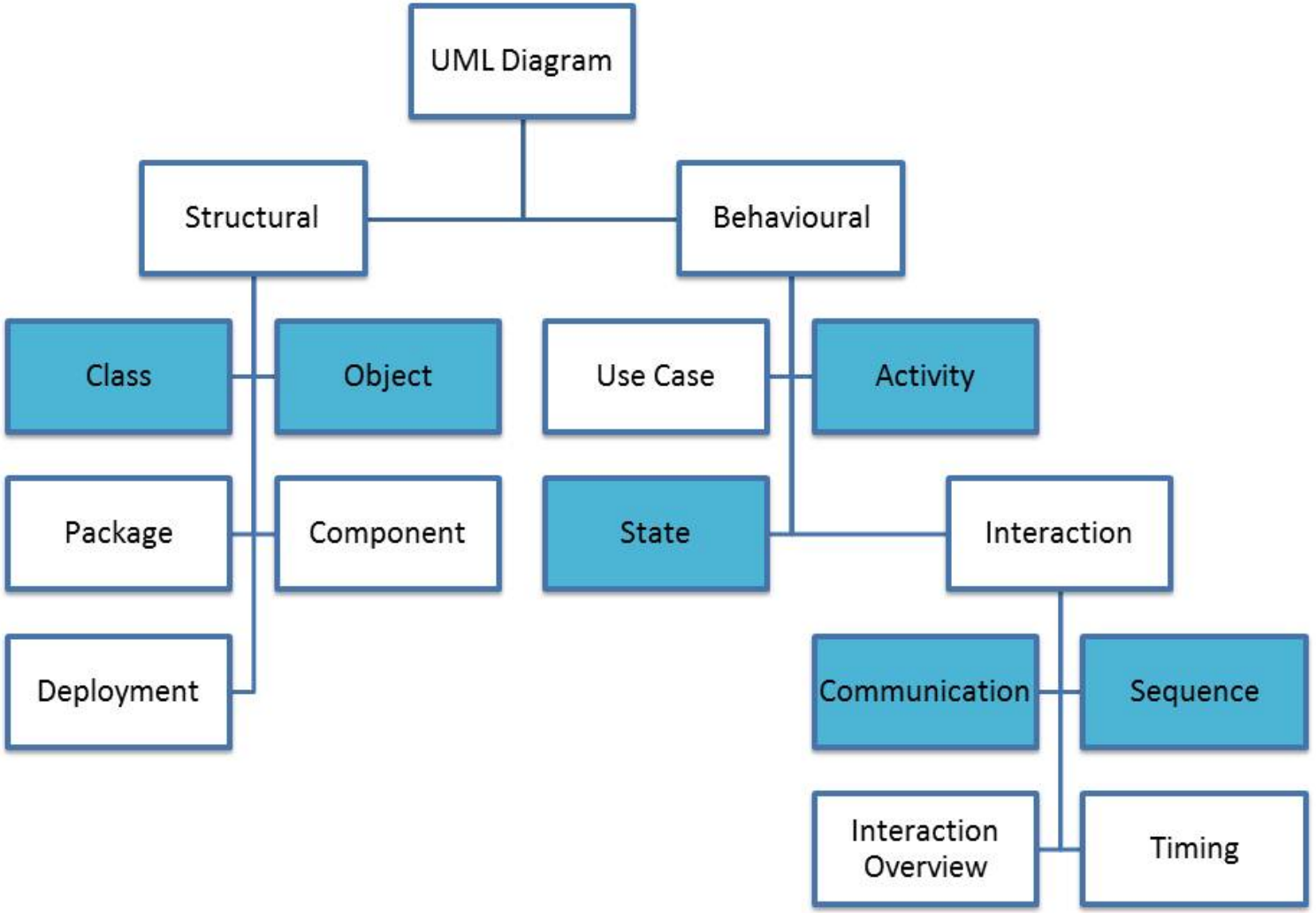
UML Diagrams

Sequence Diagram

Linda Marshall

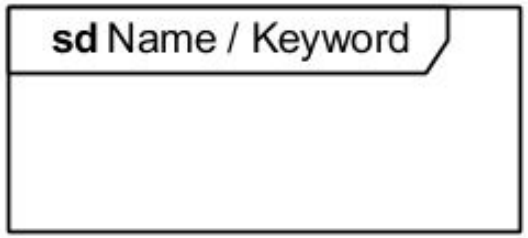
16 September 2021


UML Diagram
Structural
Behavioural
Class
Object
Use Case
Activity
Package
Component
State
Interaction
Deployment
Communication
Sequence
Interaction Overview
Timing

A sequence diagram is:

- used to model how objects interact with one another in terms of the messages (method calls) they pass to one another.
- emphasise the order of message execution as a reaction to some event.
- interactions are arranged from top to bottom, following their order of occurrence.

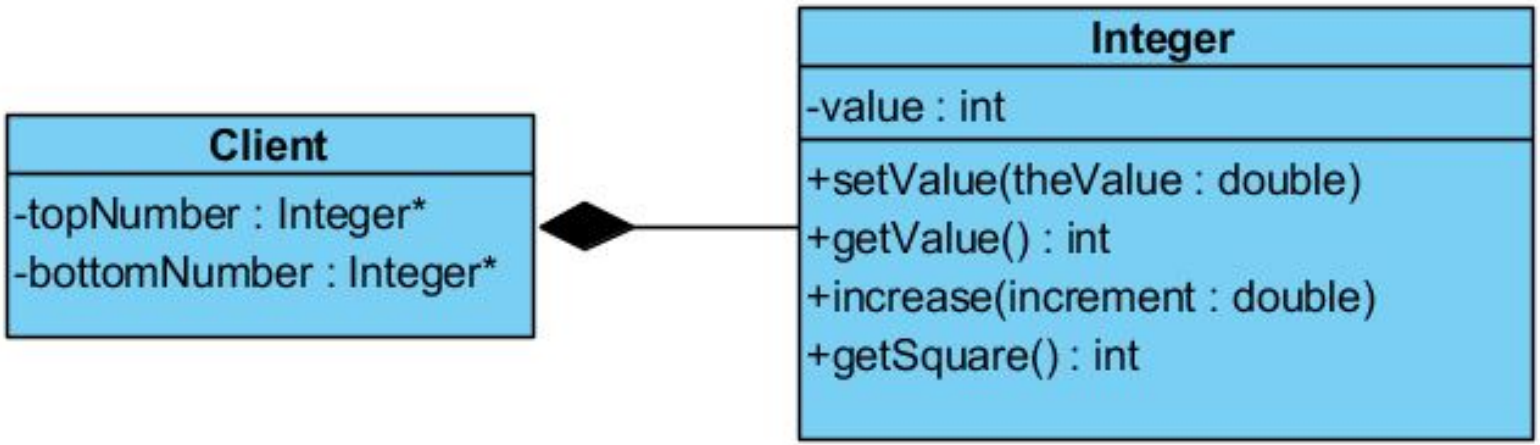
Sequence diagrams are drawn in *frames* - a rectangle with a heading



- *Name* - to name the diagram or
- *Keyword* - to indicate the scope of loop structures, conditional statements or parallel flows.

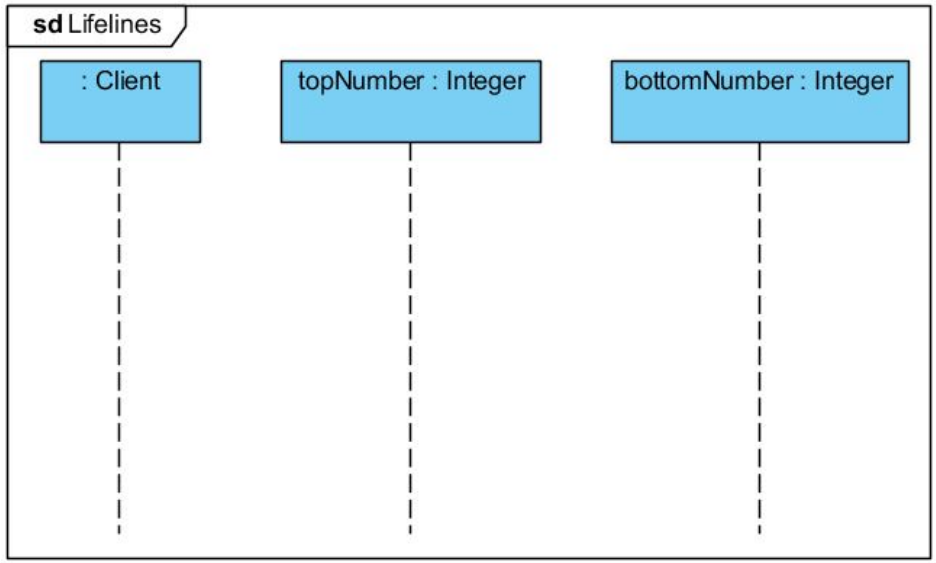
A *lifeline* represents an individual participant in the interaction (object instances).

The Integer class is a wrapper for an integer value

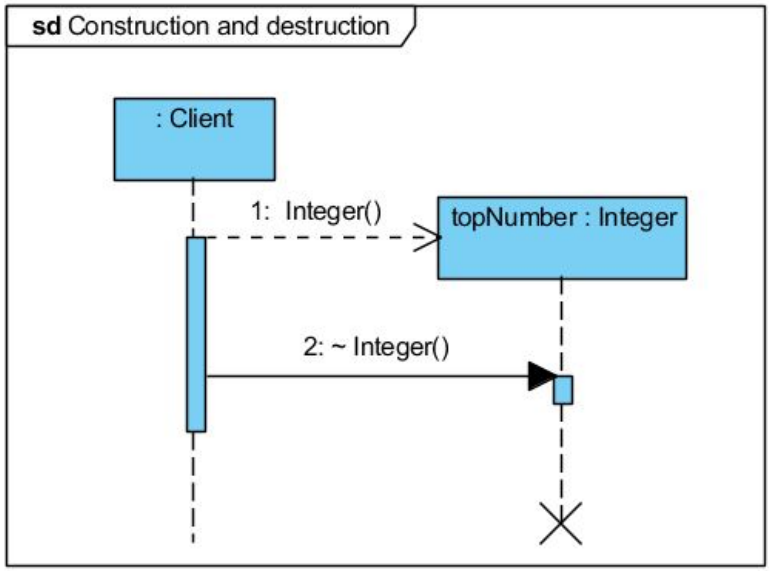



```
Integer* topNumber = new Integer();
Integer bottomNumber;
```

Objects in the system form the lifelines, the order of these lifelines is not significant.

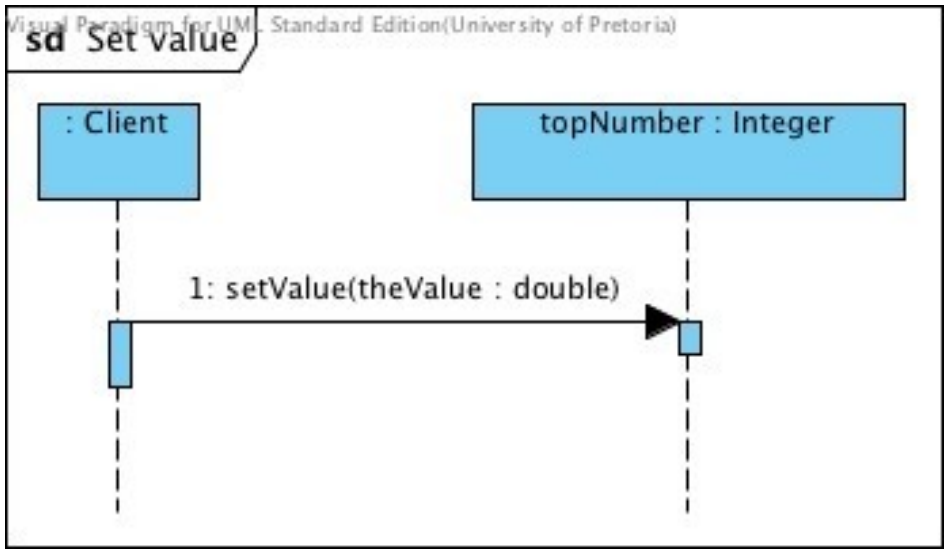


Creation and Destruction



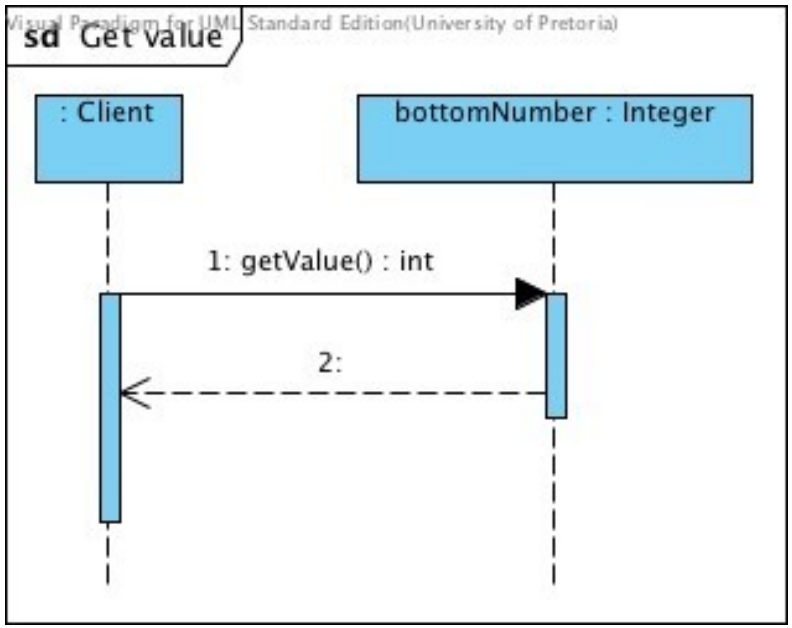

```
Integer* topNumber = new Integer();
delete topNumber;
```

Aynchronous - e.g. set value



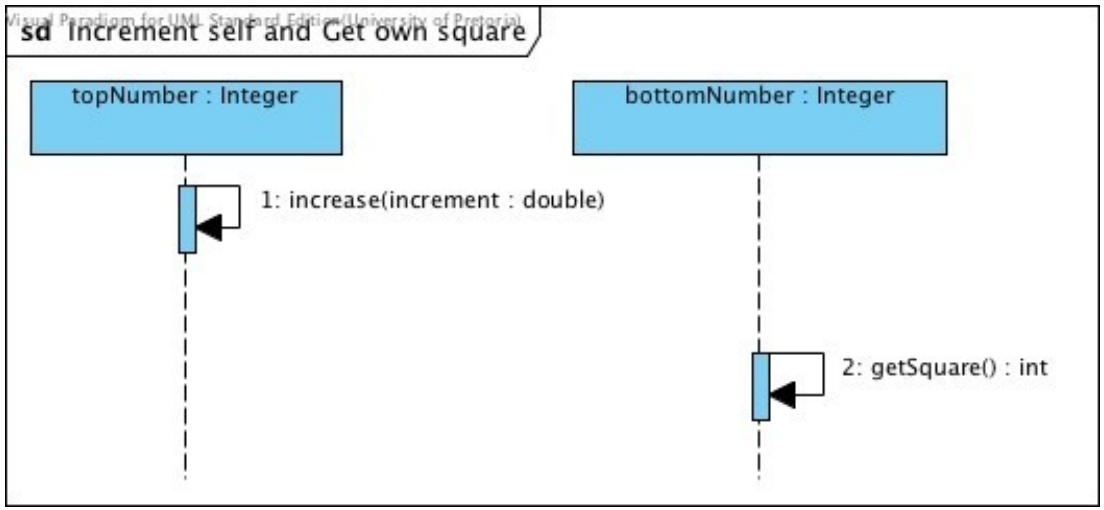

```
double aValue = 2.66;
topNumber->setValue(aValue);
```

Synchornous - e.g. get value



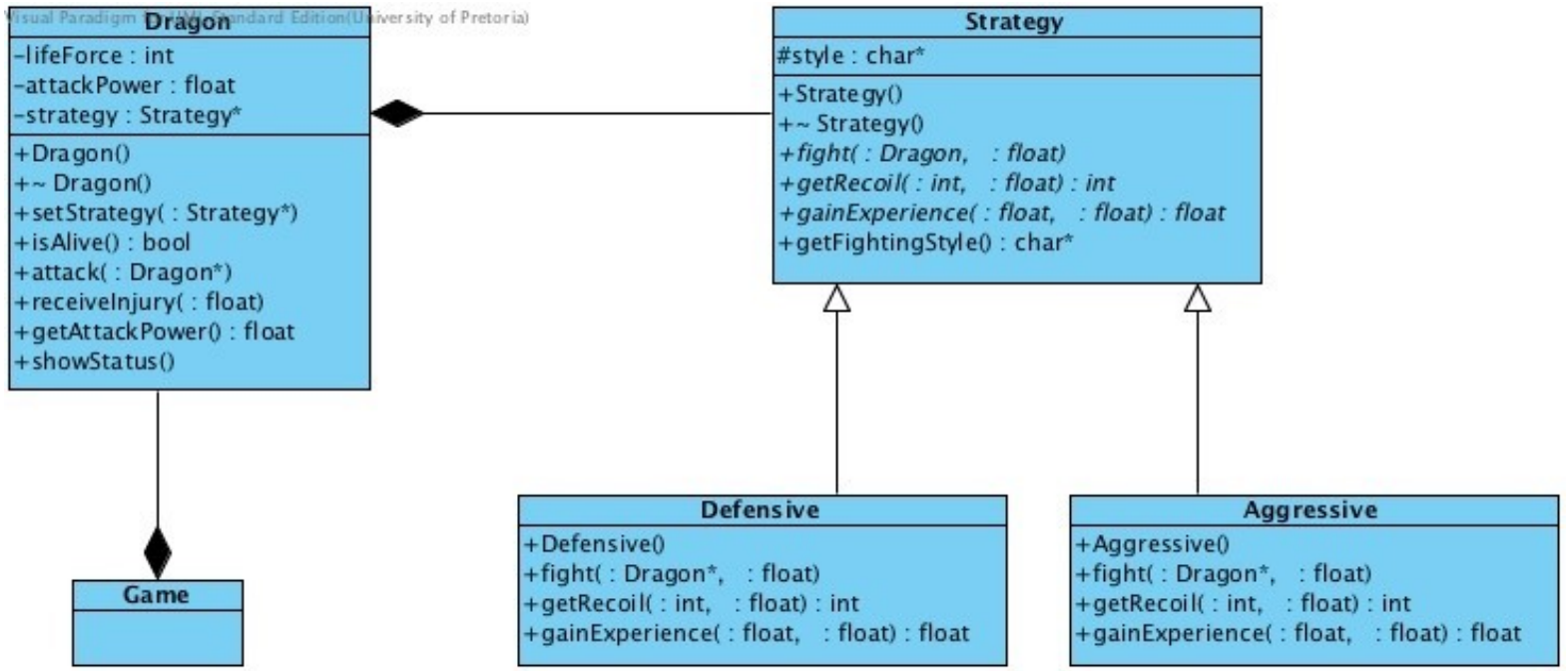

```
int   aValue = bottomNumber.getValue();
```

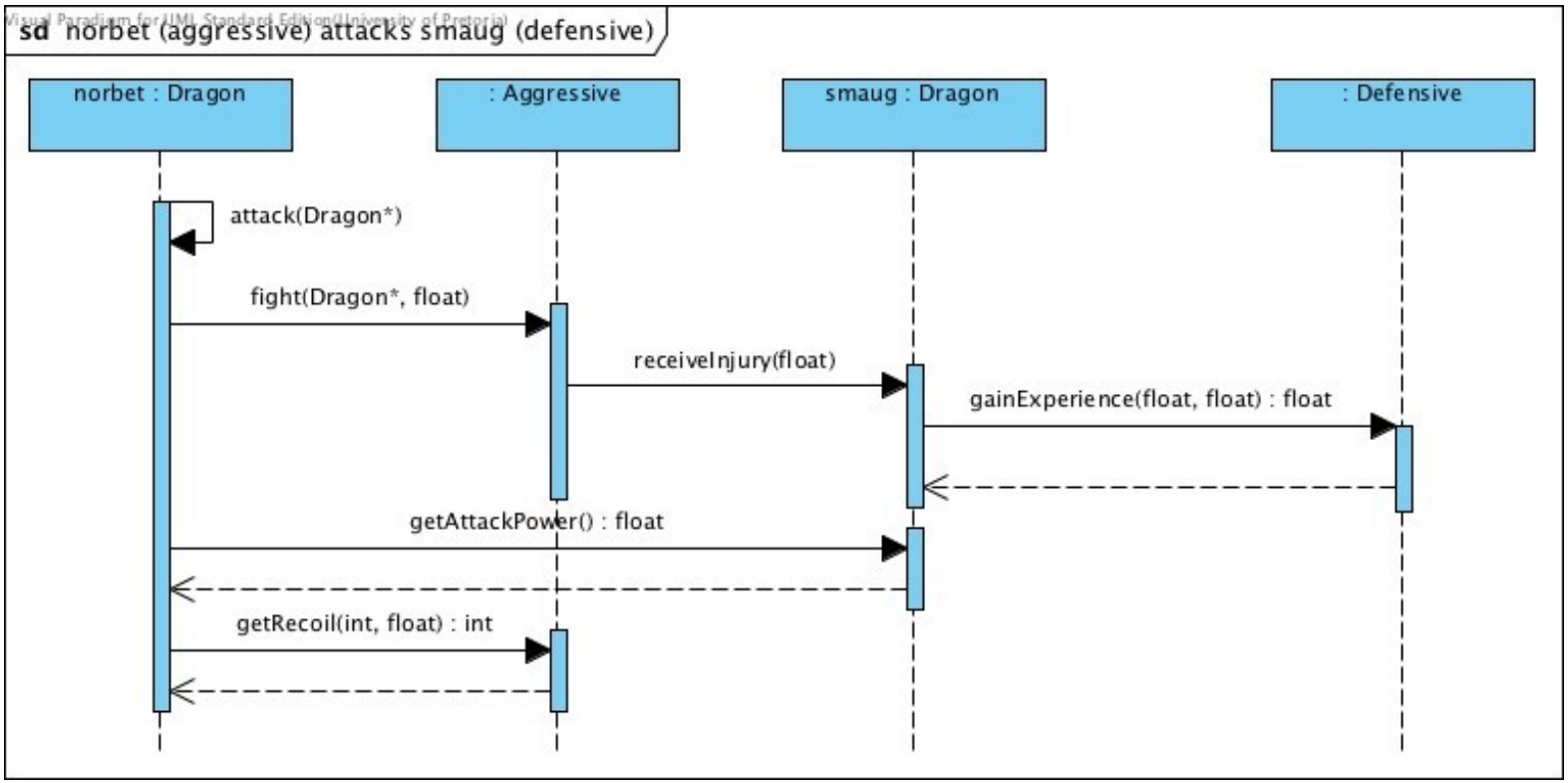
Reflexive messages (self message), when an object calls a method that is defined in its own class.



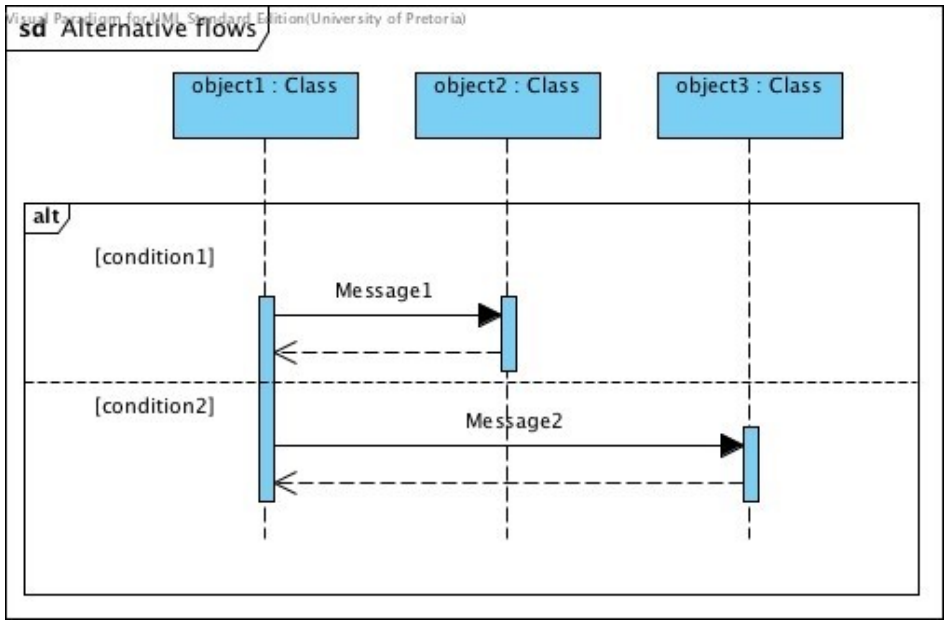

```
topNumber->increase(2);
cout << bottomNumber.getSquare();
```

Example

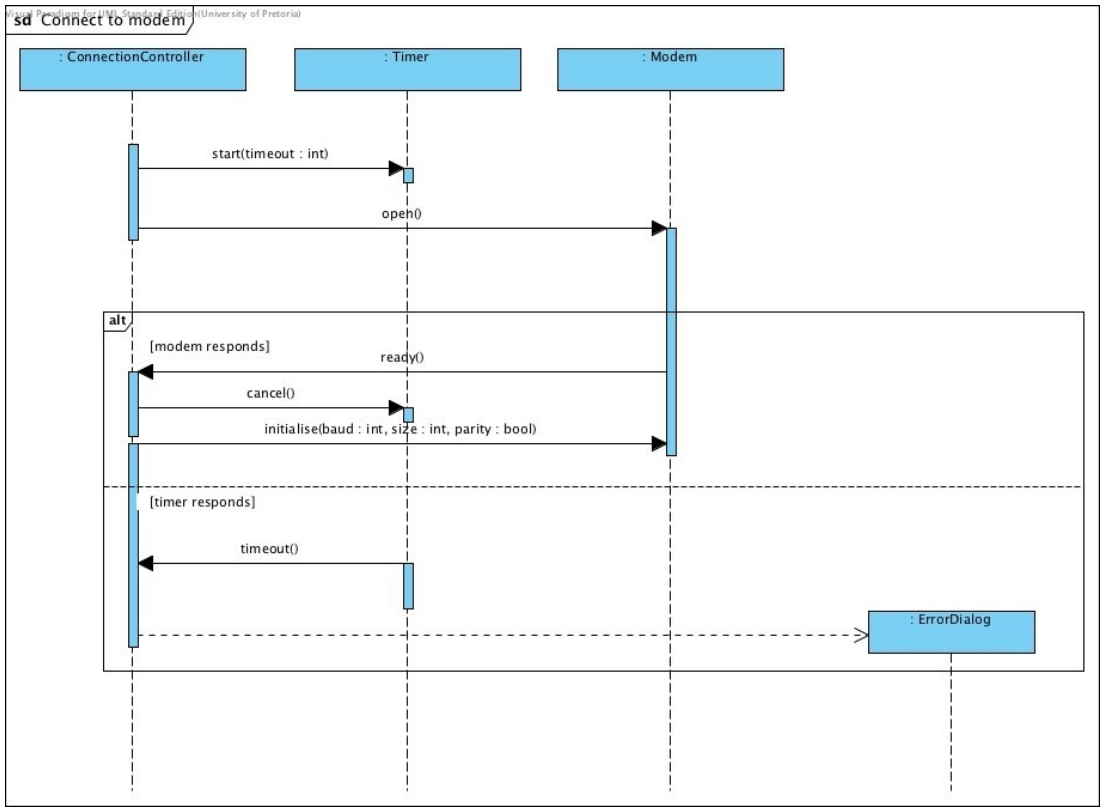

Dragon
-lifeForce : int
-attackPower : float
-strategy : Strategy*
+Dragon()
+~ Dragon()
+setStrategy(: Strategy*)
+isAlive() : bool
+attack(: Dragon*)
+receiveInjury(: float)
+getAttackPower() : float
+showStatus()
Strategy
#style : char*
+Strategy()
+~ Strategy()
+fight(: Dragon, : float)
+getRecoil(: int, : float) : int
+gainExperience(: float, : float) : float
+getFightingStyle() : char*
Game
Defensive
+Defensive()
+fight(: Dragon*, : float)
+getRecoil(: int, : float) : int
+gainExperience(: float, : float) : float
Aggressive
+Aggressive()
+fight(: Dragon*, : float)
+getRecoil(: int, : float) : int
+gainExperience(: float, : float) : float


sd norbet (aggressive) attacks smaug (defensive)
norbet : Dragon
: Aggressive
smaug : Dragon
: Defensive
attack(Dragon*)
fight(Dragon*, float)
receiveInjury(float)
gainExperience(float, float) : float
getAttackPower() : float
getRecoil(int, float) : int

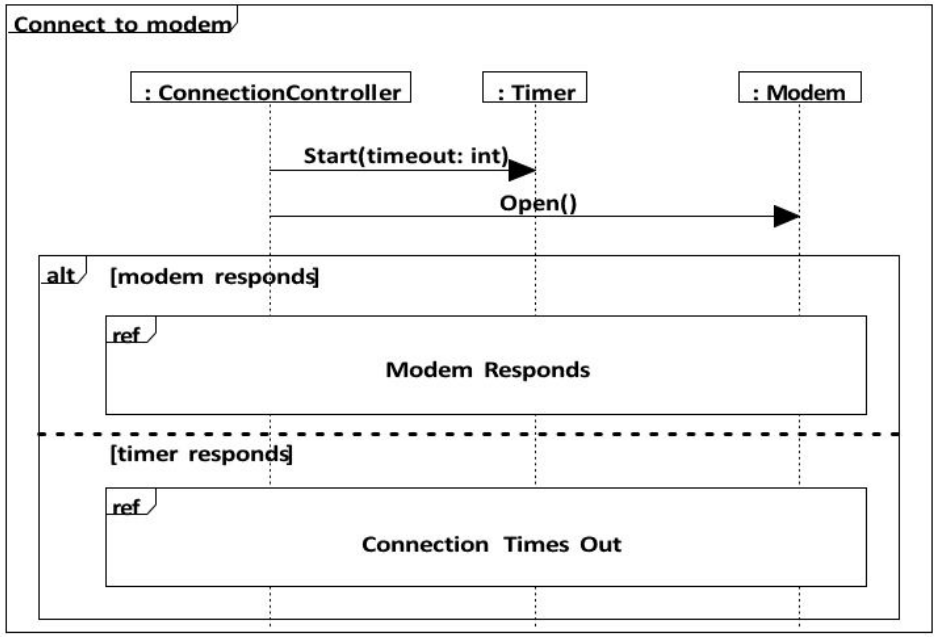
Branching happens when the program flow contains conditional statements.

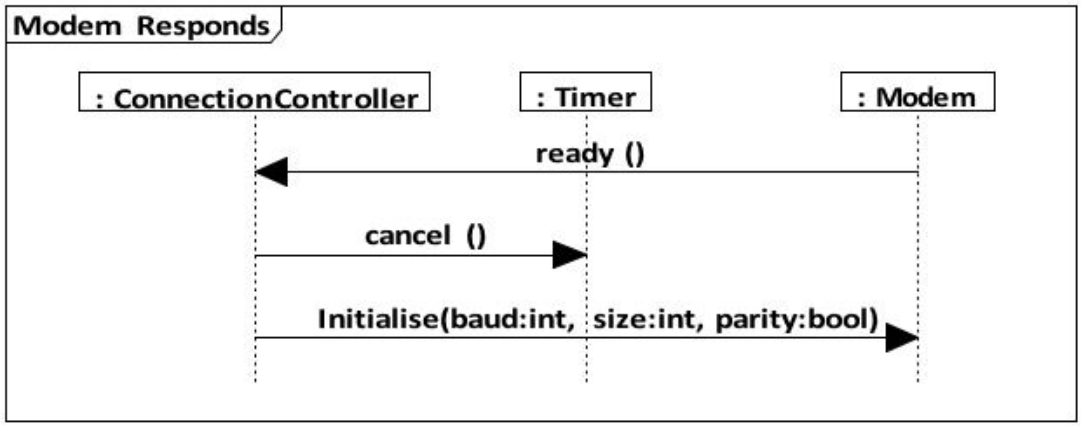


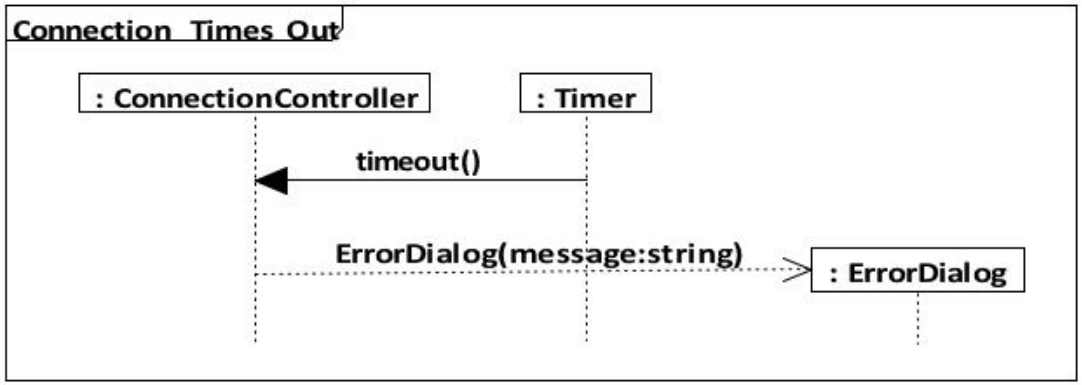
For example, connection to a modem



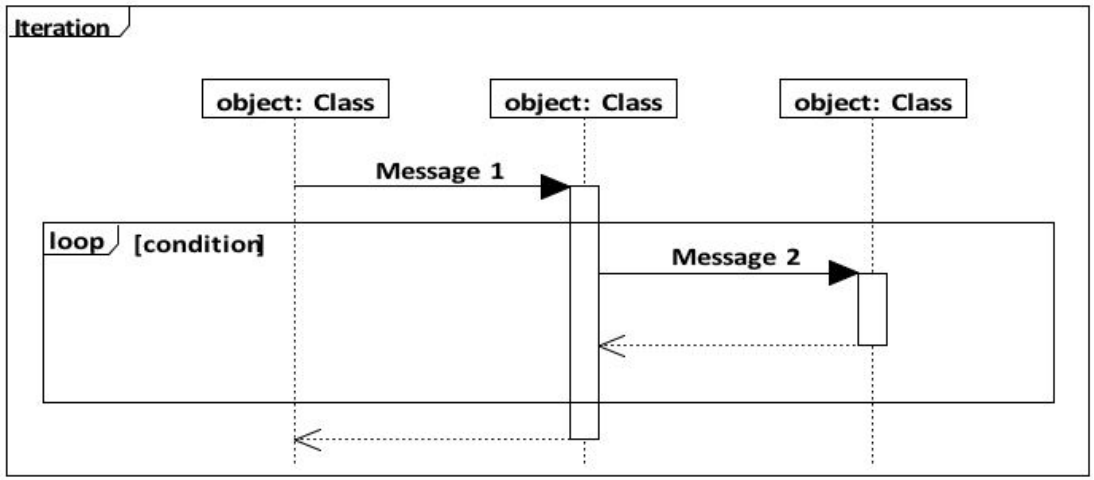
If a diagram becomes complex, it is advisable to model it in **fragments**.




Modem Responds
: ConnectionController
: Timer
: Modem
ready ()
cancel ()
Initialise(baud:int, size:int, parity:bool)

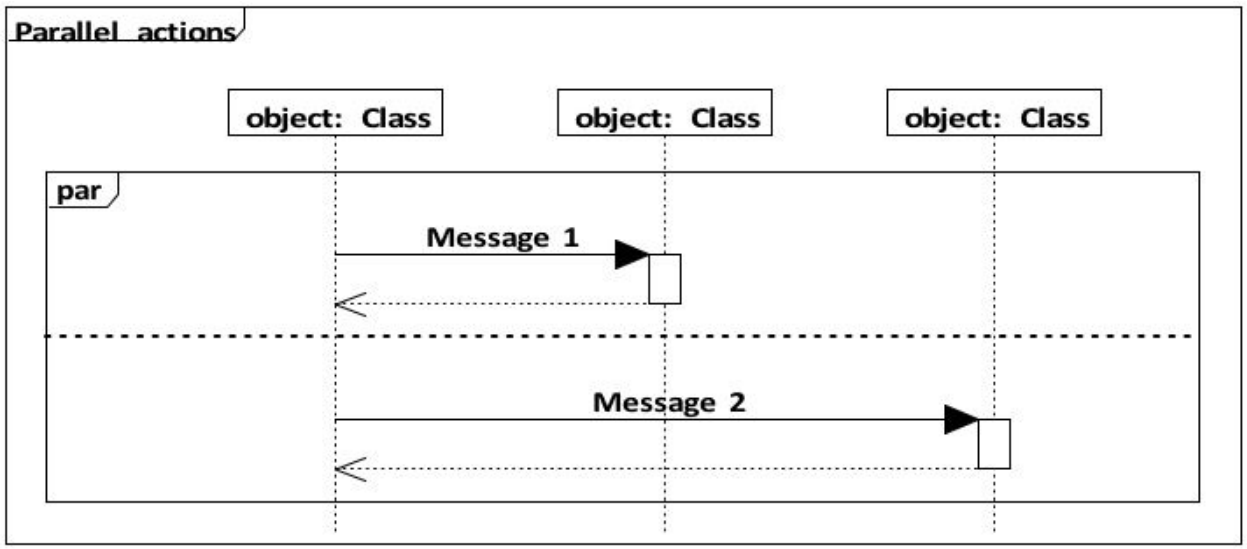

Connection Times Out
: ConnectionController
: Timer
timeout()
ErrorDialog(message:string)
: ErrorDialog

Iteration happens when the program flow contains looping statements.

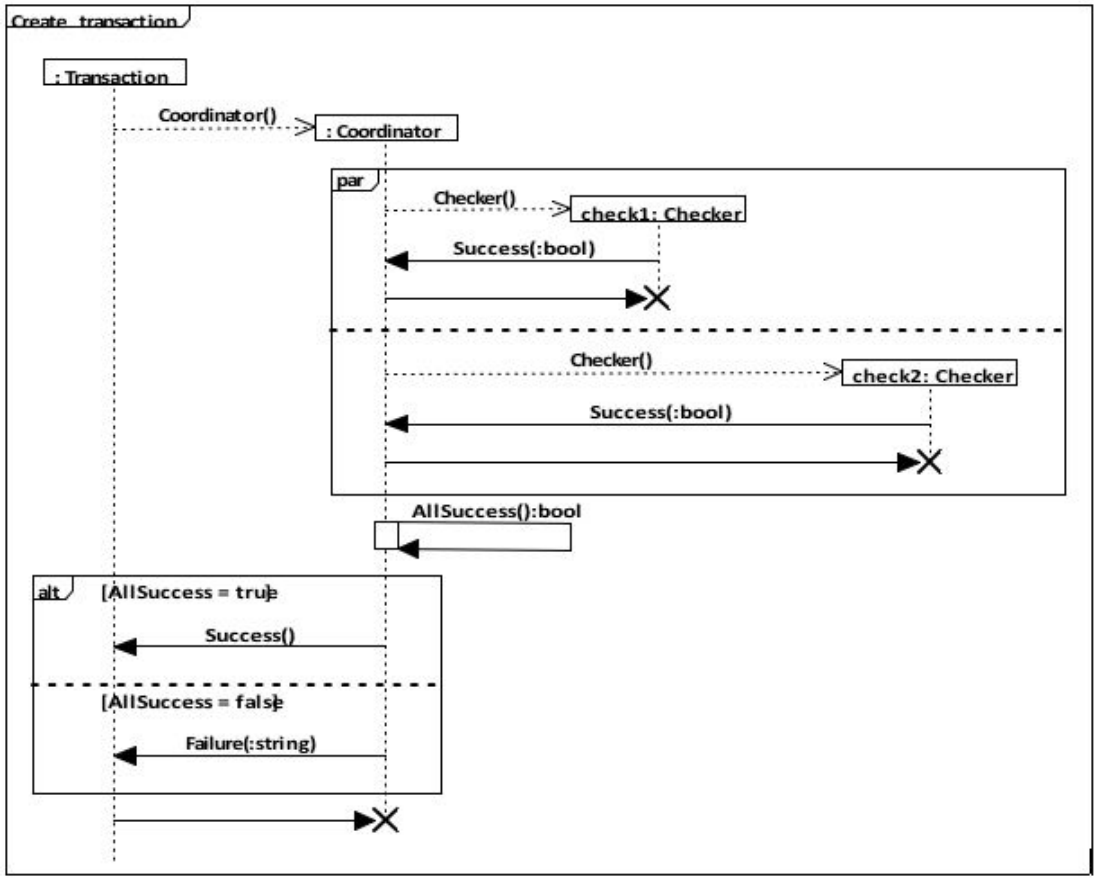


Syntax for a loop structure

Parallel actions model interactions that are executed at the same time (in parallel).



Syntax for parallel actions


Create_transaction
: Transaction
Coordinator()
: Coordinator
par
Checker()
check1: Checker
Success(:bool)
Checker()
check2: Checker
Success(:bool)
AllSuccess():bool
alt
[AllSuccess = true]
Success()
[AllSuccess = false]
Failure(:string)