Department of Computer Science

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Tackling Design Patterns
Chapter 13: UML Sequence diagrams

Contents

13.1 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 will look specifically at sequence diagrams. They are used to model how objects interact with one another in terms of the messages they pass to one another. While all interaction diagrams model these interactions, sequence diagrams emphasise the order of the messages over time.

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 sequence diagram can be used to model the order in which methods are executed as a reaction to some event.

13.2 Notational Elements

We show the notational elements of sequence diagrams in terms of the interaction between two simple classes in the class diagram shown in Figure 1. The Integer class is a wrapper for an integer value. Its getter simply returns its current value while its setter passes a double value which is rounded before it is converted to an integer. The constructor of these classes initialise their respective instance variables to 0. In the case of `Client`, its instance variables are initialised to NULL pointers.

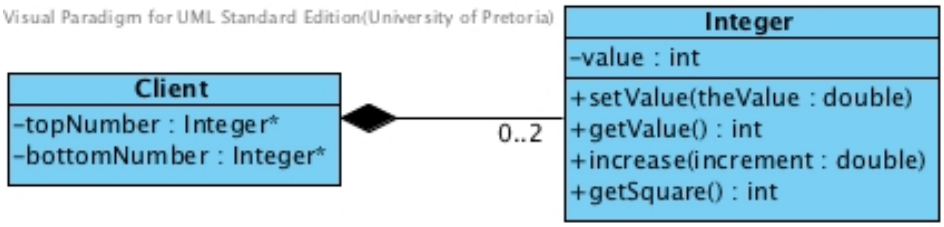

Figure 1: A class containing pointers to another class

13.2.1 Frames

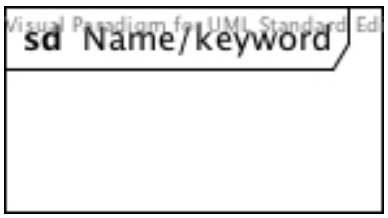

Figure 2: Sequence diagram frame

Sequence diagrams are drawn in frames. Figure 2 shows a frame of a sequence diagram. A frame is a rectangle with a heading in a compartment in the top left corner. The compartment for the heading is drawn as a rectangle with the lower right corner cut off.

The heading is used to name a diagram or to indicate the scope of loop structures (Section 13.4), conditional statements (Section 13.3) and parallel flows (Section 13.5) within a diagram. If it is used to name a diagram it is recommended that the name describe the essence of the interaction modeled in the diagram. When the frame is used within a diagram to show the scope of some subsection containing non-sequential flows, the heading should contain the appropriate keyword.

13.2.2 Lifelines

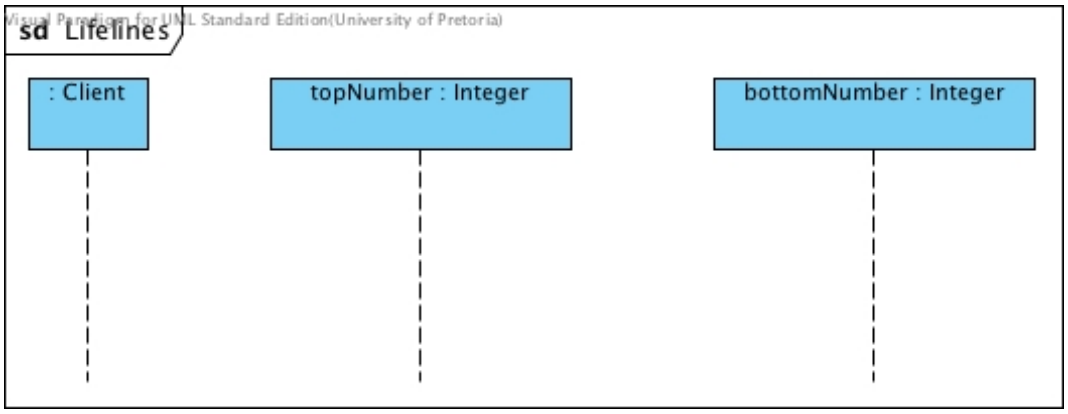

Figure 3: Lifeline notation

In sequence diagrams lifelines represent object instances. They are vertical lines inside the frame. Figure 3 shows three lifelines of objects that might be instantiated in our example. The rectangle at the top of each vertical dashed line identifies the participating object. The syntax is the same as for objects in object diagrams; an object name, a colon and a class name. In this figure the instance of the `Client` class is an anonymous object. Therefore it has only the class name in its identifier. Note that a colon precedes the class name in all cases.

The order of these lifelines is not significant, but by convention general flow starts at the leftmost lifeline and there is a general flow of messages across the diagram from left to right.

13.2.3 Creation and Destruction

It is important to realise that objects in a sequence diagram are *instantiated* instances of classes in a system. Upon creation of the `Client` class the pointers to `topNumber` and `bottomNumber` are initiated to NULL pointers. Therefore, the lifelines of these objects should appear only after they are created.

Assume the following is the main method executed by the `Client` application:

```
int main()
{
        topNumber = new Integer();
        delete topNumber;
        return 0;
}
```

Figure 4 shows how this program should be modeled. A creation action is shown with a dashed arrow with lined arrowhead to the identifier of a new lifeline. The arrow is labelled with the signature of the constructor that is called. Destruction is shown with an unlabeled arrow with filled head from the object that sends the destruction message to the point of destruction of the destructed object. The pont of destruction is shown with a heavy cross at the end of the lifeline.

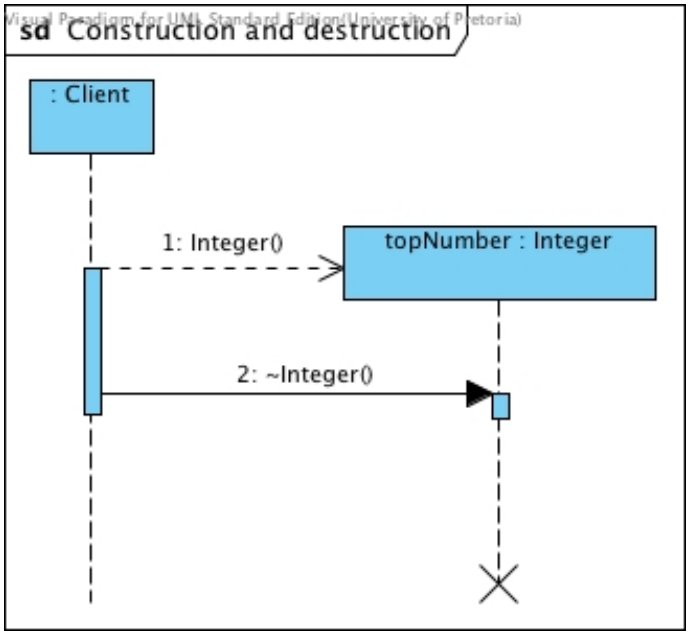

Figure 4: Construction and destruction

13.2.4 Messages

We will show simple messages that can be executed by instances of the two classes in the class diagram in Figure 1.

Assume the following program fragment executed by the `Client` application after `topValue` was created:

```
double aValue = 2.66;
topNumber->setValue(aValue);
```

Figure 5 shows how this interaction is modeled in a sequence diagram. The only interaction modeled here is the call to the Integer instance called `topNumber` using its setter. We assume that the creation and destruction time is not relevant in our model. This is

a simple method call with no return value. This notation should be used for all method calls using methods with void return values.

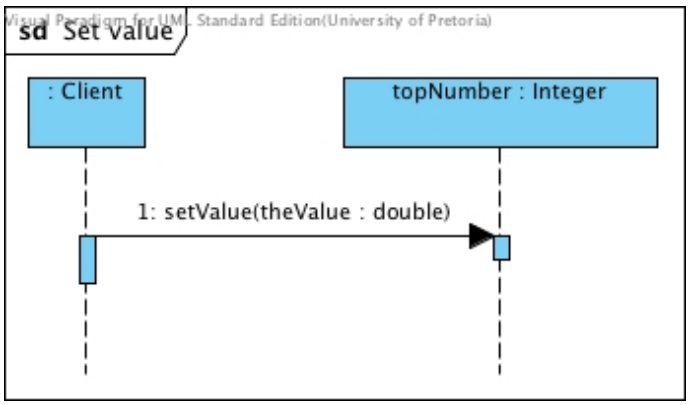

Figure 5: Message call without return value

Note that the label on the message in the diagram is the method signature and does not contain reference to the variable that was used when this message was sent.

Next assume the followingline of code is executed by the `Client` application:

```
int aValue = bottomNumber.getValue();
```

Figure 6 shows how the interaction of this line of code is modeled in a sequence diagram under the assumption that `bottomNumber` was properly instantiated. The interaction that is modeled here is a call to the Integer instance called `bottomNumber` using its getter. This is a method call with a return value. The return of a value is shown using a dashed arrow. This notation should be used for all method calls using methods with return values.

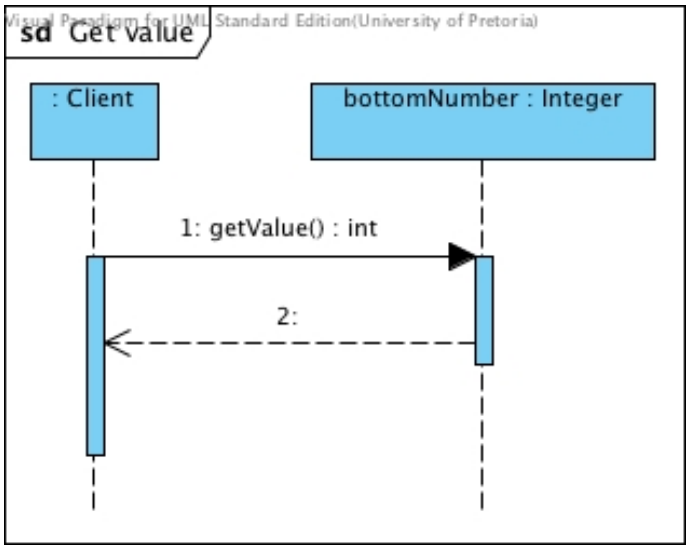

Figure 6: Message call with a return value

13.2.5 Reflexive messages

A reflexive message, referred to as *self message* in Visual Paradigm, is when an object calls a method that is defined in its own class. Typically methods with private scope can only be called by objects that are instances of the class that implements the method. This is the case with the `increase(:double)` and `getSquare():int` methods defined in the `Integer` class in Figure 1. `getSquare():int` has an `int` return value while the return value of `increase(:double)` is `void`. Figure 7 illustrates how these method calls are modelled.

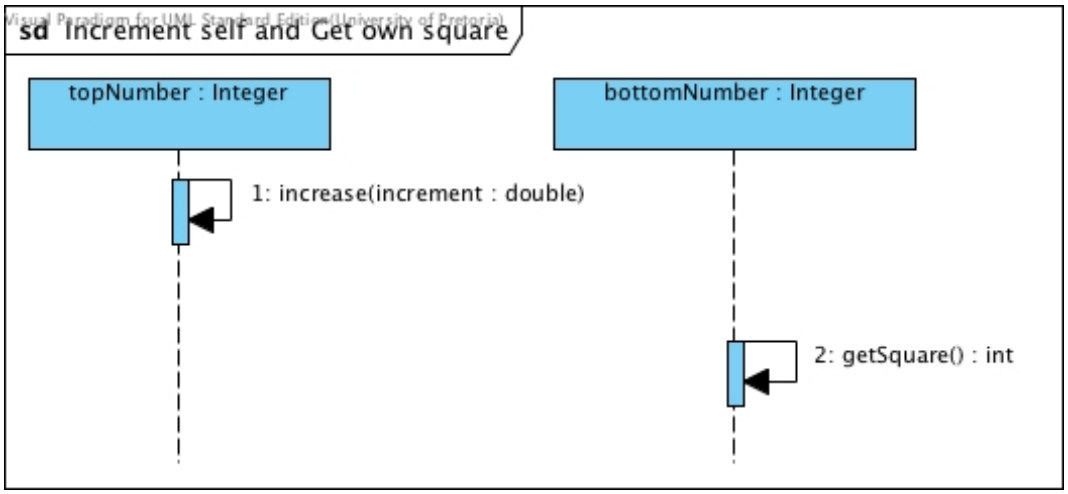

Figure 7: Reflexive message call without a return value and with a return value

13.2.6 Example

We will show an example of interaction between instances of the classes in the class diagram in Figure 8. This application implements the Strategy design pattern. These classes are used in a simple text-based two-player game where the user can select the strategy for his dragon and attack his opponent's dragon.

The modeling of interaction in the system requires knowledge of the implementation of the methods involved in the interaction. The following code shows the implementation of the methods we model in this example:

```
void Dragon :: attack(Dragon* enemy)
{
    strategy->fight(enemy, attackPower);
    float enemyPower = enemy->getAttackPower();
    lifeForce -= strategy->getRecoil(lifeForce, enemyPower);
}

void Dragon :: receiveInjury(float enemyPower)
{
    attackPower
```

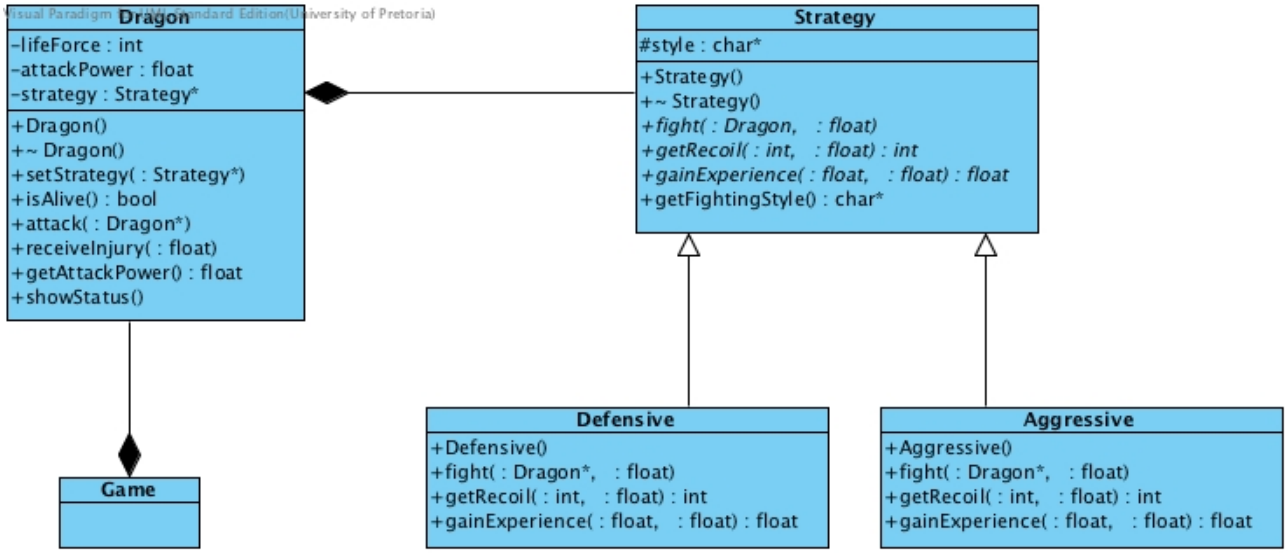

Figure 8: An application of the Strategy Design Pattern

```
    = strategy->gainExperience(attackPower, enemyPower);
}


void Aggressive::fight(Dragon* d, float ownPower)
{
    int num = rand();
    d->receiveInjury(ownPower * (num % 2 + 2));
}

float Defensive::
gainExperience(float ownPower, float enemyPower)
{
    return (ownPower + enemyPower/2);
}
```

Assume the that two dragons (`norbert` and `smaug`) have been instantiated. Further assume that the following statements have already been executed:

```
norbert->setStrategy(new Aggressive());
smaug->setStrategy(new Defensive());
```

Figure 9 shows how the interaction resulting from the following statement is modeled in a sequence diagram.

```
norbert->attack(smaug);
```

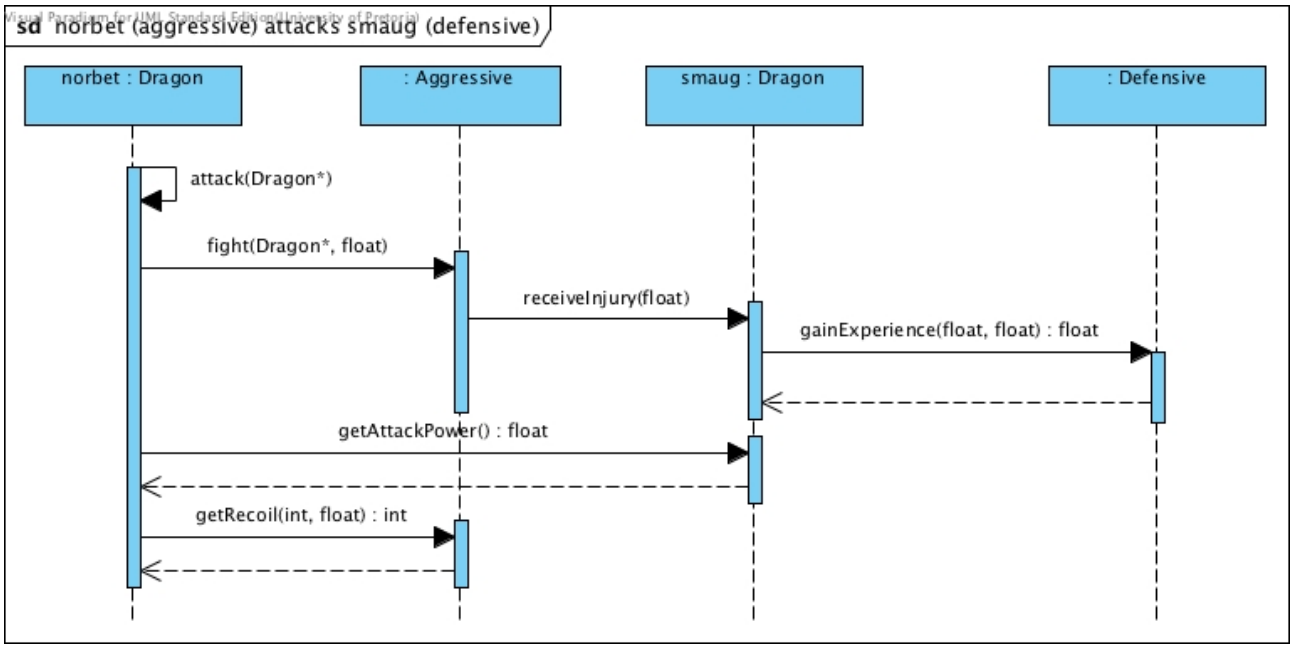

Figure 9: Example of interaction between multiple objects

13.3 Branching

13.3.1 Notation

Branching happens when the program flow contains conditional statements. Figure 10 shows the notation to model an alternate structure. A typical `if` or `switch` statement is modeled by showing the alternative interactions in different sections in a frame with the keyword `alt` in its header. The different sections are separated with dotted lines. The condition that needs to be true for a section to execute is shown as a guard in the top left corner of each section. One may have as many sections as needed. If there is only one section, the frame may alternatively be labeled with the keyword `opt`.

13.3.2 Example

Figure 11 models the interaction for an application for which a `ConnectionController` object has to open up a connection to a `Modem` object. This example was adapted from [2]. If the modem does not respond within a certain amount of time signaled by a `Timer` object, the operation times out and an error dialogue box is created and displayed. If the modem responds, the timer is cancelled and the modem is initialised.

13.4 Iteration

13.4.1 Notation

Iteration happens when the program flow contains looping statements. Figure 12 shows the notation to model recurrence. A typical `for` or `while` statement is modeled by showing

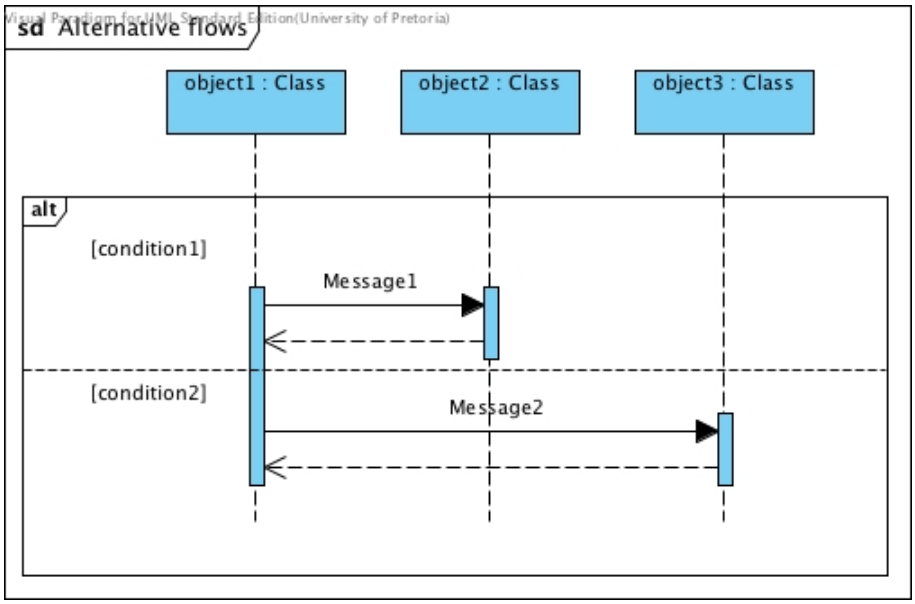

Figure 10: Syntax for alternate flows

the repeated interactions in a frame with the keyword `loop` in its header. The condition that needs to be true for the interactions n the frame to be executed is shown as a guard in the top left corner of the frame. In this example `message 1` returns its result only after `message 2` had completed all its iterations.

13.4.2 Example

The sequence diagram in Figure 13 contains a loop fragment. This example is taken from [1]. The guard in the loop fragment tests to see if the value of `hasAnotherReport` equals true. If the hasAnotherReport value equals true, then the sequence goes into the loop fragment, else the flow proceeds to the first event after the loop. In this case it is the termination of the `getAvailableReports()` message execution. The content of the loop fragment is interpreted the same way one would follow the messages in the loop as you would normally in a sequence diagram.

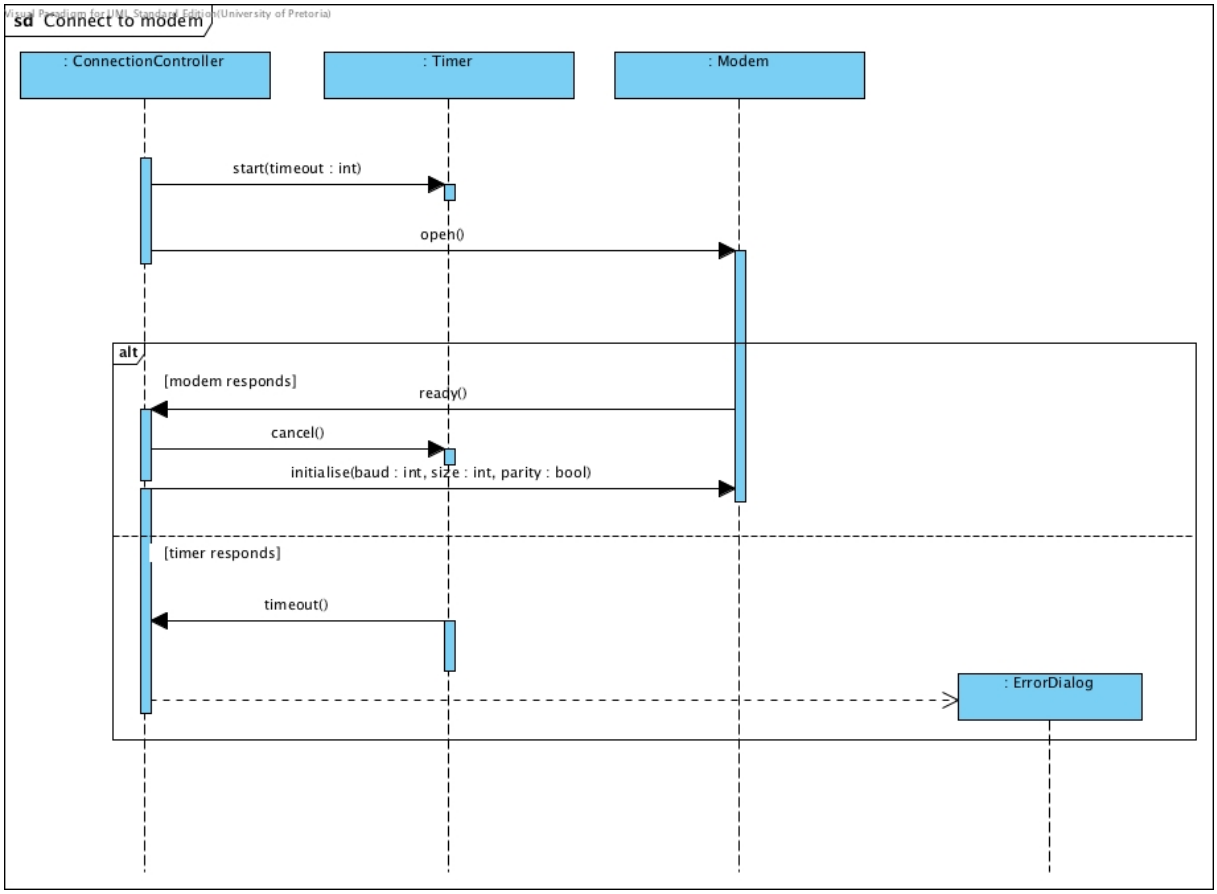

Figure 11: Connection to a modem with alternate flows

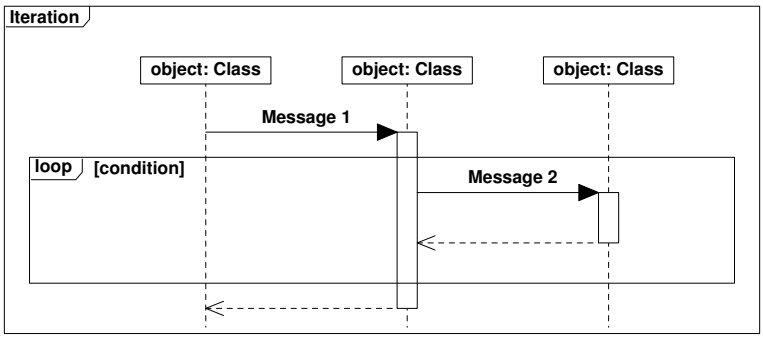

Figure 12: Syntax for a loop structure

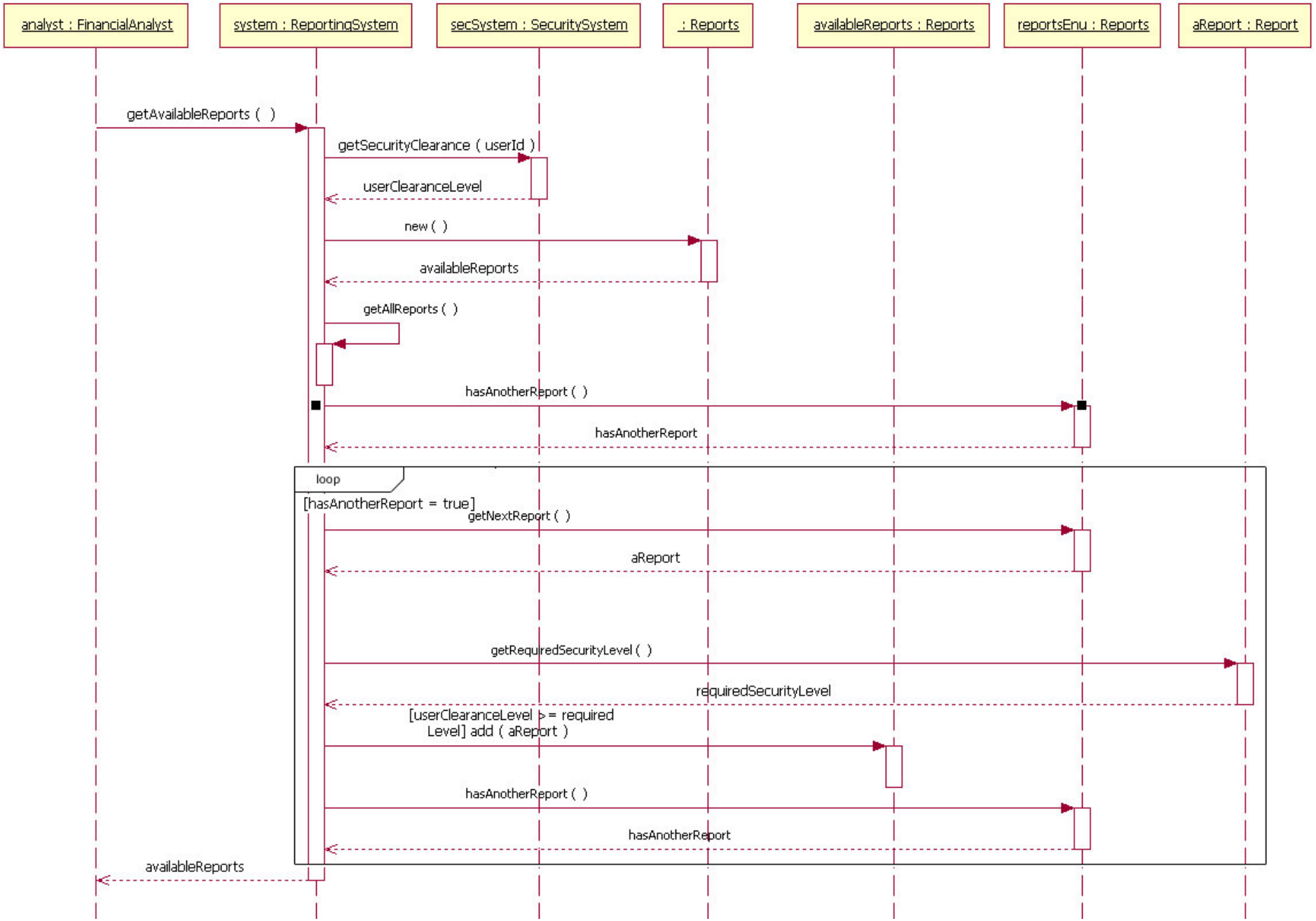

Figure 13: Get all reports

13.5 Parallel actions

13.5.1 Notation

It is also possible to model interactions that are executed at the same time in parallel. Figure 14 shows the notation to model this. It is modeled by showing the parallel interactions in different sections in a frame with the keyword `par` in its header. The different sections are separated with dotted lines. In this case all the sections will be executed at the same time and execution of the interactions after the parallel fragment commences after all the parallel threads has completed their actions. One may have as many sections as needed.

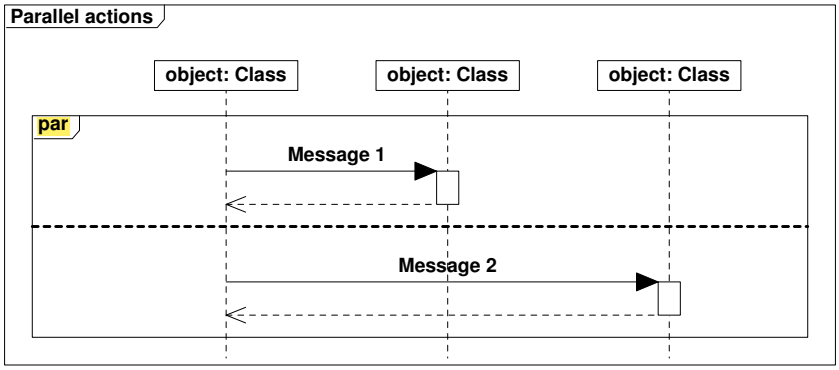

Figure 14: Syntax for parallel actions

13.5.2 Example

The sequence diagram in Figure 15 contains concurrent events as well as a condtional statement. This example was adapted from [3]. When a transaction is created, it creates a transaction coordinator to coordinate the checking of the transaction. This coordinator creates a number (in this case, two) of transaction checker objects, each of which is responsible for a particular check. These checkers are called asynchronously and proceeds in parallel. Thus, the top of the two subframes contains parallel flows.

When a transaction checker completes, it notifies the transaction coordinator about its success whereafter the controller destroys it. The coordinator continuously looks if checkers calls back and will only continue with operation once all the transaction checkers have reported back. This concludes the parallel flows.

The conditional flow is selected after the coordinator have executed a reflexive check to determine if all the checkers were successful. If so, it reports success to the transaction, else it reports failure. After completion of the conditional statement, the transaction destroys the coordinator.

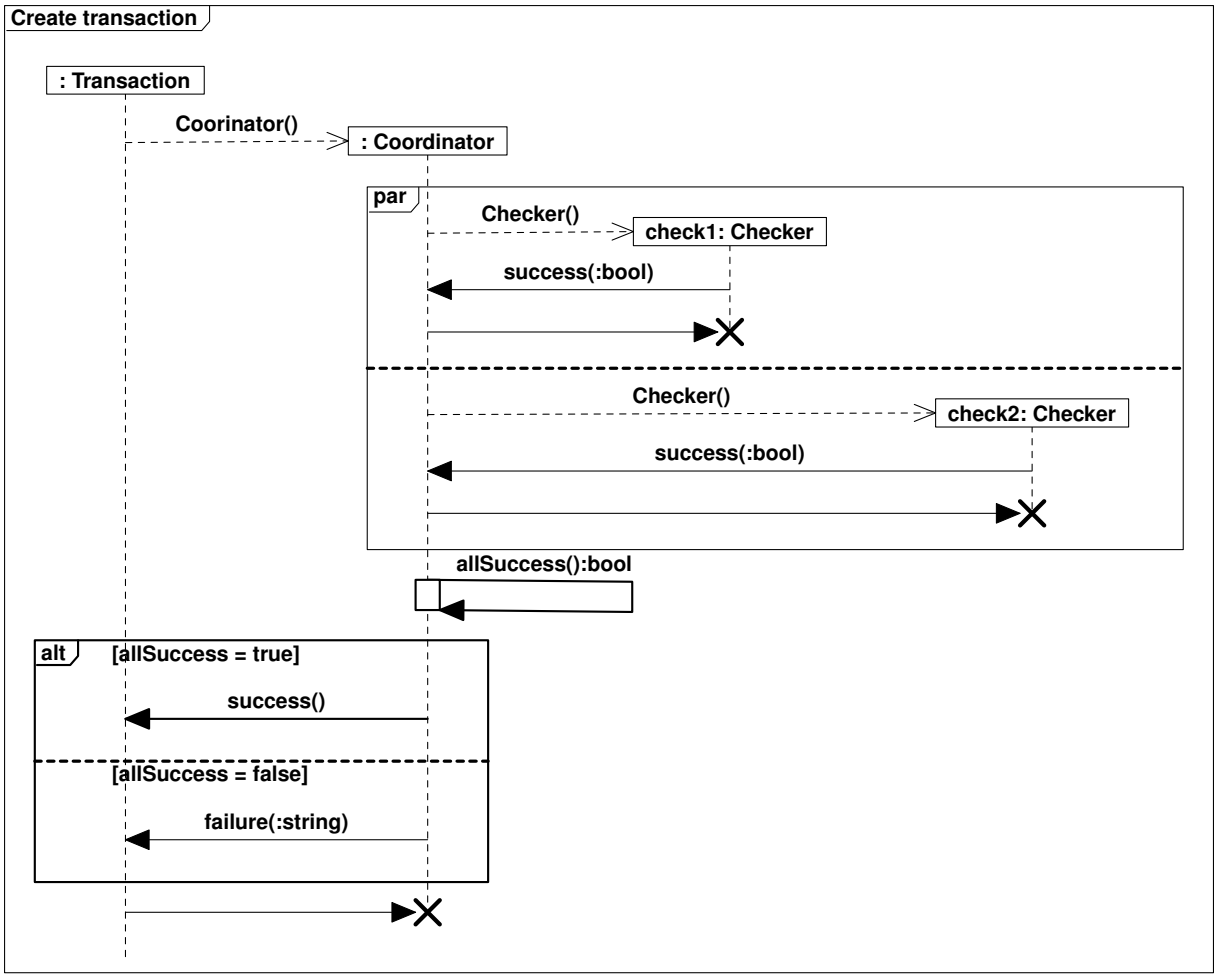

Figure 15: Creating a banking transaction

13.6 Reference to fragments

If a diagram becomes complex, it it advisable to model it in fragments. A fragment is a sub-diagram. A placeholder for a fragment is shown in a diagram using a frame with the keyword `ref` in its header. This placeholder contains only the name of the diagram that contains the detail of the fragment. Figure 16 models the same application as in Section 13.3.2 but by defining parts of the diagram in fragments.

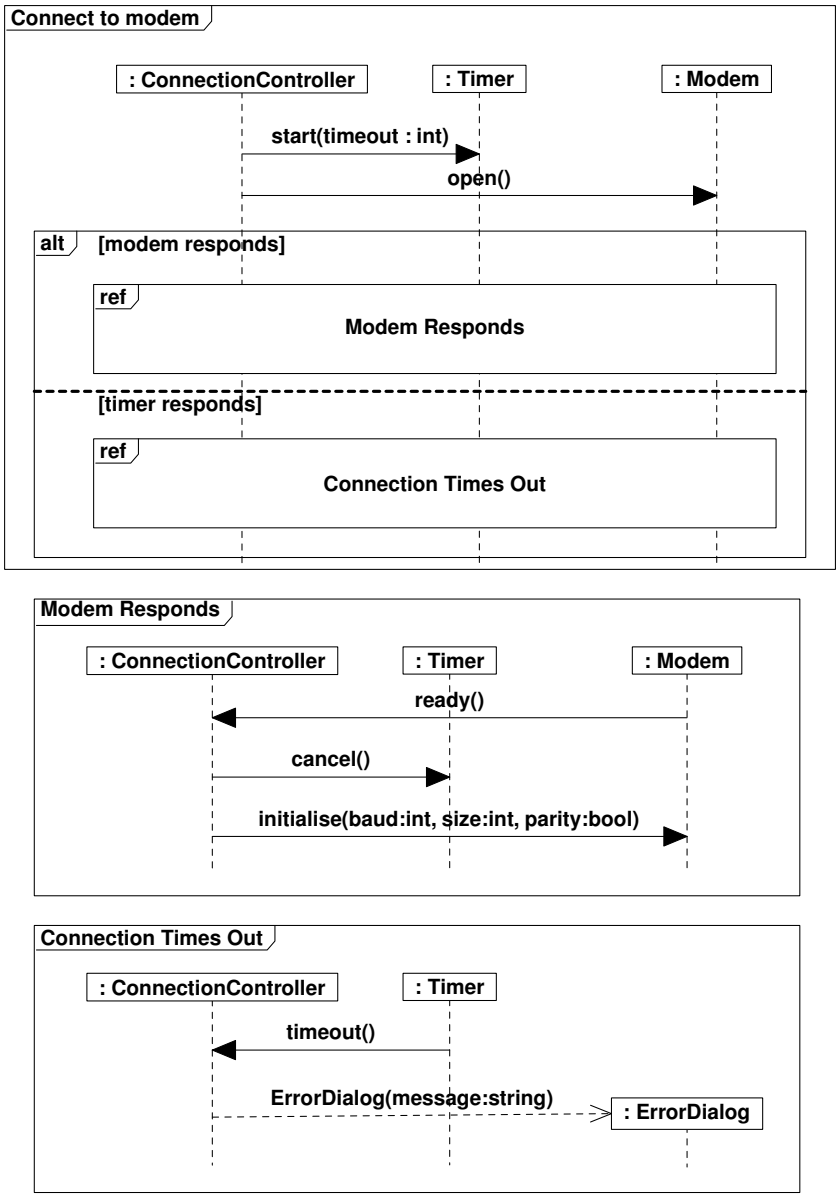

Figure 16: Connection to a modem with alternate flows presented in fragments

13.7 Exercises

1. Use the code given in Section 13.2.6 and the implementations of other relevant methods shown in the class diagram in Figure 8 that can be found in the code that was given for Prac 4. Assume the that two dragons (`norbert` and `smaug`) have been instantiated. Model the interaction resulting from the following code fragment in the `Game` class.

```
norbert->setStrategy(new Aggressive());
smaug->setStrategy(new Defensive());
smaug->attack(norbert);
```

References

[1] Donald Bell. Uml basics: The sequence diagram. `http://www.ibm.com/developerworks/rational/library/3101.html`, 2004. [Online] accessed 2011/08/22.

[2] Simon Bennett, John Skelton, and Ken Lunn. *Schaum's Outline of UML*. McGraw-Hill Professional, UK, 2001. ISBN 0077096738.

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