as an actor. An actor is a stick man labeled with the object name and the class name of which it is an instance. Often actors are anonymous. When anonymous objects are named, technically the class name should be preceded by a colon. However, this colon is often deemed redundant.

The actor notation for objects are usually reserved to represent a user. Such user is labeled with a name (without a colon) to specify the type of user. This usage can be seen in Figure 7.

21.2.2 Link line

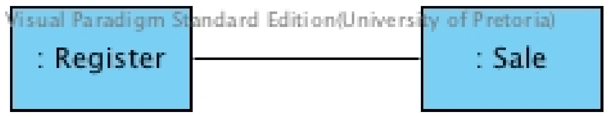


Figure 4: A link line between objects

When objects exchange messages at any stage during execution, they are connected to one another using a link line. A link line is a solid line connecting two objects. It merely indicates that communication between the objects is possible. It is preferable that link lines do not intersect in a diagram.Intersecting link lines, however, are often unavoidable.

21.2.3 Messages

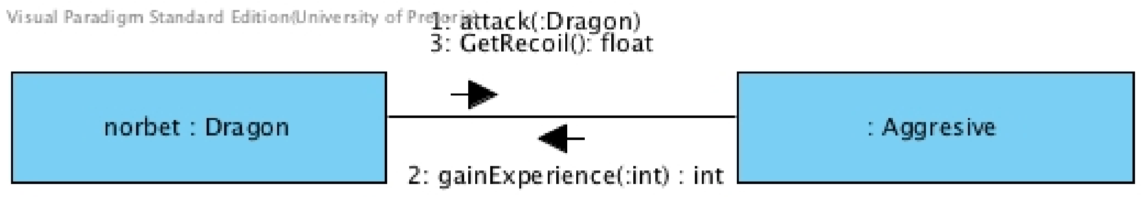

Visual Paradigm Standard Edition(University of Pretoria)
1: attack(:Dragon)
3: GetRecoil(): float
norbet : Dragon
: Aggresive
2: gainExperience(:int) : int

Figure 5: Messages flowing on a link line between objects

The detail of the communication between objects is visualised in terms of the messages that are passed between objects. A message is indicated with a small arrow showing the direction of the method call. The arrow head points toward the object that has the method that is executed. The arrow is usually labeled with the signature of a method that is called. The position of the label is not prescribed. It should, however, be placed in such a way that it is clear which arrow belongs to which label. The signature of the method need only include detail that is relevant to the situation. Usually the data types of its parameters and return type are sufficient. If the return type of a method is `void`, the return type is often omitted. Sometimes descriptive names without data types are used. As can be seen in the label with number 3 in Figure 7, the label may even be a description of an action in natural language.

In UML Sequence Diagrams there is an explicit distinction between methods that are synchronous and methods that are asynchronous. Synchronous methods are those that returns value. As explained in Section **??**, different arrow head shapes are used to indicate the distinction. Contrary to this, UML Communication diagrams do not distinguish between methods returning values to their callers and methods that do not return values. In UML Sequence diagrams a returning dashed line is used to indicate the passing of a return value. In UML Communication diagrams this detail is omitted.

When multiple messages are sent between two objects, they are indicated along the same single link line between objects. As can be seen in Figure 5, messages in *both* directions may flow along the same link line.

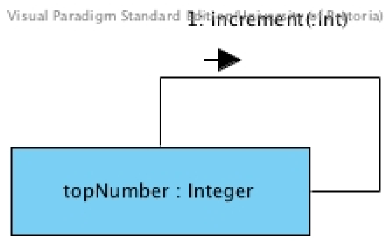


Figure 6: A reflexive message

A reflexive message is when an object calls a method that is defined in its own class. More often than not, reflexive messages are not shown in communication diagrams. It is, however, permissible to show a reflexive message as shown in Figure 6.

21.2.4 Sequence numbers

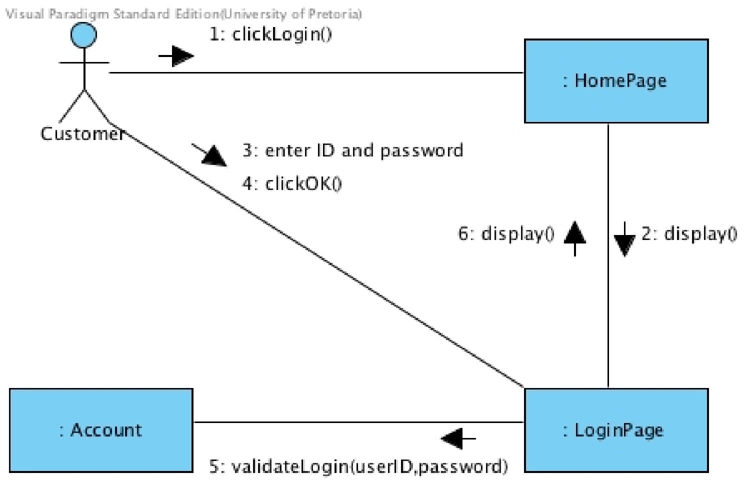


Figure 7: Messages that are numbered to indicate the order of execution

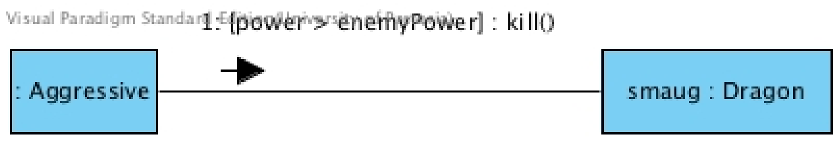
Although the order of messages are usually not relevant in communication diagrams. The order of actions may be shown by numbering the messages in the order that they are executed as shown in Figure 7 taken from [2]. The numbering scheme for numbering messages in a communication diagram is not prescribed and may include sub-numberings. Figure 11 is an example of a communication diagram that uses a decimal numbering scheme with sub-numberings.

21.3 Advanced notational elements

Sometimes it is necessary to indicate advanced detail about communications between objects. In most cases when this is required a UML Sequence Diagram will serve better to model the required detail. Additional detail that can be modelled with some limitations is the modelling of conditional actions and repetition. In this section we deal with these special cases.

21.3.1 Creation and Destruction

It is important to realise that objects in a communication diagram are *instantiated* instances of classes in a system. Creation and destruction of objects are not shown UML communication diagrams.



Visual Paradigm Standard Edition(University of Pretoria)
1: [power > enemyPower] : kill()
: Aggressive
smaug : Dragon

Figure 8: Syntax for a conditional flow

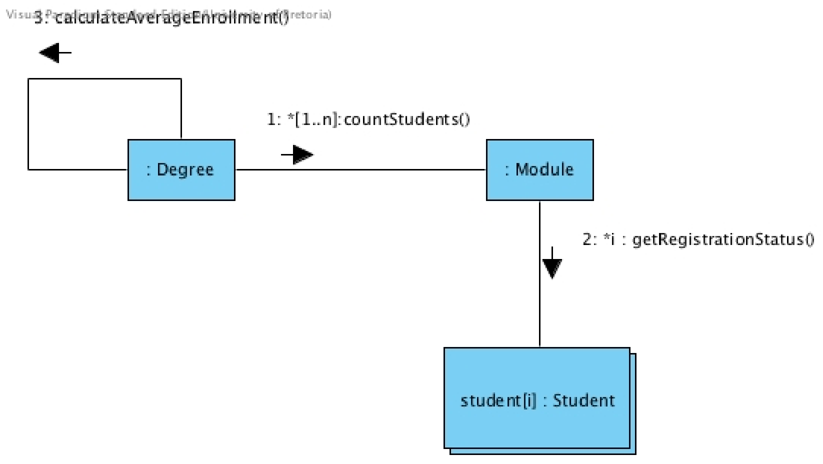
21.3.2 Conditional messages

Branching happens when the program flow contains conditional statements. Figure 8 shows the notation to model a conditional message. The condition that needs to be true for a message to be called is shown as a guard. The syntax to indicate the condition in square brackets is the same as in UML Activity diagrams and UML State diagrams. In this example an anonymous Aggresive object will order a Dragon object named smaug to kill if its power is greater than that of its enemy.

21.3.3 Iteration

Figure 9 contains notations to model iteration. Iteration is modeled by showing a * before the method that is repeatedly executed. The condition that needs to be true for operation to be repeated may be shown as a guard next to the *. The guard condition may be shown as

- a boolean expression (See Figure 10)
- starting and ending values (See Figure 9)
- a counter name (See Figure 9)
- a natural language expression (See Figure 11)



Visual Paradigm Standard Edition(University of Pretoria)
3: calculateAverageEnrollment()
1: *[1..n]:countStudents()
: Degree
: Module
2: *i : getRegistrationStatus()
student[i] : Student

Figure 9: Communications including loops

Figure 9 models the communication of a system that loops through a number of modules and within each module loops through a number of students in order to calculate the average enrolment per module for a certain degree. In this example the message `countStudents()` is executed with a starting value of 1 and ending value of n. In this case n represents the number of modules. The `getRegistrationStatus()` statement is executed for each value of i. In this case the counter i is used to iterate over the students in the module.

The guard condition for a repeated statement in a UML Communication diagram is optional (it may be omitted). In Figure 13 the guard is omitted because the condition for the loop is obvious from the context and therefore deemed redundant.

21.4 Examples

21.4.1 Get relevant reports

The communication diagram in Figure 10 correlates with the loop fragment of the sequence diagram shown in Figure 14 of L16_Sequence.pdf. Note how the condition for execution of the `add()` message is given as a guard in terms of a boolean expression. The guard of the loop fragment is shown as a guard for the repetition of `getNextReport()` message.

Note that although the `getRequiredSecurityLevel()` message appears in the loop fragment in Figure 14 of L16_Sequence.pdf, it is not shown as an action that is repeated, because this message is only sent once to a specific instance. Every time the loop is executed, it executes with another instance of `Report` that gets the message only once. Similarly the `add()` message as well as the `hasAnotherReport()` messages are also not shown as a repeated statements. Having them starred here would mean that they are loops nested inside the loop indicated by the star preceding the `getNextReport()` message, which is not the case here.

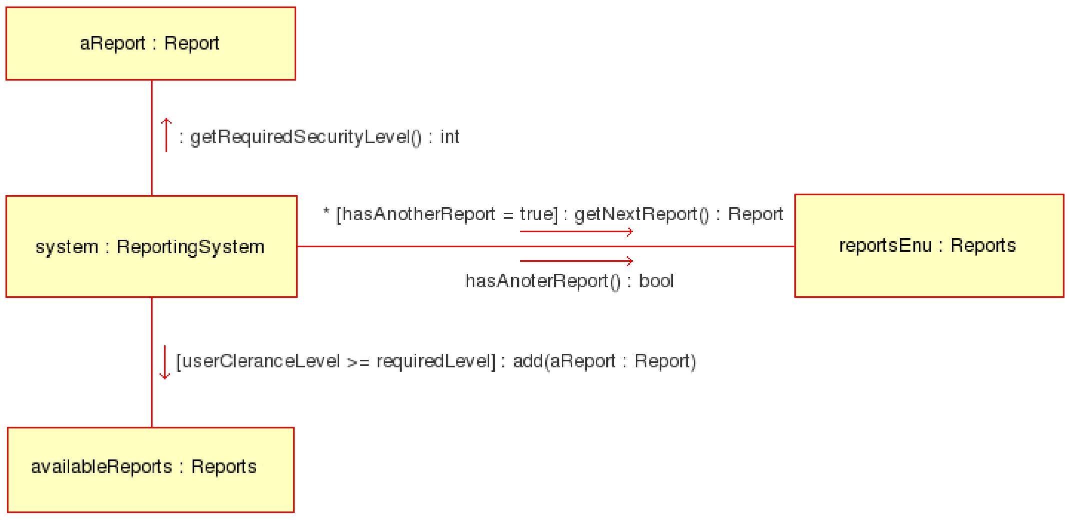
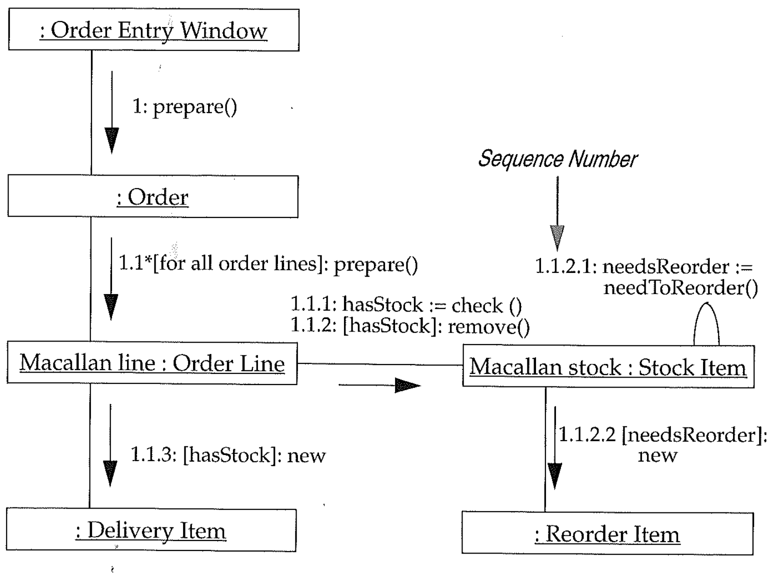

aReport : Report
: getRequiredSecurityLevel() : int
system : ReportingSystem
* [hasAnotherReport = true] : getNextReport() : Report
hasAnoterReport() : bool
reportsEnu : Reports
[userCleranceLevel >= requiredLevel] : add(aReport : Report)
availableReports : Reports

Figure 10: Get all reports and add the relevant ones to `availableReports`



: Order Entry Window
1: prepare()
: Order
Sequence Number
1.1*[for all order lines]: prepare()
1.1.2.1: needsReorder := needToReorder()
1.1.1: hasStock := check ()
1.1.2: [hasStock]: remove()
Macallan line : Order Line
Macallan stock : Stock Item
1.1.3: [hasStock]: new
1.1.2.2 [needsReorder]: new
: Delivery Item
: Reorder Item

Figure 11: Preparing an order

21.4.2 Preparing an order

The communication diagram in Figure 11 was taken form [1]. It is actually a Collaberation diagram in UML 1.2. It shows the communication between objects when executing the `prepare()` method in the `Order` class.

Note that the object names are underlined. This was required in UML 1.2 Collaboration diagrams. In UML 2.0 object names are no longer underlined.

In this diagram decimal numbering with sub-numbers to model the order in which the operations are executed.

The * in operation number 1.1 indicates that this operation is executed repeatedly. The guard condition for this repetition is given as a natural language expression. It indicates that this operation should be executed for all the order lines that exist in the system.

Operations number 1.1.2, 1.1.2.2 and 1.3 are conditional statements. The guards for all these conditional statements are boolean values. Operations number 1.1.3 and 1.1.2.2 are creational operations.

21.4.3 Sending mail to a mailing list

Figure 12 is a sequence diagram showing the actions of a portion of a system that creates a `Mailer` object and a `Mailing List` object. It then applies the Iterator design pattern to iterate through the items on the mailing list to send an email message to each of them.

Figure 13 is the communication diagram to model the communication between the objects in the same portion of the system that is modelled in Figure 12.

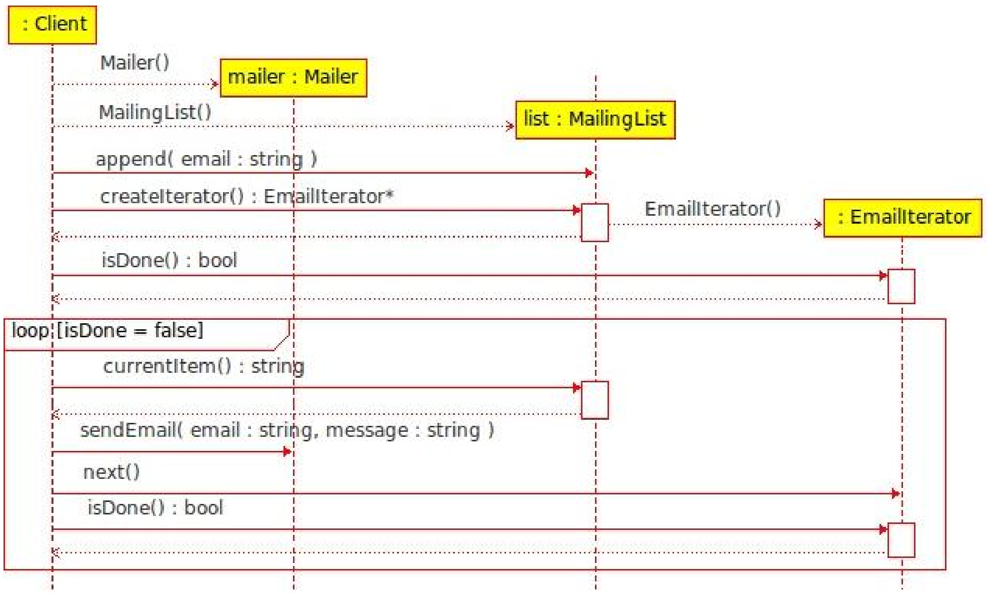


Figure 12: Sequence diagram

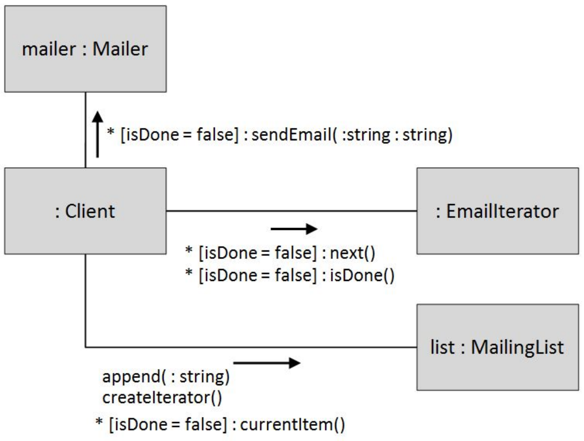

mailer : Mailer
* [isDone = false] : sendEmail(:string : string)
: Client
: EmailIterator
* [isDone = false] : next()
* [isDone = false] : isDone()
list : MailingList
append(: string)
createIterator()
* [isDone = false] : currentItem()

Figure 13: Communication diagram correlating with Figure 12

References

[1] Martin Fowler and Kendall Scott. *UML Distilled: Applying the Standard Object Modeling Language*. Addison-Wesley, Reading, Mass, 1997.

[2] Kendall Scott. *Fast Track UML 2.0*. Apress, Berkeley, CA, 2004.