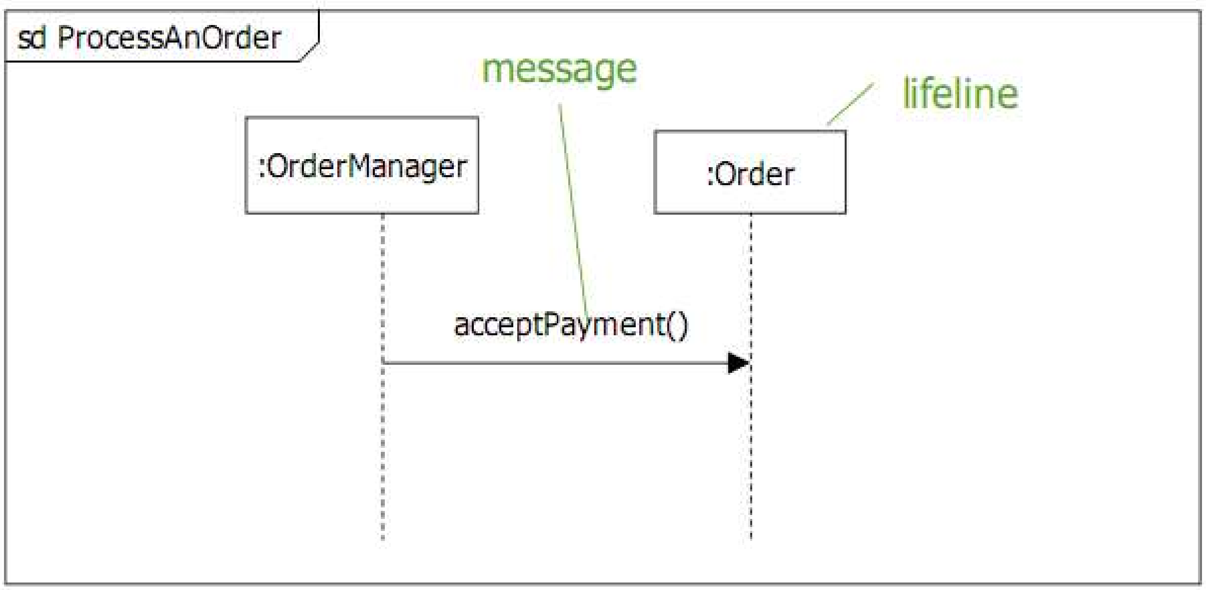
Behavioral (Dynamic) Modeling

- Behavioral (Dynamic) Modeling
  - Use Case Diagrams (Already covered)
  - Activity Diagrams (Already covered)
  - Statechart Diagrams (Next chapter)
  - **Interaction Diagrams**
    - Sequence Diagram
    - Communication Diagram

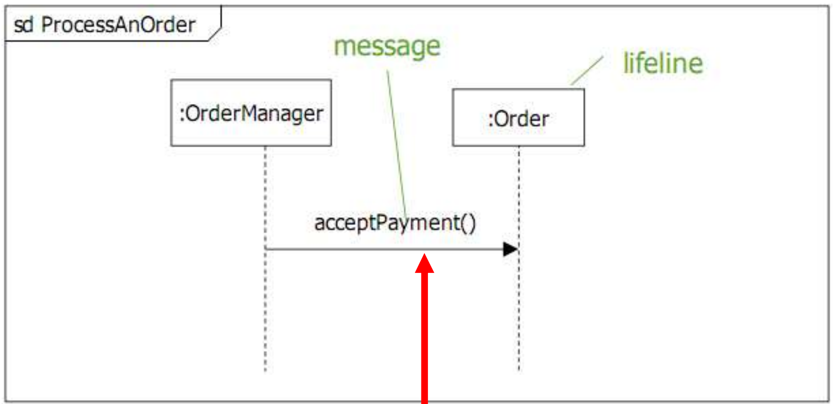
Sequence Diagram
(Messages, Method calls, etc.)

Sequence Diagrams

- Interaction diagrams capture an interaction as:
  - *Lifelines* – participants in the interaction
  - *Messages* – communications between lifelines



Interaction Diagrams

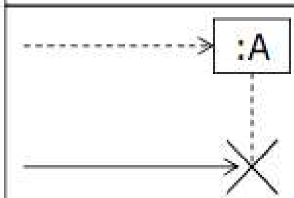



```
Class OrderManager{
    Order o;
     void someMethod {
     o.acceptPayment();
     }
}
```

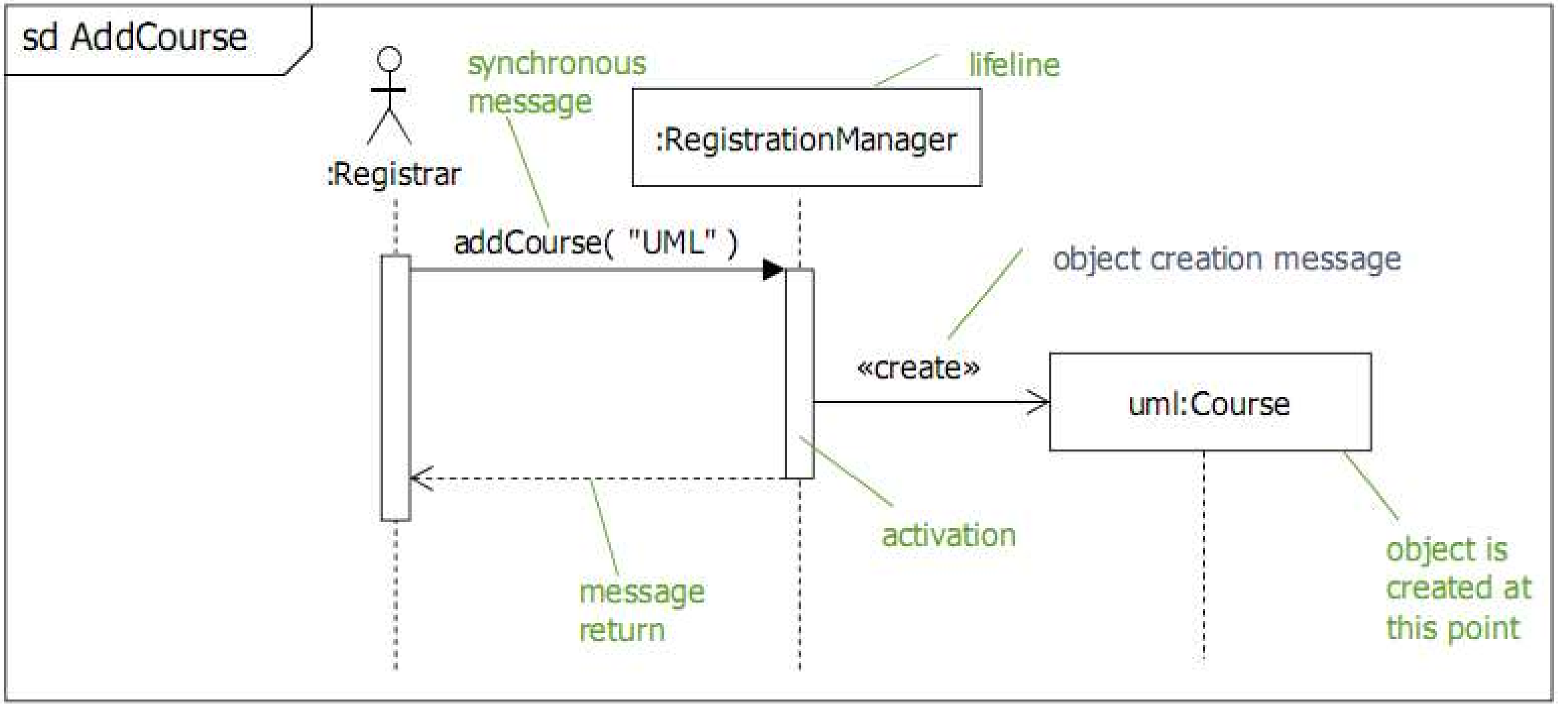
```
Class Order {
    void acceptPayment(){
        //some implementation
    };
}
```

Messages

- A message represents a communication between two lifelines

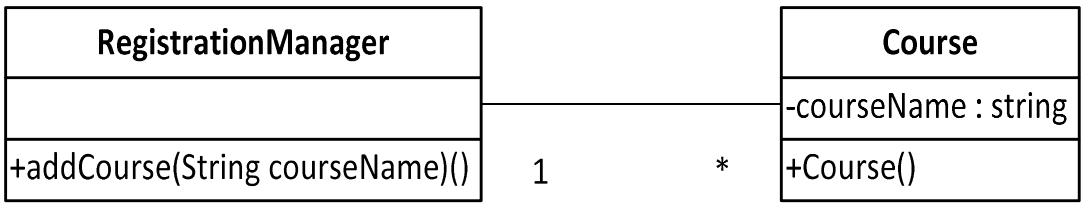
sender → receiver/ target	type of message	semantics
	synchronous message	calling an operation synchronously the sender waits for the receiver to complete
	asynchronous send	calling an operation asynchronously, sending a signal the sender *does not* wait for the receiver to complete
	message return	returning from a synchronous operation call the receiver returns focus of control to the sender
	creation destruction	the sender creates the target the sender destroys the receiver
	found message	the message is sent from outside the scope of the interaction
	lost message	the message fails to reach its destination

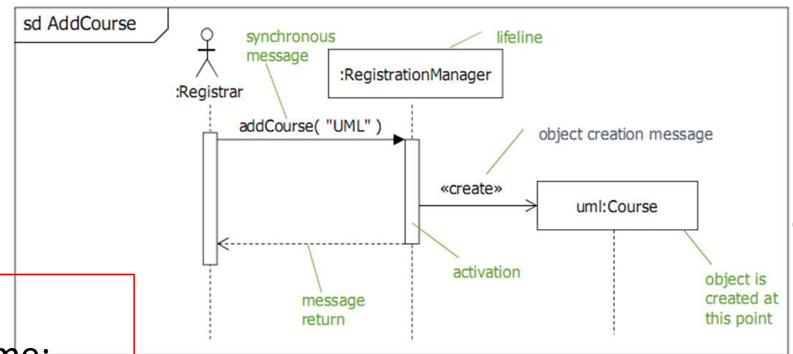
Sequence diagram syntax



- *All* interaction diagrams may be prefixed sd to indicate their type
- Activations indicate when a lifeline has focus of control - they are often omitted from sequence diagrams

Sequence diagram syntax

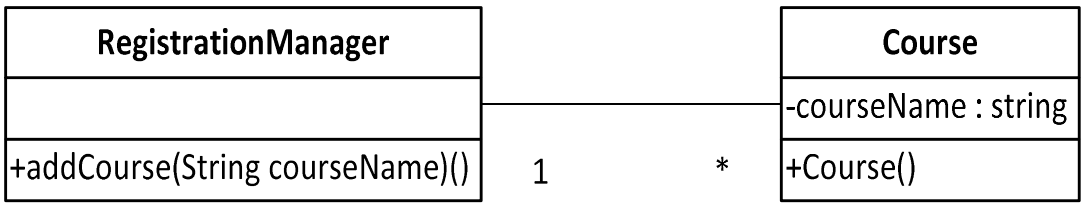


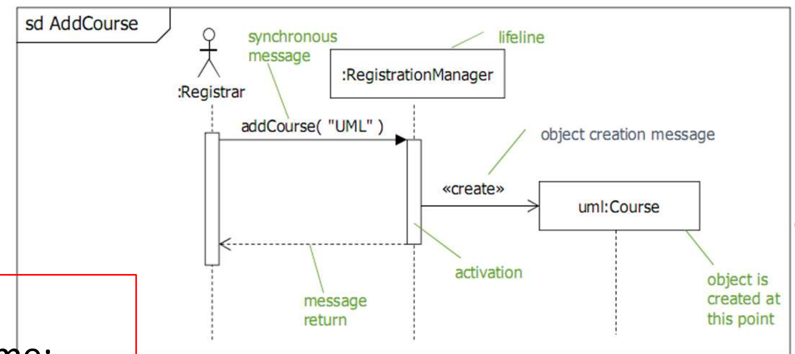



```
Class RegistrationManager{
  Course [ ] courses;

   addCourse(String courseName){
      Course c = new Course(courseName);
      courses.add(c);
   }
}
```

```
Class Course{
  String courseName;

  Course (String name){
     courseName = name;
   }

}
```

Sequence diagram syntax

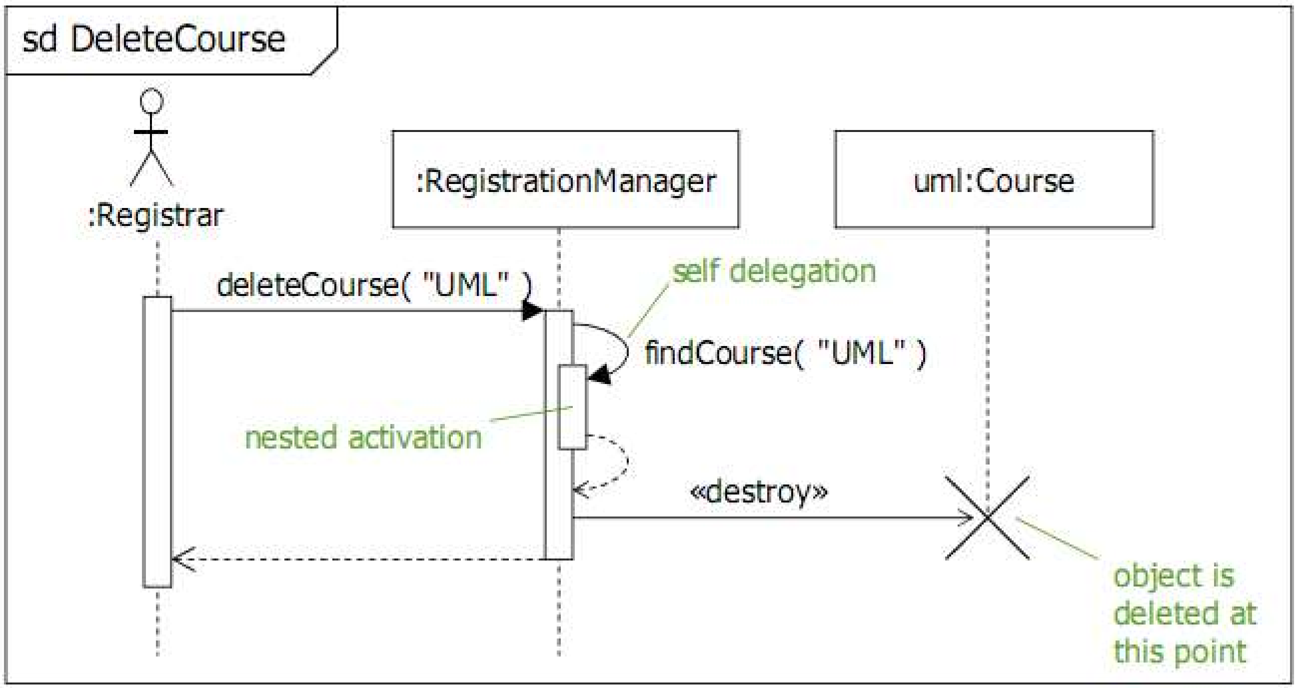





```
Class RegistrationManager{
  Course [ ] courses;

   addCourse(String courseName){
      Course c = new Course(courseName);
      courses.add(c);
   }
}
```

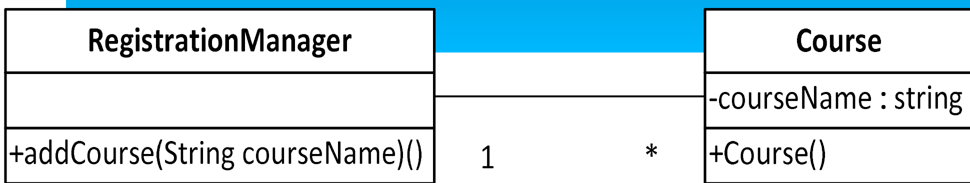
```
Class Course{
  String courseName;

  Course (String name){
     courseName = name;
   }

}
```

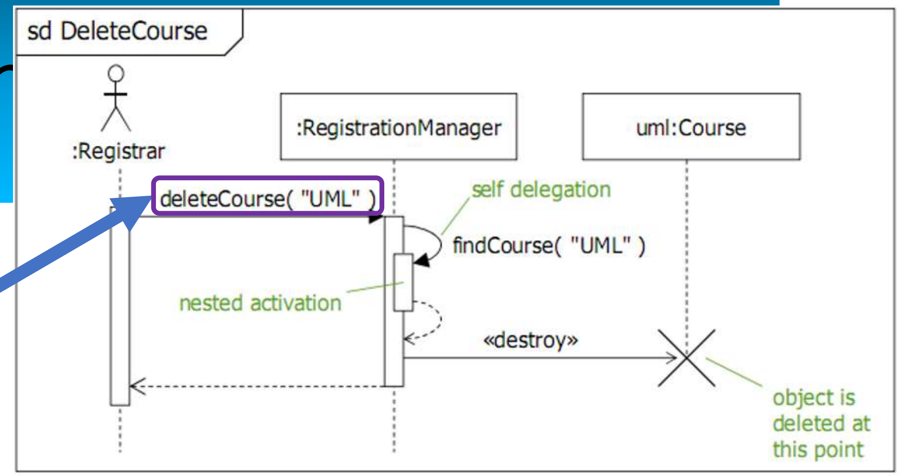
Deletion and self-delegation



- Self delegation is when a lifeline sends a message to itself
  - Generates a nested activation
- Object deletion is shown by terminating the lifeline's tail at the point of deletion by a large X

Sequence diagram syn

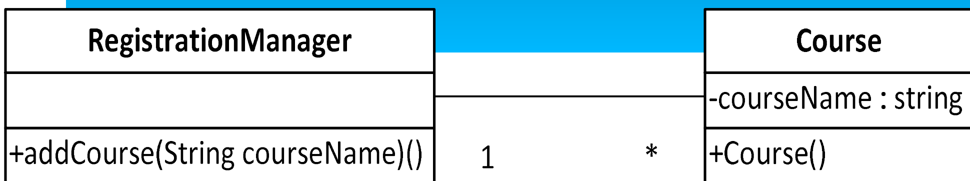




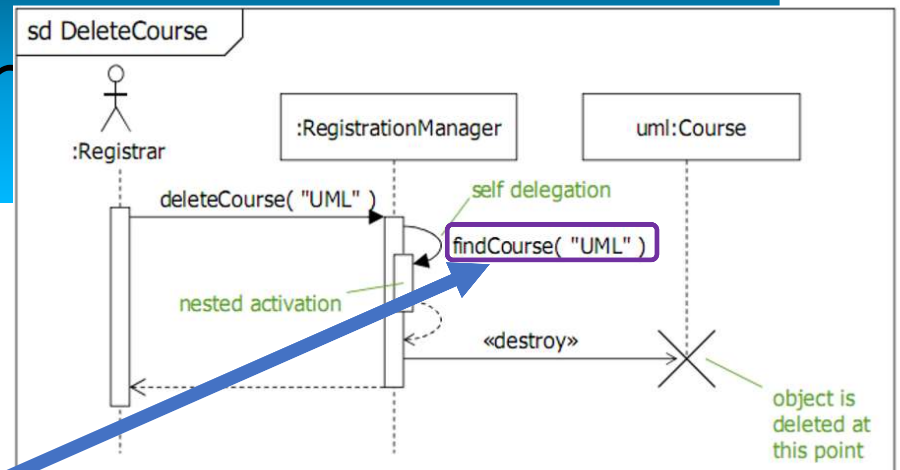

```
Class RegistrationManager{
  Course [ ] courses;

   addCourse(String courseName){
      Course c = new Course(courseName);
      courses.add(c);
   }

   deleteCourse(String courseName){


   }



}
```

```
Class Course{
   String courseName;

   Course (String name){
      courseName = name;
   }

}
```

Sequence diagram syn

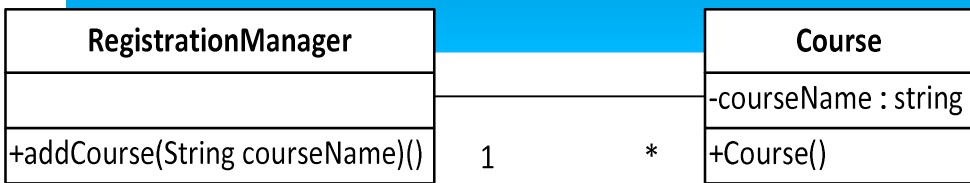


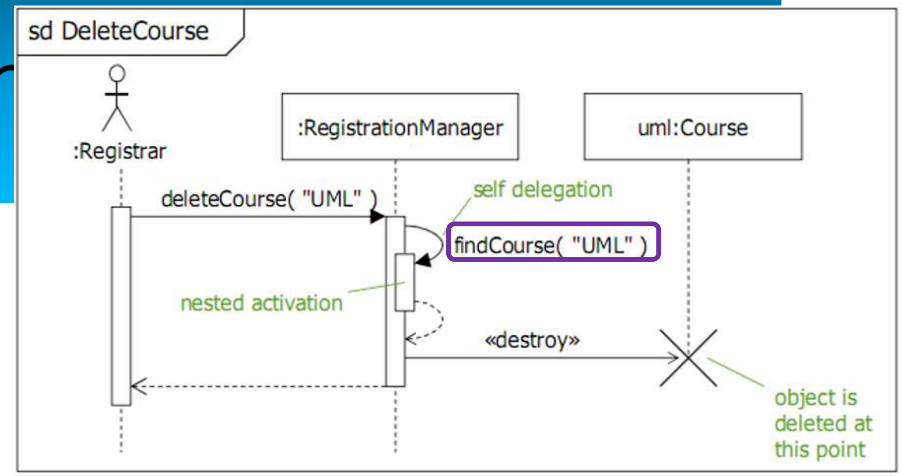

```
Class RegistrationManager{
  Course [ ] courses;

   addCourse(String courseName){
      Course c = new Course(courseName);
      courses.add(c);
   }

   deleteCourse(String courseName){
      deletedCourse = findCourse(courseName);

   }

}
```




```
Class Course{
   String courseName;

   Course (String name){
      courseName = name;
   }

}
```

Sequence diagram syn

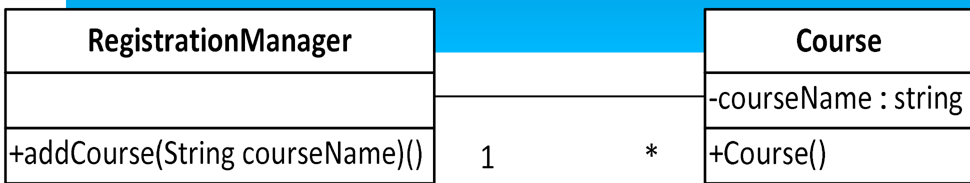


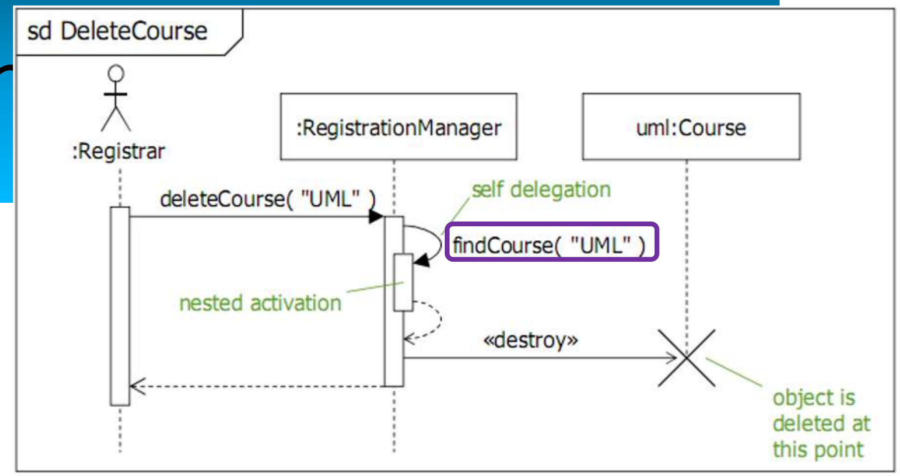



```
Class RegistrationManager{
  Course [ ] courses;

   addCourse(String courseName){
      Course c = new Course(courseName);
      courses.add(c);
   }

   deleteCourse(String courseName){
      deletedCourse = findCourse(courseName);

   }

Course findCourse(String courseName){



   }
}
```

```
Class Course{
   String courseName;

   Course (String name){
      courseName = name;
    }

}
```

Sequence diagram syn

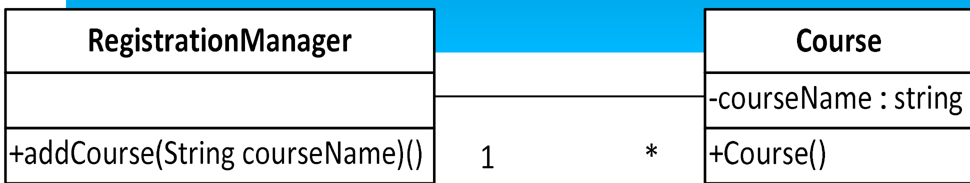




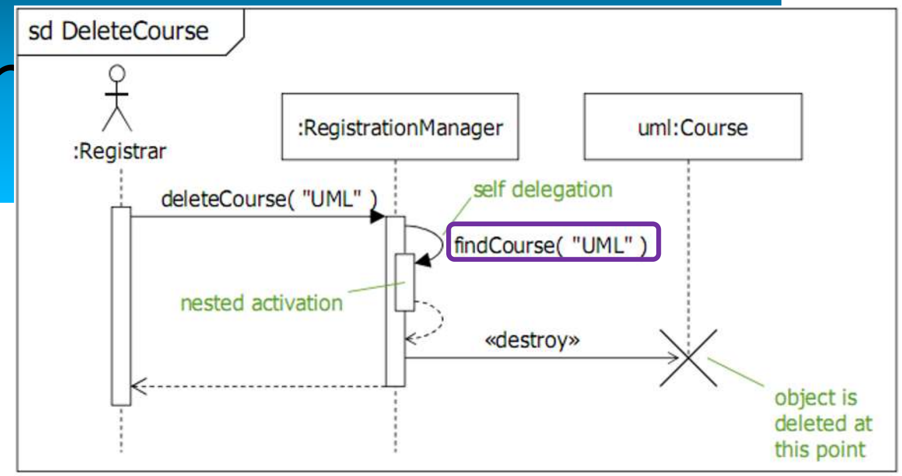

```
Class RegistrationManager{
  Course [ ] courses;

   addCourse(String courseName){
      Course c = new Course(courseName);
      courses.add(c);
   }

   deleteCourse(String courseName){
      deletedCourse = findCourse(courseName);

   }

Course findCourse(String courseName){
      for (Course c : courses){
          if (c.courseName.equals(courseName)
              return c;
      }
   }
}
```

```
Class Course{
   String courseName;

   Course (String name){
      courseName = name;
    }

}
```

Sequence diagram syn




```
Class RegistrationManager{
  Course [ ] courses;

   addCourse(String courseName){
      Course c = new Course(courseName);
      courses.add(c);
   }

   deleteCourse(String courseName){
      deletedCourse = findCourse(courseName);
      courses.remove(deletedCourse);
   }

Course findCourse(String courseName){
      for (Course c : courses){
         if (c.courseName.equals(courseName)
             return c;
      }
   }
}
```



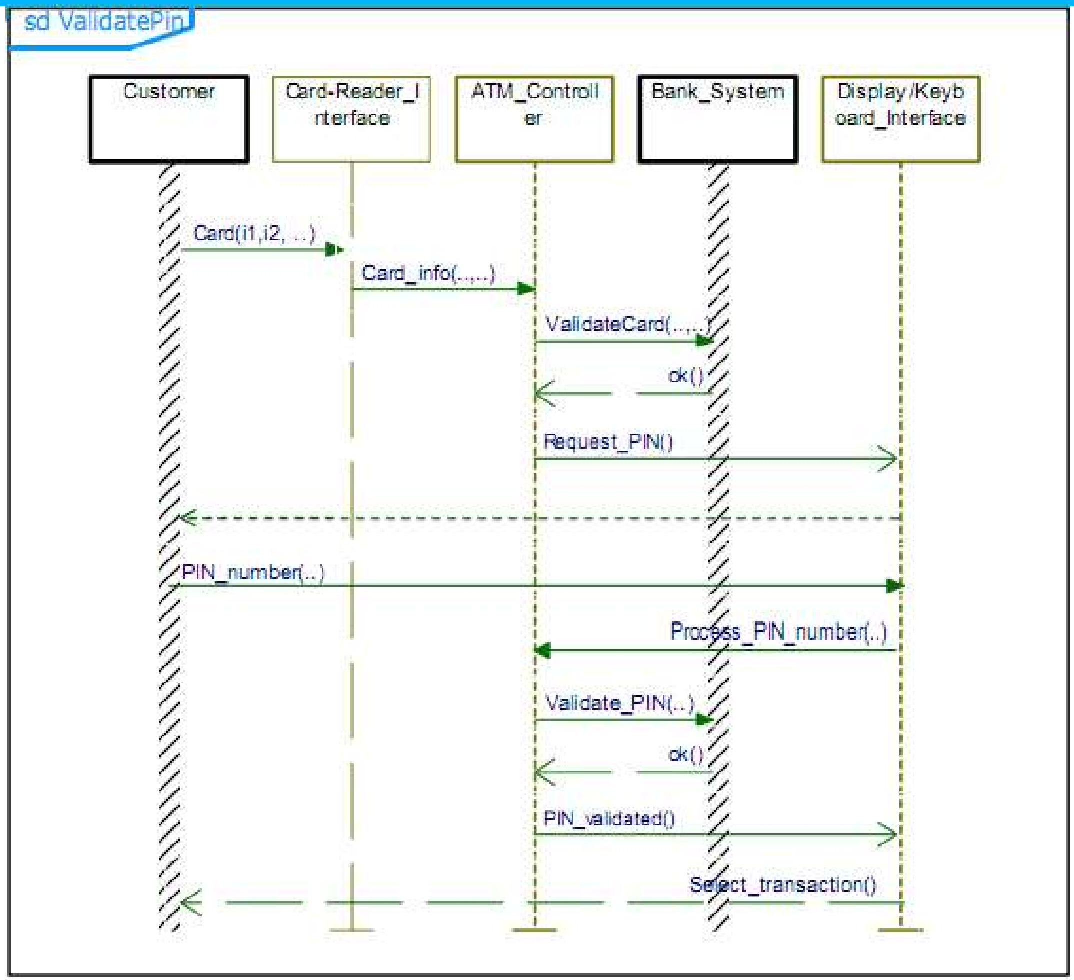

```
Class Course{
  String courseName;

  Course (String name){
     courseName = name;
   }

}
```

Exercise -- ATM

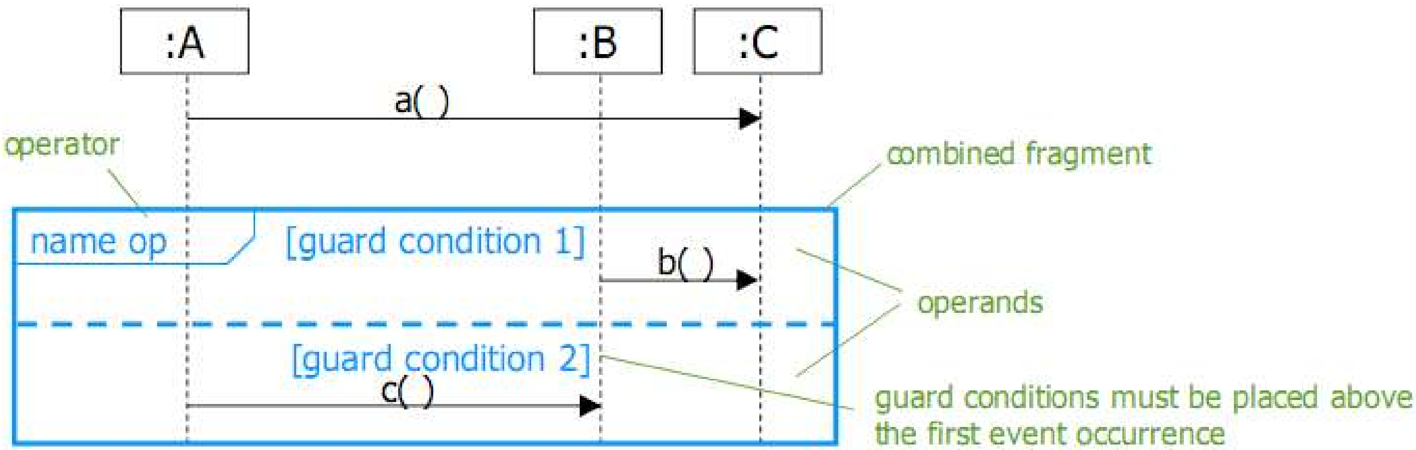
A bank has several ATMs which are geographically distributed and connected via a wide area network to a central server.

An ATM machine has a card reader, a cash dispenser, a keyboard/display, and a receipt printer. By using the ATM machine, a customer can withdraw cash from either a checking or savings account, query the balance of an account, or transfer funds from one account to another. A transaction is initiated when a customer inserts an ATM card into the card reader . Encoded on the magnetic strip on the back of the ATM card are the card number, the start date, and the expiration date. Assuming the card is recognized, the banking system validates the ATM card to determine that the user-entered PIN matches the PIN maintained by the system, and that the card is not lost or stolen. The customer is allowed three attempts to enter the correct PIN; the card is confiscated if the third attempt fails. Card that have been reported lost or stolen are also confiscated. The ATM operator is responsible for adding cash, starting the ATM, shutting it down.

ATM Example: Validate PIN



Combined fragments



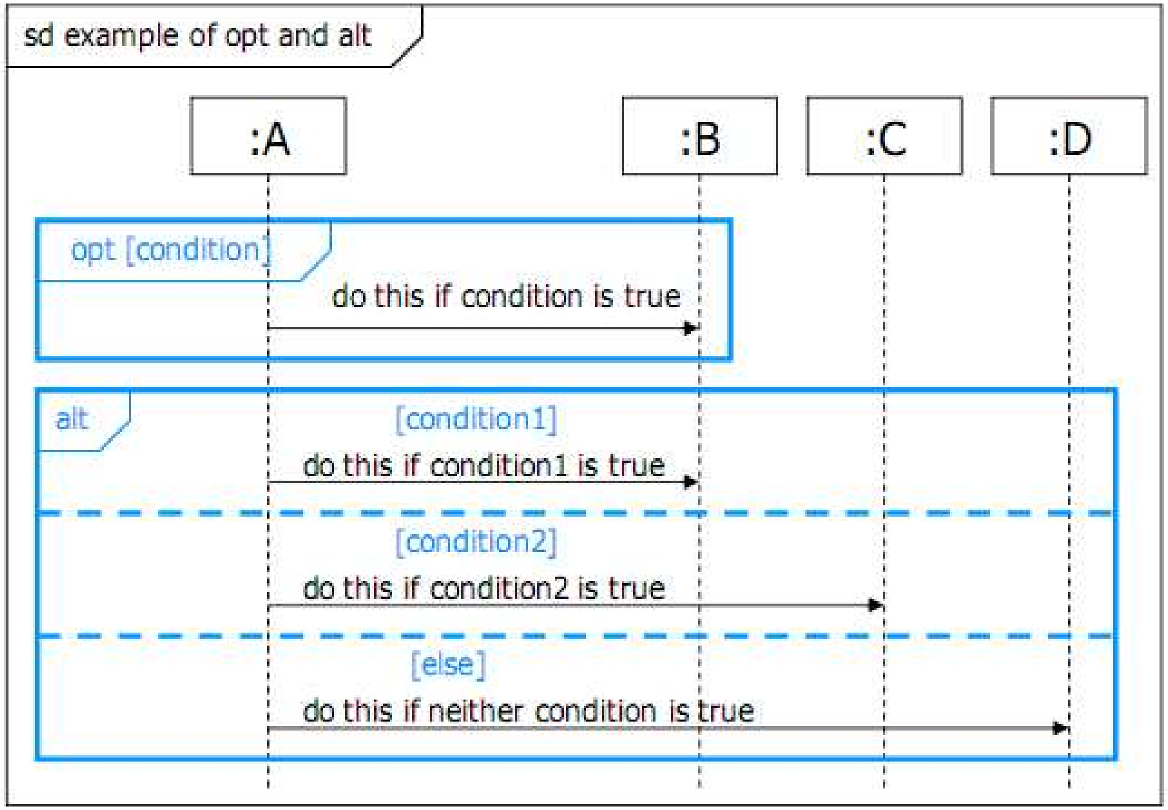
- Sequence diagrams may be divided into areas called *combined fragments*
- Combined fragments have one or more *operands*
- *Operators* determine how the operands are executed
- *Guard conditions* determine whether operands execute. Execution occurs if the guard condition evaluates to true
  - A single condition may apply to all operands OR
  - Each operand may be protected by its own condition

Common operators

operator	long name	semantics
opt	Option	There is a single operand that executes if the condition is true (like if … then)
alt	Alternatives	The operand whose condition is true is executed. The keyword else may be used in place of a Boolean expression (like select… case)
loop	Loop	This has a special syntax: loop min, max [condition] Iterate min times and then up to max times while condition is true
break	Break	The combined fragment is executed rather than the rest of the enclosing interaction
ref	Reference	The combined fragment refers to another interaction

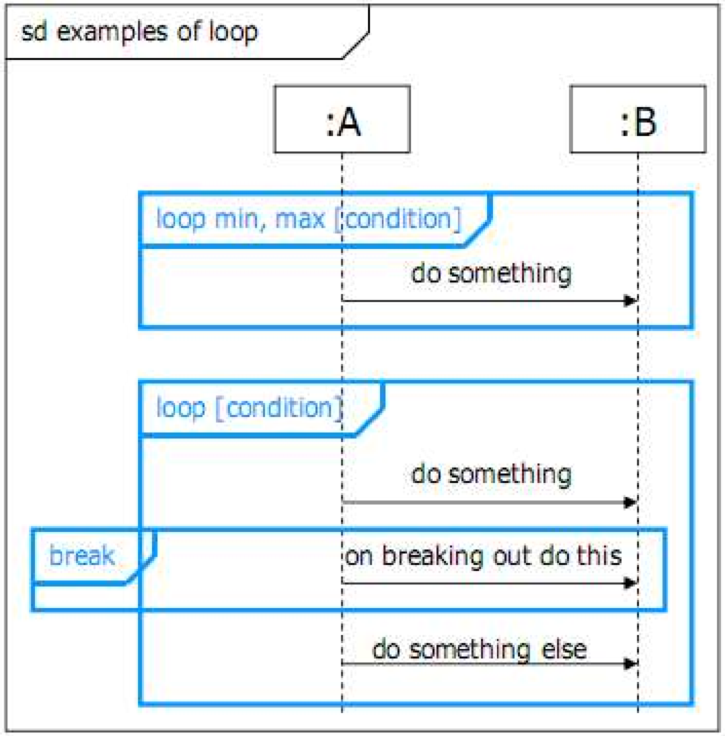
Branching with opt and alt

- **opt** semantics:
  - single operand that executes if the condition is true
- **alt** semantics:
  - two or more operands each protected by its own condition
  - an operand executes if its condition is true
  - use else to indicate the operand that executes if *none* of the conditions are true



Iteration with loop and break

- loop semantics:
  - Loop min times, then loop (max – min) times while condition is true
- loop syntax
  - A loop without min, max or condition is an infinite loop
  - If only min is specified then max = min
  - condition can be
    - Boolean expression
    - Plain text expression *provided* it is clear!
- Break specifies what happens when the loop is broken out of:
  - The break fragment executes
  - The rest of the loop after the break does *not* execute
- The break fragment is *outside* the loop and so should overlap it as shown



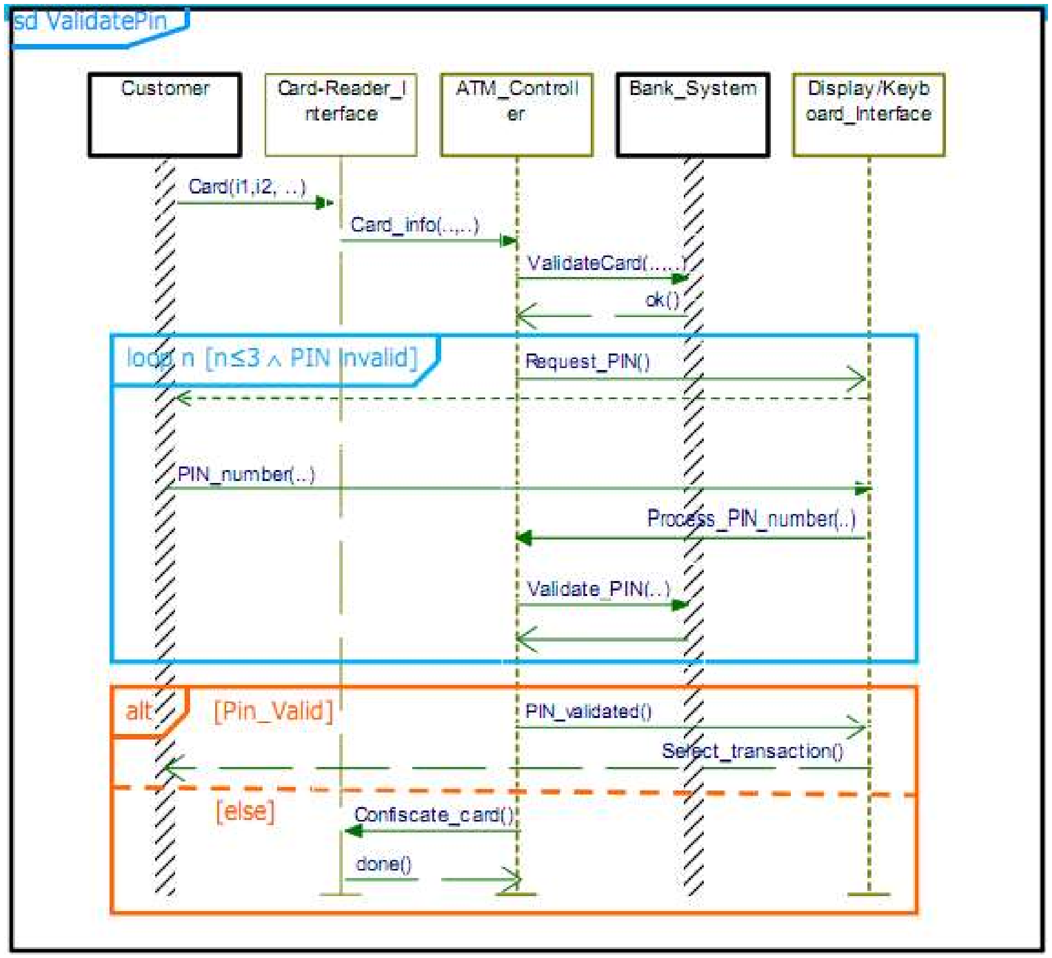
Loop idioms

type of loop	semantics	loop expression
infinite loop	keep looping forever	loop *
for i = 1 to n {body}	repeat (n) times	loop n
while(booleanExpression) {body}	repeat while booleanExpression is true	loop [booleanExpression]

repeat
{body}
while(booleanExpression)
	execute once then repeat while booleanExpression is true	loop 1, * [booleanExpression]

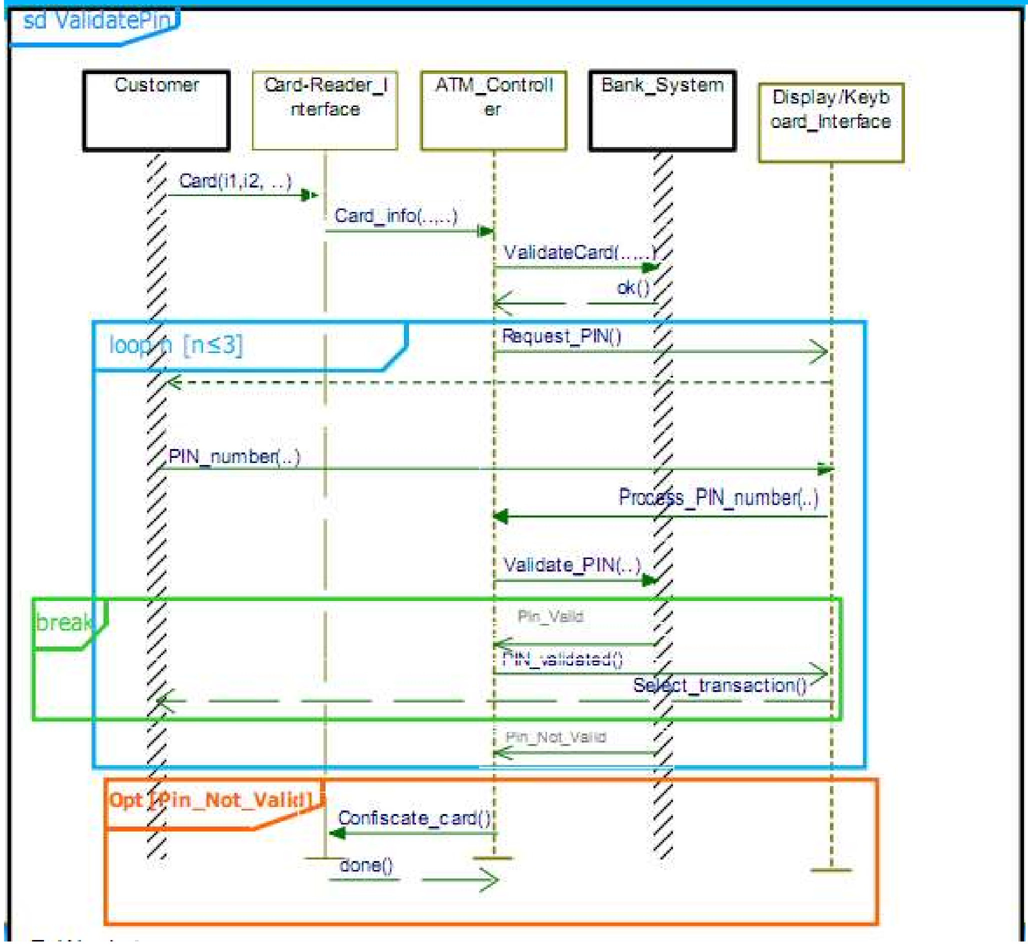
forEach object in set {body}	Execute the loop once for each object in a set	loop [for each object in objectType]

ATM Example: Validate PIN



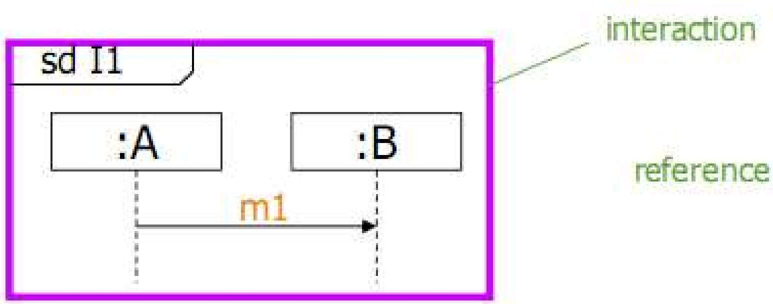
The customer can enter a PIN number up to three times. If the third attempt fails, the card is confiscated

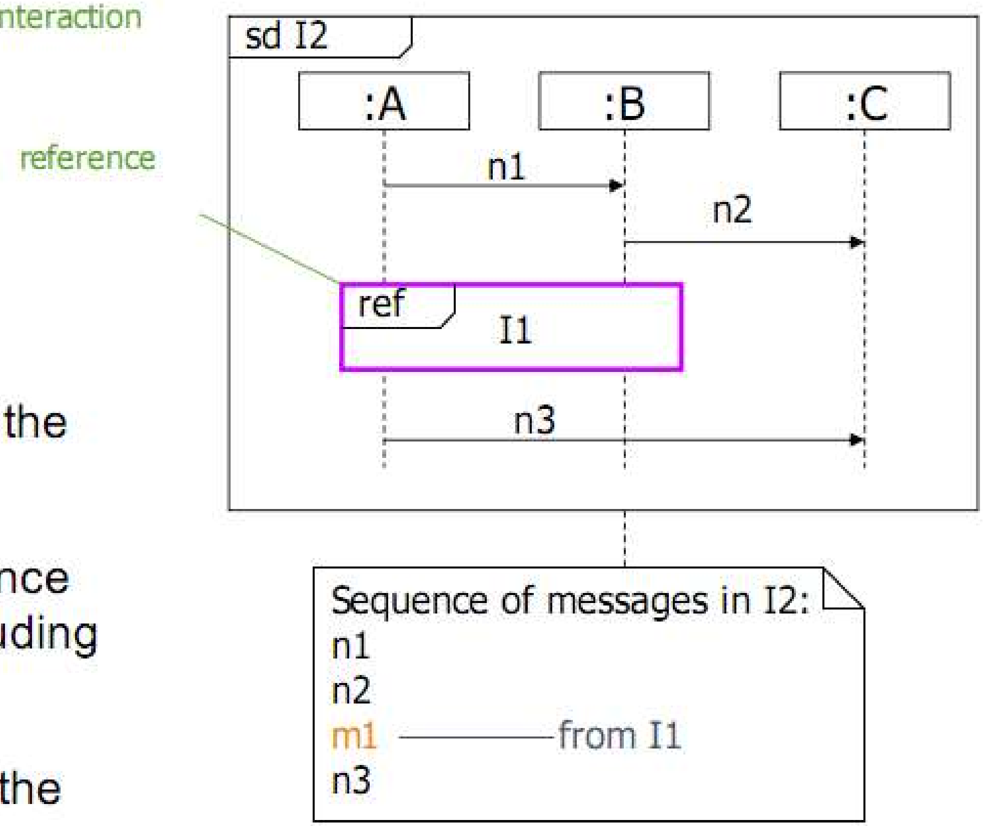
ATM Example: Validate PIN



The customer can enter a PIN number up to three times. If the third attempt fails, the card is confiscated

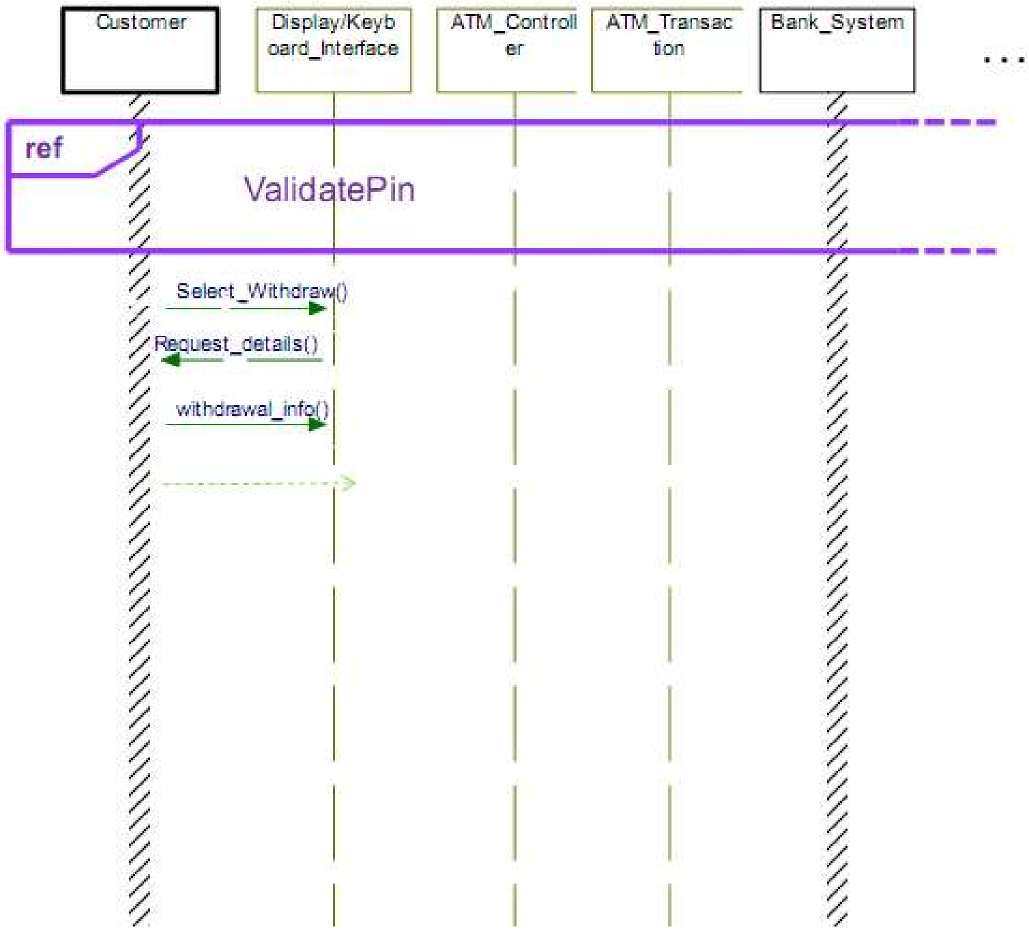
Interaction occurrences

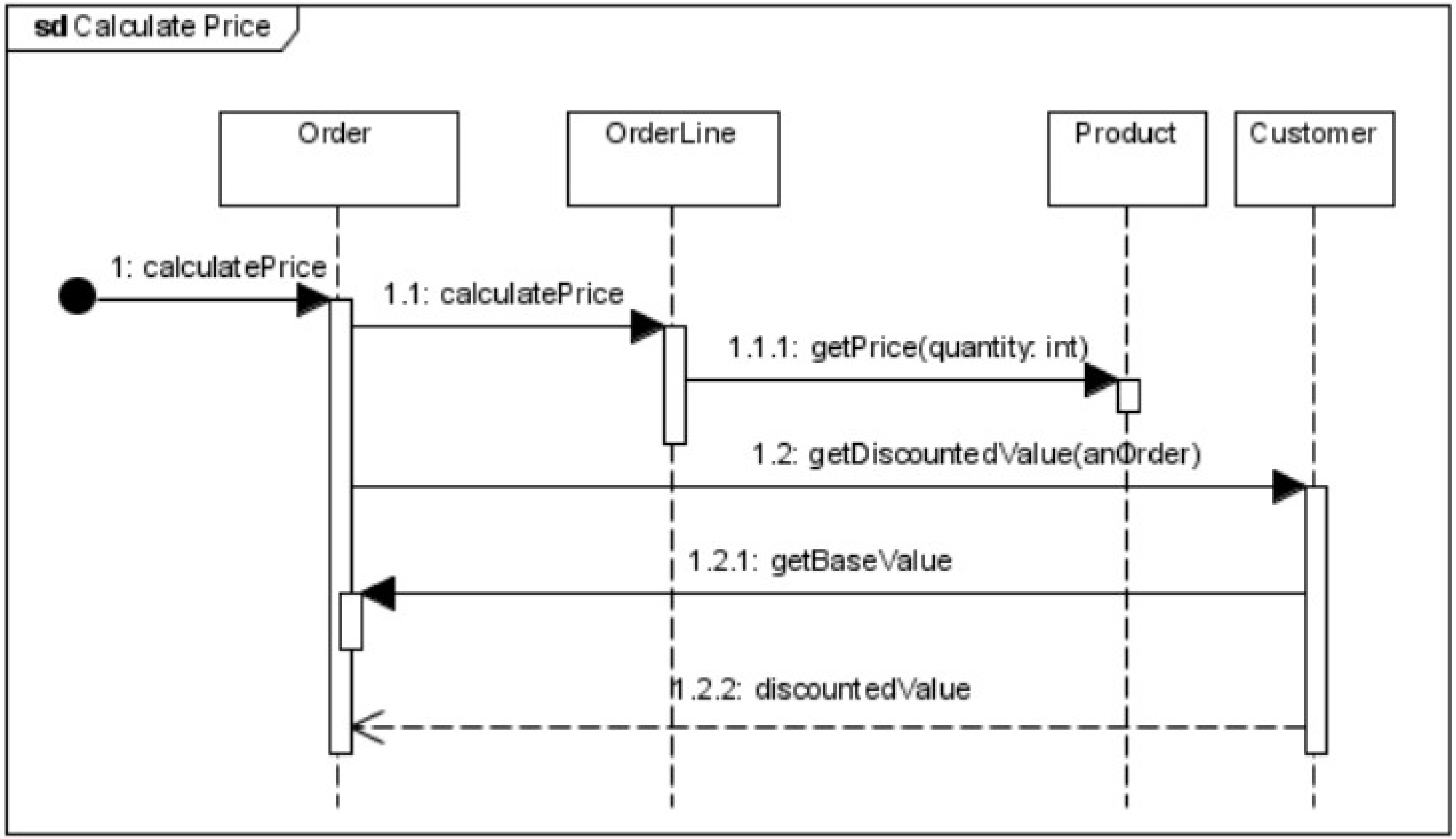


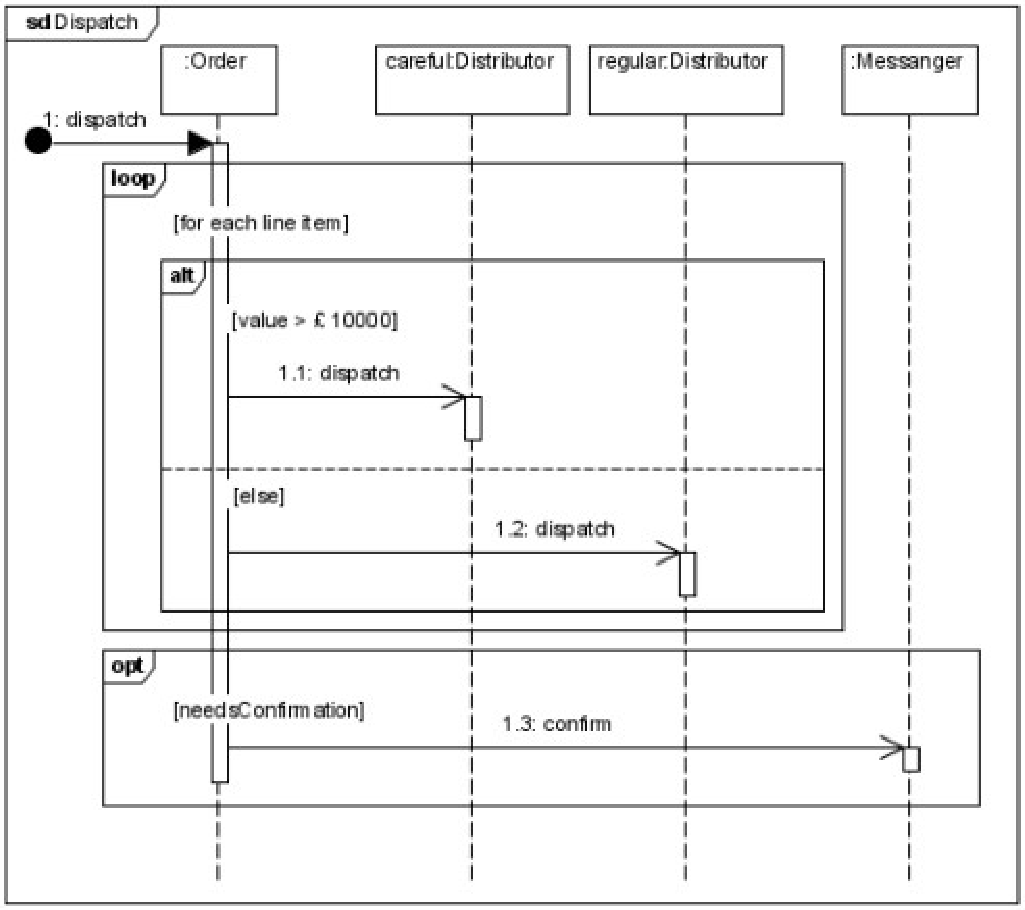


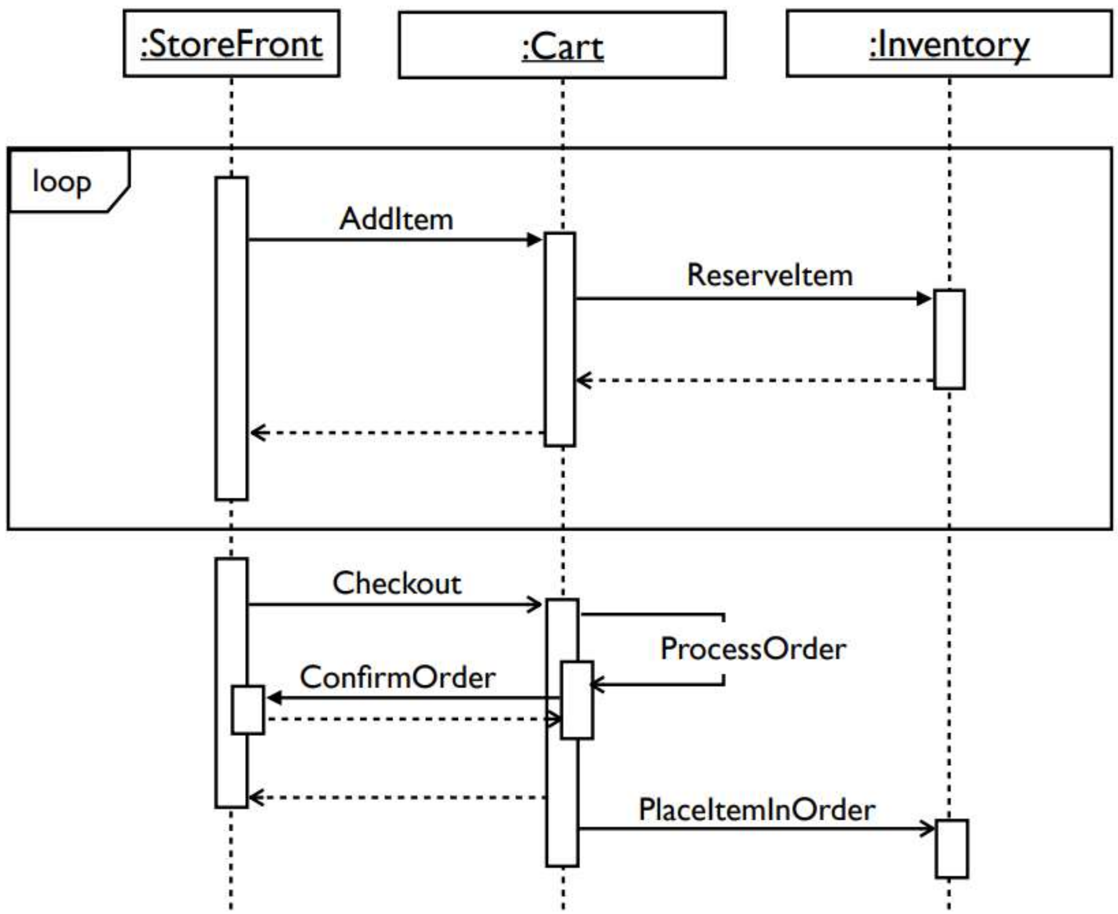
- A reference is inserted into the including interaction
  - All lifelines in the reference must also be in the including interaction
- Draw the reference across the lifelines it uses

ATM Interaction Diagram: Withdrawal




sd Calculate Price
Order
OrderLine
Product
Customer
1: calculatePrice
1.1: calculatePrice
1.1.1: getPrice(quantity: int)
1.2: getDiscountedValue(anOrder)
1.2.1: getBaseValue
1.2.2: discountedValue


sd Dispatch
:Order
careful:Distributor
regular:Distributor
:Messanger
1: dispatch
loop
[for each line item]
alt
[value > £ 10000]
1.1: dispatch
[else]
1.2: dispatch
opt
[needsConfirmation]
1.3: confirm

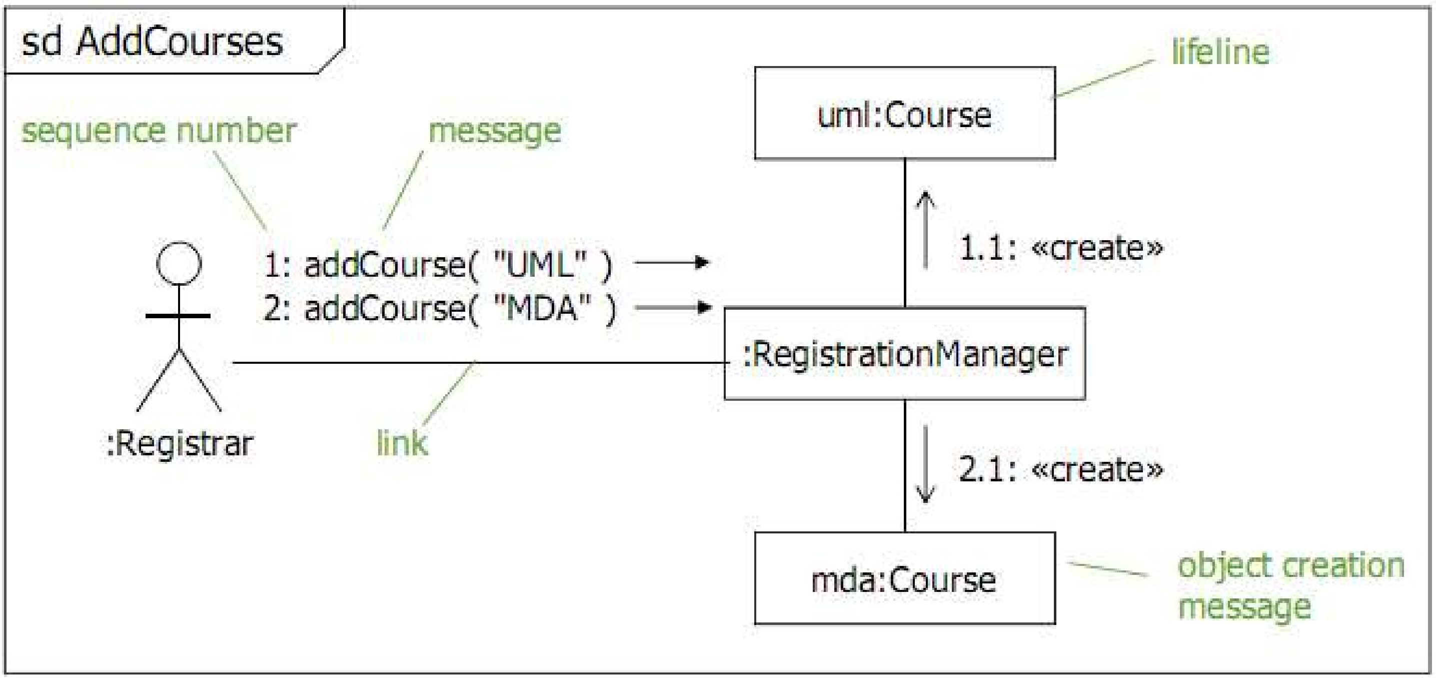


:StoreFront
:Cart
:Inventory
loop
AddItem
ReserveItem
Checkout
ProcessOrder
ConfirmOrder
PlaceItemInOrder

Communication Diagram (Messages, Method calls, etc.)

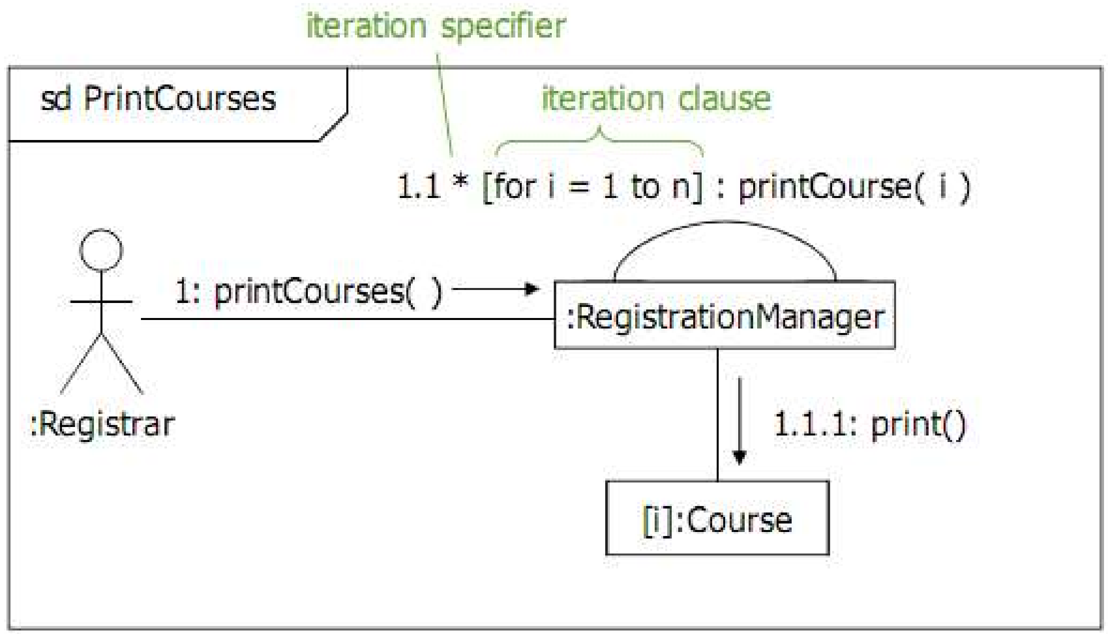
Communication diagram syntax

- Communication diagrams emphasize the structural aspects of an interaction - how lifelines connect together

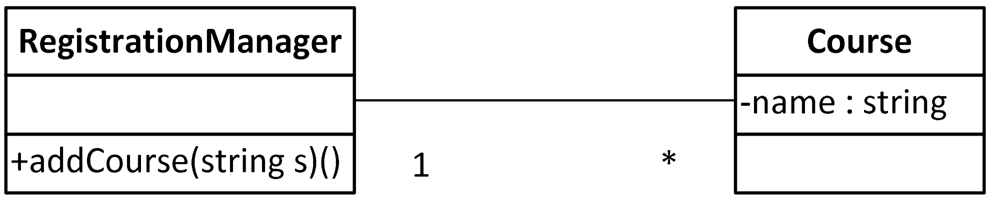


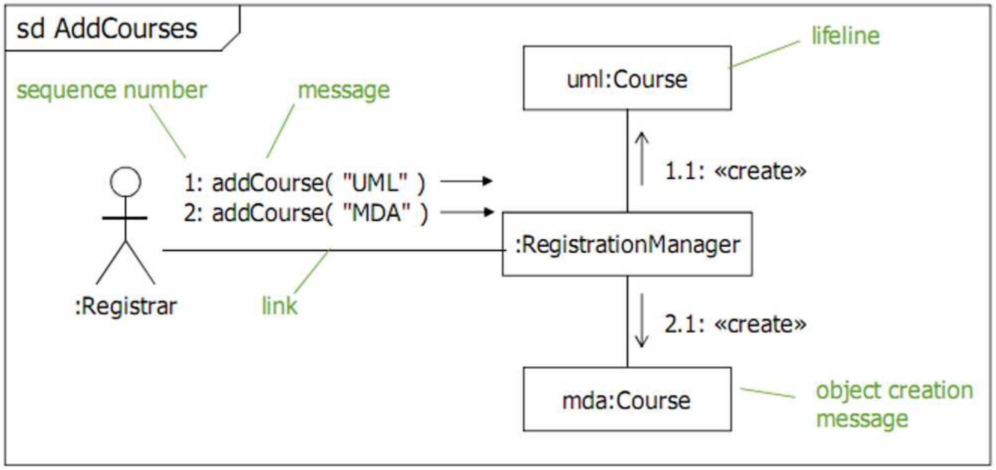
Iteration

- Iteration is shown by using the *iteration specifier* (*), and an optional *iteration clause*
  - There is no prescribed UML syntax for iteration clauses
  - Use code or pseudo code
- To show that messages are sent in parallel use the parallel iteration specifier, *//

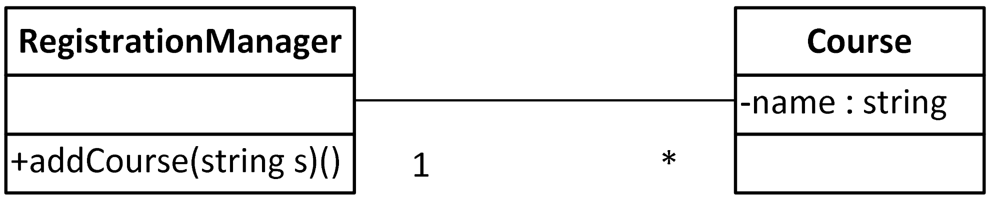


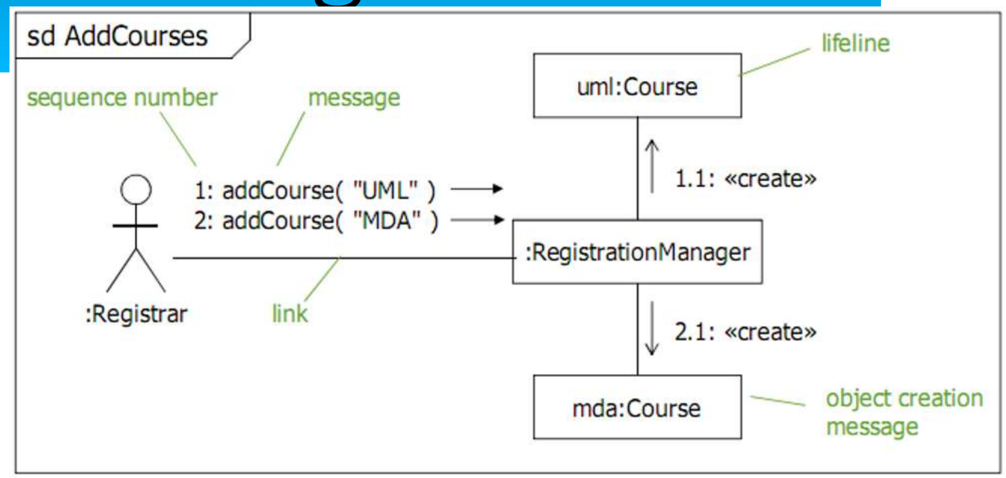
Communication diagram

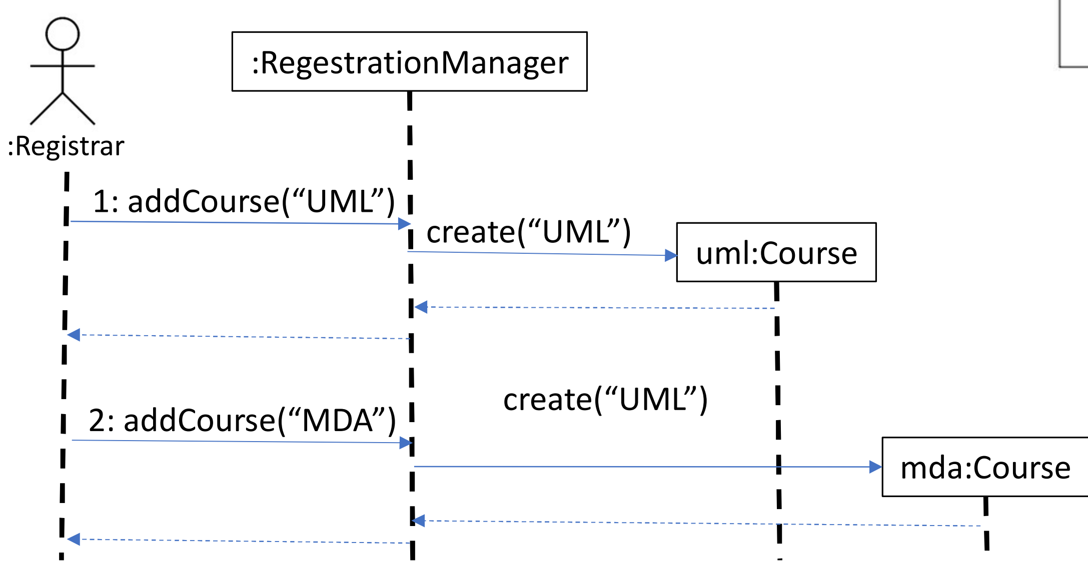




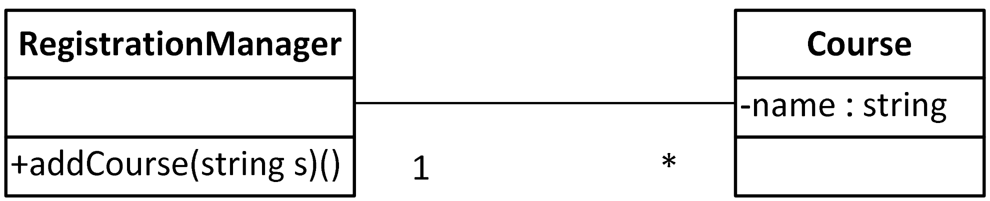
Communication diagram

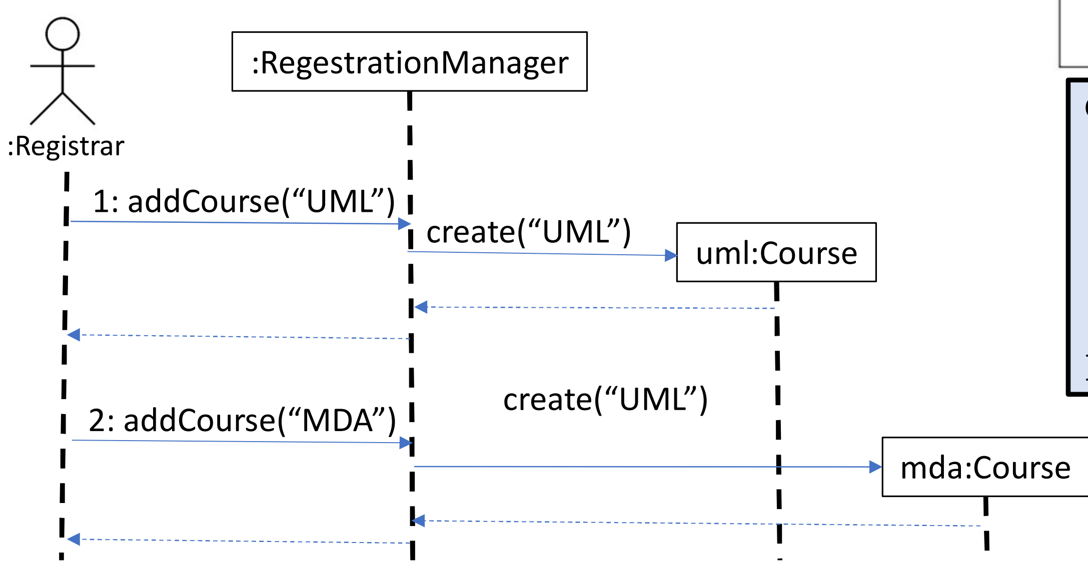


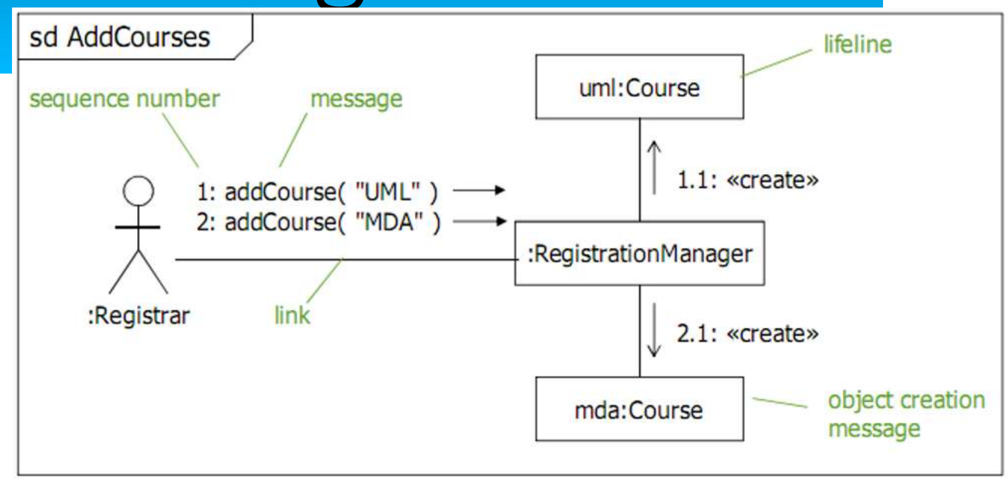




Communication diagram





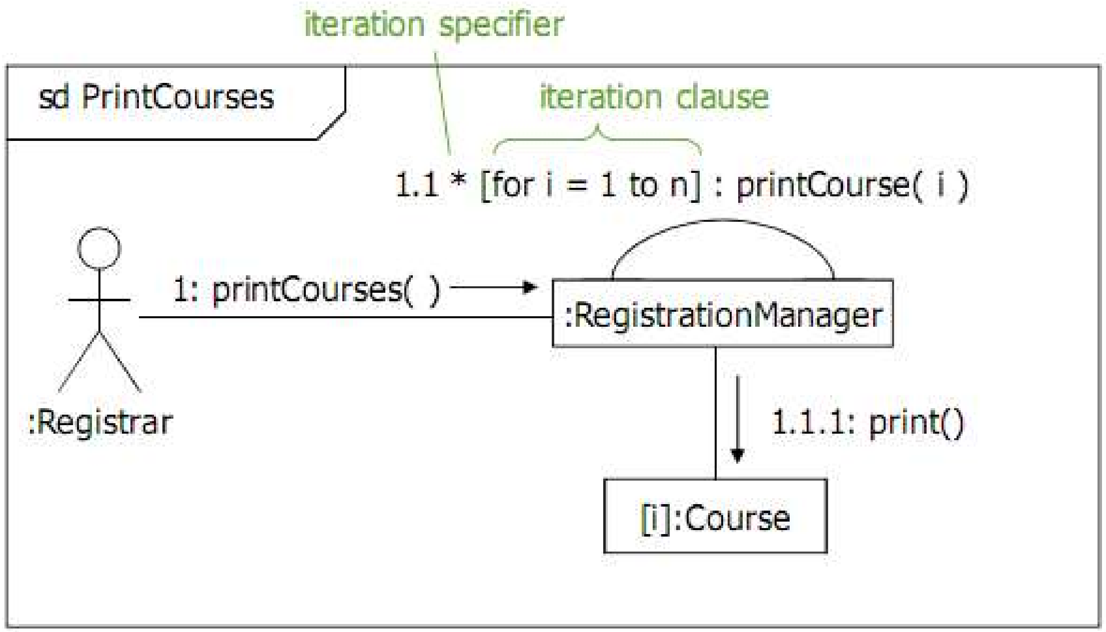



```
Class RegestrationManager{
  List <Course> courses = new ArrayList<Course>();

  void addCourse (String s){
    Course c = new Course(s);
     courses.add(c);
  }
}
```

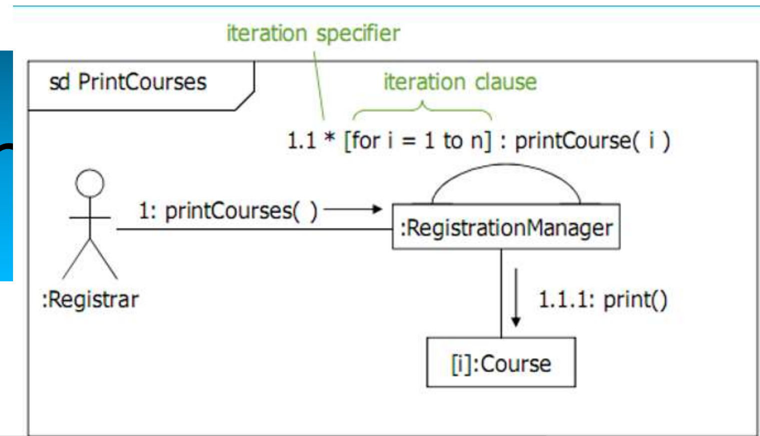
```
Class Course{
  String courseName;

  Course (String s){
    courseName = s;
  }
}
```

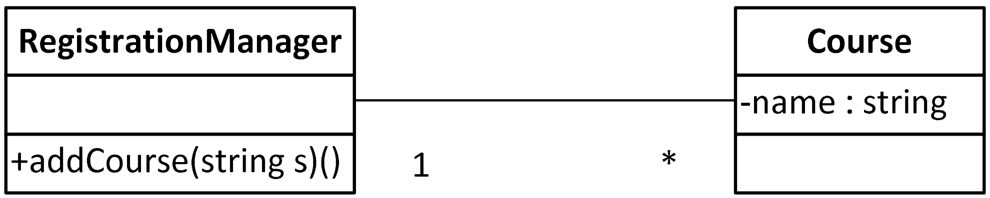
Iteration

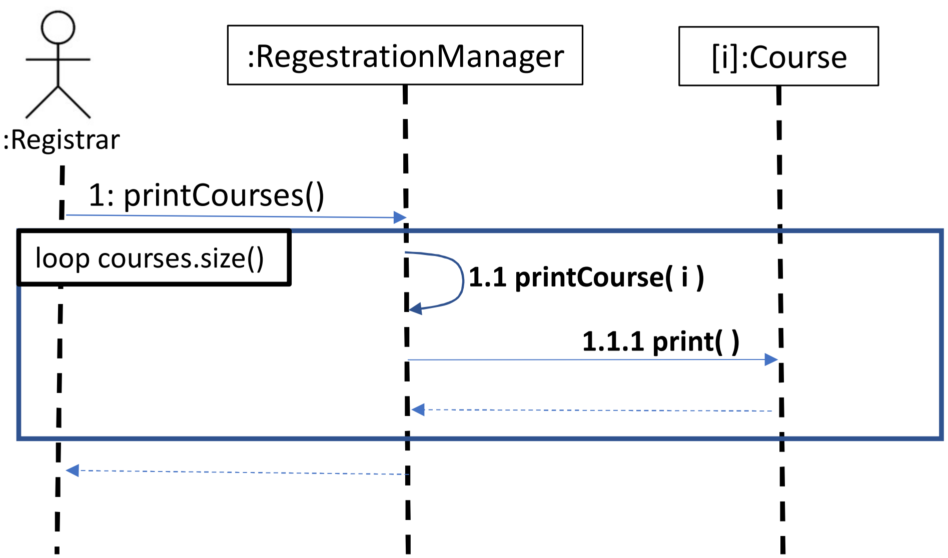
- Iteration is shown by using the *iteration specifier* (*), and an optional *iteration clause*
  - There is no prescribed UML syntax for iteration clauses
  - Use code or pseudo code
- To show that messages are sent in parallel use the parallel iteration specifier, *//



Communication



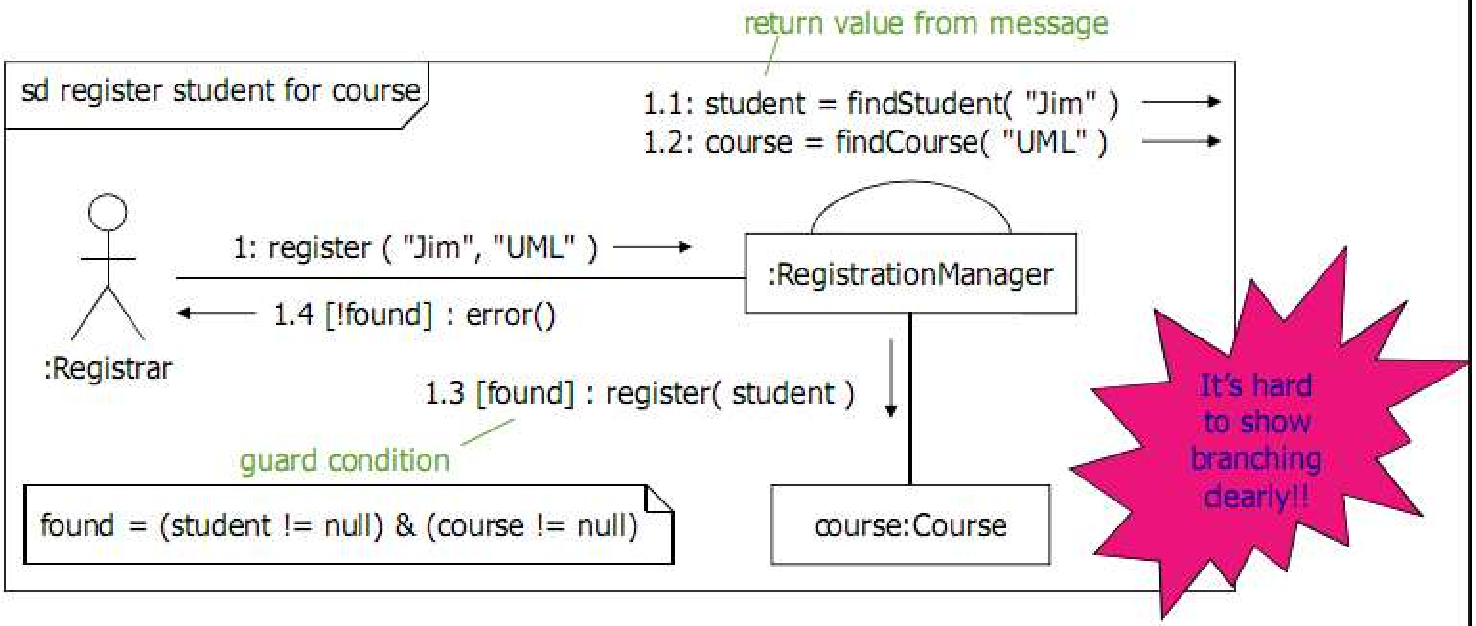




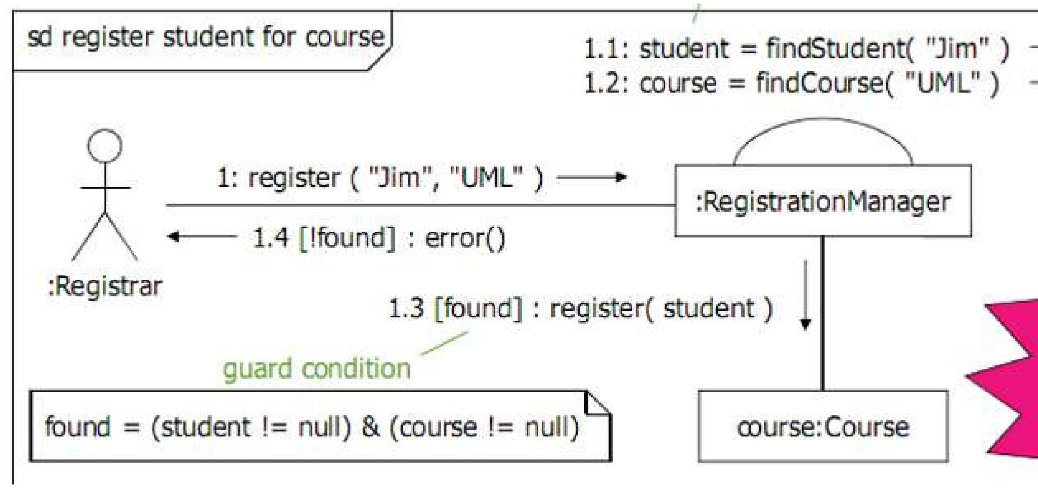

```
Class RegestrationManager{
  List <Course> courses = new ArrayList<Course>();

  void addCourse (String s){
    Course c = new Course(s);
     courses.add(c);
  }

  void printCourses( ){
    for (i=1 to courses.size( ){
      printCourse(i);
    }
  }

  void printCourse(int i){
    Course c = courses.get(i);
    c.print( );
  }
}
```

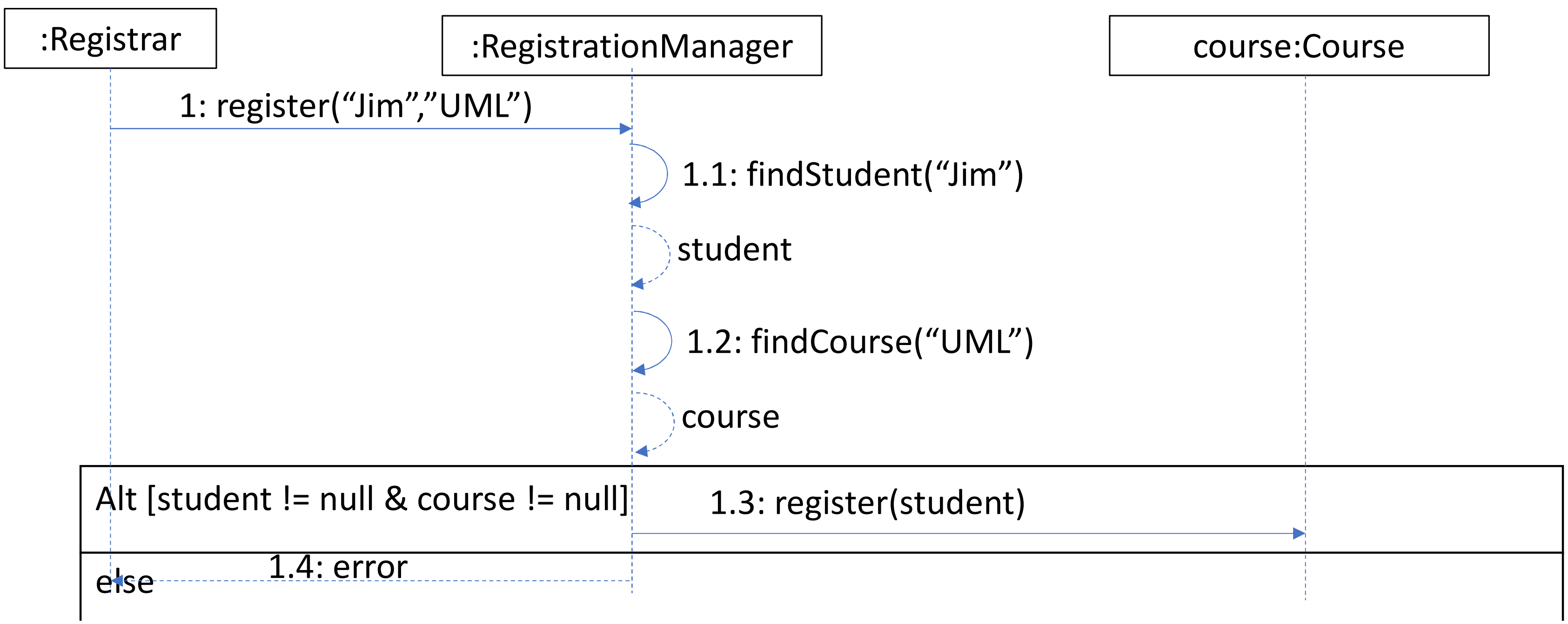
```
Class Course{
  String courseName;

  Course (String s){
    courseName = s;
  }

  void print( ){
    println(courseName);
  }
}
```

Branching

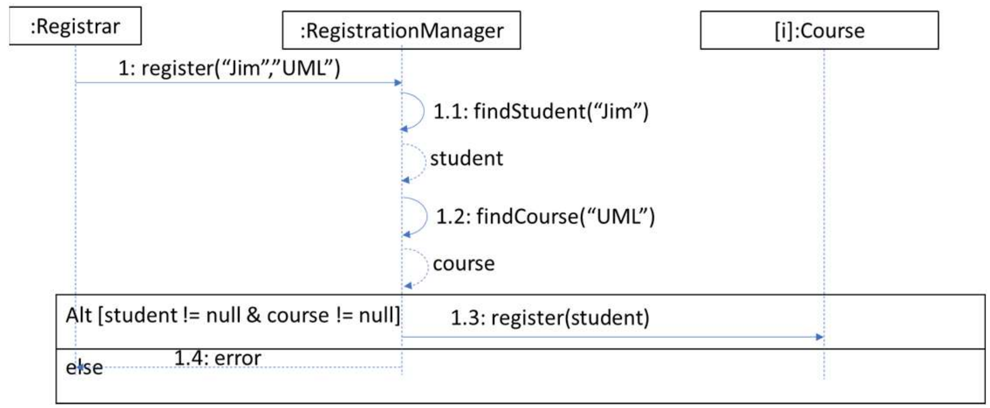


- Branching is modelled by prefixing the sequence number with a *guard condition*
  - There is no prescribed UML syntax for guard conditions
  - In the example above, we use the variable **found**. This is true if both the student and the course are found, otherwise it is false


sd register student for course
1.1: student = findStudent("Jim")
1.2: course = findCourse("UML")
1: register ("Jim", "UML")
:RegistrationManager
1.4 [!found] : error()
:Registrar
1.3 [found] : register(student)
guard condition
found = (student != null) & (course != null)
course:Course


:Registrar
:RegistrationManager
course:Course
1: register(“Jim”,”UML”)
1.1: findStudent(“Jim”)
student
1.2: findCourse(“UML”)
course
Alt [student != null & course != null]
1.3: register(student)
else
1.4: error

```
Class RegestrationManager{
  List <Course> courses = new ArrayList<Course>();
  List <Student> students = new ArrayList<Student>();

  void addCourse (String s){
    Course c = new Course(s);
     courses.add(c);
  }

  void printCourses( ){
    for (i=1 to courses.size( ){
      printCourse(i);
    }
  }

  void printCourse(int i){
    Course c = courses.get(i);
    c.print( );
  }

 void register(String stuName, String courseName){
    Student student = findStudent(stuName);
    Course course = findCourse(courseName);
    if (student != null && course != null){
        course.register(student);
    else
        print(error);
    }
 }

 Student findStudent(String sName){
   for (Student s:students){
      if (s.getName.equals(sName)
         return s;
   }
  }

 Course findCourse(String cName){
  for (Course c: courses){
     if (c.getCourseName.equals(cName)
       return c;
  }
}
```




```
Class Course{
  String courseName;
  List <Student> students = new ArrayList<Student>();

  Course (String s){
    courseName = s;
  }

  void print( ){
    println(courseName);
  }
  String getCourseName(){
    return courseName;
  }

 void register(Student s){
   students.add(s);
 }
}
```