Unified Modeling Language (UML)

Feza BUZLUCA
Istanbul Technical University
Computer Engineering Department
http://faculty.itu.edu.tr/buzluca
http://www.buzluca.info

1

Overview

- Introduction
- Class Diagrams
- Communication Diagrams
- Sequence Diagrams

2

Unified Modeling Language (UML)

- is a visual language for specifying, constructing, and documenting the artifacts of a software
- is not a method to design systems; it is used to visualize the analysis and design
- makes it easier to understand and document software systems
- supports teamwork because UML diagrams are more understandable than program code

Types of UML Diagrams

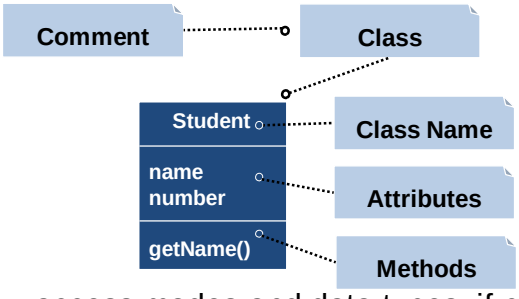
- There are different kinds of UML diagrams, which are used in different phases of a software development process
- Here, we will discuss three types, which are used in design and coding phases
  - Class diagrams
  - Communication diagrams
  - Sequence diagrams

UML Current Specification

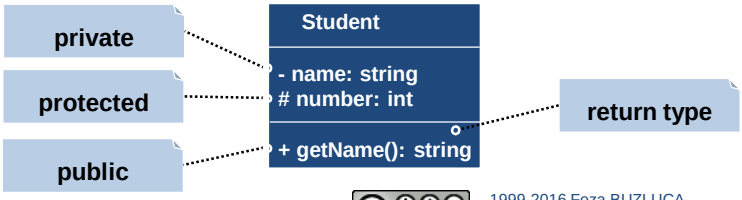
- Current specification of UML is available on the Web site of the Object Management Group (OMG)
- URL: http://www.omg.org/spec/UML/
- In this course, the current specification of UML (version 2.5) is used

Class Diagrams

- A class diagram shows the structure of the classes and the relationships between them

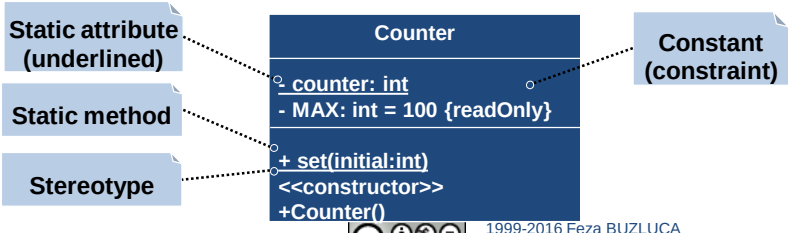


- It can also show access modes and data types, if needed



Class Diagrams

- **Comments:** Comments in UML are placed in dog-eared rectangles. We can use comments to
  - put anything we want in a diagram
  - add application- and program-specific details
- **Stereotypes:** A stereotype is a way of extending UML in a uniform way and remaining within the standard
  - We indicate a stereotype using: <<stereotype name>>
- **Constraints:** A constraint in UML is a text string in curly braces ({usually language specific}).
  - UML defines a language (Object Constraint Language – OCL) that we can use for writing constraints

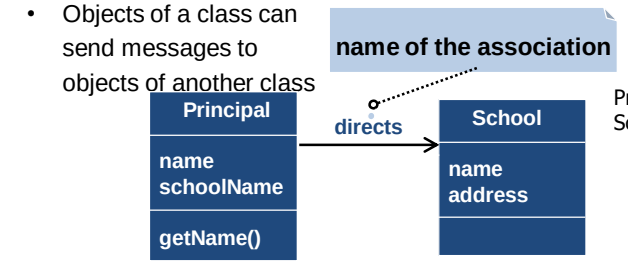


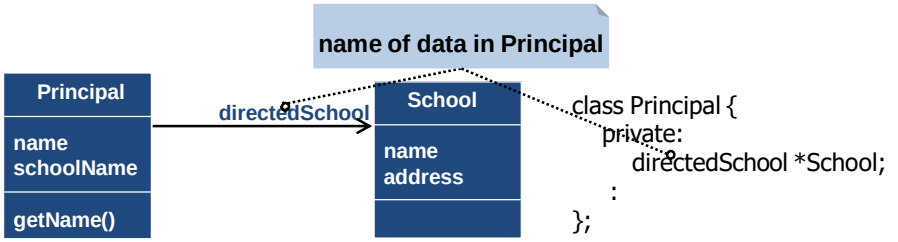
Relationships Between Classes

- A class diagram also shows the relationships between classes such as
  - association
  - aggregation
  - composition
  - inheritance

Association

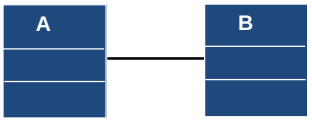
- A general type of relationship
- Objects of a class can send messages to objects of another class



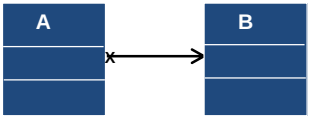


Association names are used in the conceptual perspective (analysis phase). In the software perspective, association names are unnecessary.

Association: Direction of the Message Flow



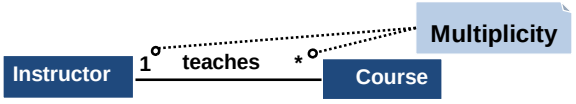
Direction of messages is unspecified.
Both may send messages to each other.



A can send messages to B.
A gets a service from B.
B cannot send messages to A.

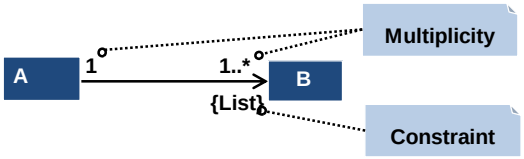
Association Relationship: Multiplicity

- Multiplicity indicates the number of any possible combination of objects of one class associated with objects from another class
- In other words, it shows the number of objects from that class that can be linked at runtime with one instance of the class at the other end of the association line
- Example:

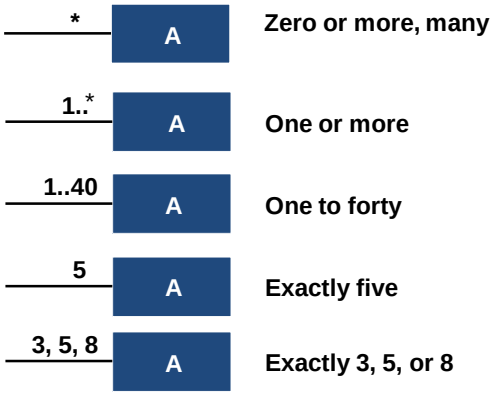


  - An instructor teaches zero or more courses
  - An association may also be read in reverse order
  - A course is given exactly by one instructor

Multiplicity Example



One object of class A is associated with one or more objects of class B at a time.
Class A includes a list that can contain one or more objects of class B.

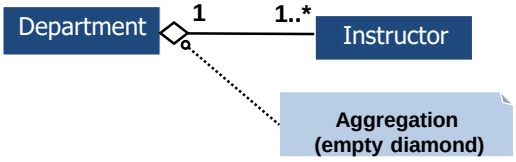


Aggregation and Composition

- Both are a type of association
- They are qualified by a “has-a” relationship
- There is a small difference between them

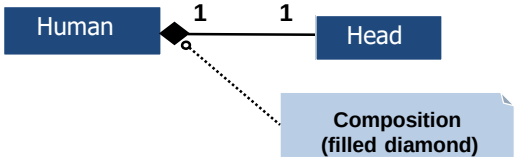
Aggregation

- Aggregation: It represents a “whole-part” relationship
- Example: A department has instructors
  - Parts (instructors) can still exist even if the whole (the department) does not exist
  - The same “part” object can belong to more than one “whole” object at the same time



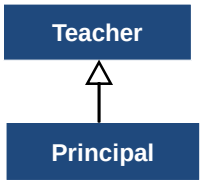
Composition

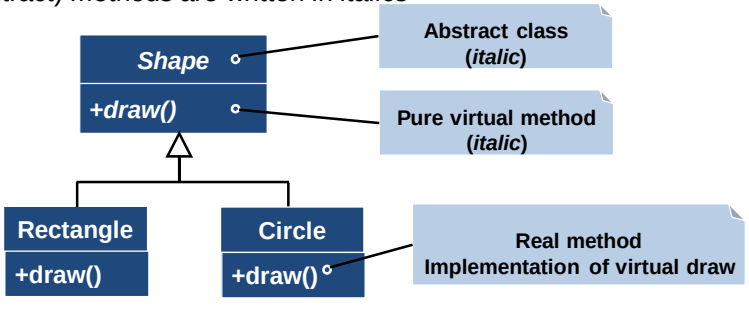
- Composition: Composition is a strong kind of aggregation where the parts cannot exist independently of the "whole" object
- Examples:
  - A human has a head
  - A car has an engine
- A composition relationship implies that
  a) An instance of the part belongs to only one composite object.
  b) An instance of the part must belong to a composite object. It cannot exist without the “whole” object.
  c) The composite is responsible for the creation and deletion of its parts. If the composite is destroyed, its parts must also be destroyed.



Inheritance

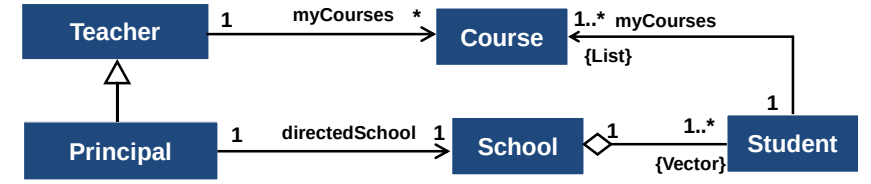
- Inheritance: The white triangular arrow should point toward the class being extended
- The arrow should point upward (This is not a rule of UML, but it feels more logical and is easier to read in this form)
- The names of abstract classes and pure virtual (abstract) methods are written in italics





Inheritance: Example

- Example: Partial class diagram of a part of a system




```
class Teacher {
   private:
     Course *myCourses; //may be a linked list
   :
};

class Principal: public Teacher {
  private:
     School directedSchool;
     // or
     School *directedSchool;
};
```

```
class School {
   private:
      vector<Student*> students;
   :
};

class Student {
  private:
     list<Course*> myCourses;
     :
};
```

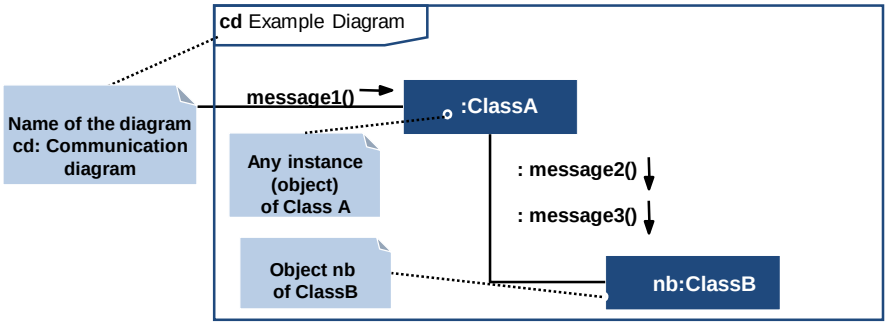
- vector and list are data types from C++ STL

UML Interaction Diagrams

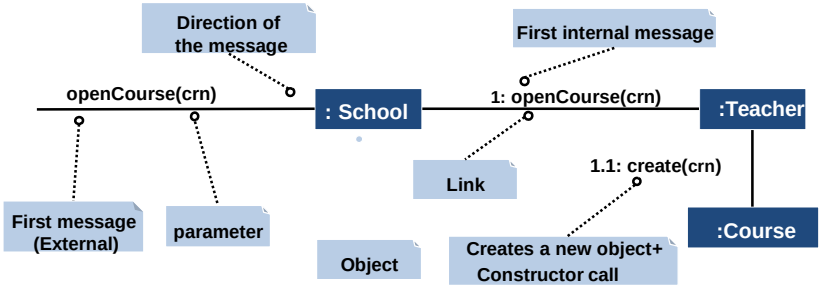
- Interaction diagrams illustrate how objects interact via messages
- There are two common types
  - Communication diagrams
  - Sequence diagrams
- Both can express similar interactions
- Sequence diagrams are more notationally rich, but communication diagrams have their use as well, especially for wall sketching

Communication Diagrams

- They illustrate object interactions in a graph or network format, in which objects can be placed anywhere on the diagram



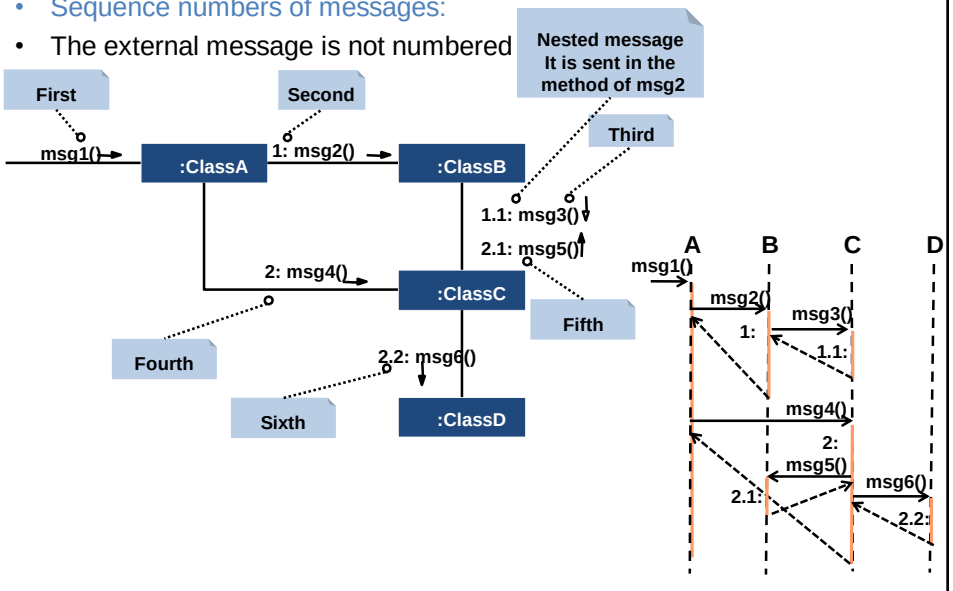
Communication Diagram Example




```
class Teacher {
  private:
     Course *myCourse;
  public:
    void openCourse( int crn ){                    // openCourse method of the Teacher
            myCourse = new Course( crn );          // An object of type Course is created
            // Other operations ...
    }
    // Other members ...
};
```

Communication Diagrams: Seq. No.

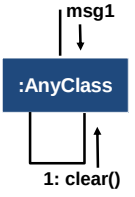
- Sequence numbers of messages:
- The external message is not numbered



21

Communication Diagrams: Messages to Self or "This"

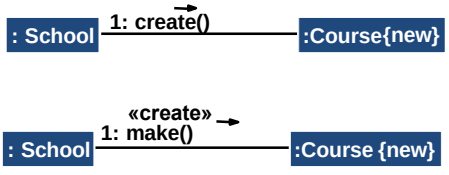
- A message can be sent from an object to itself



22

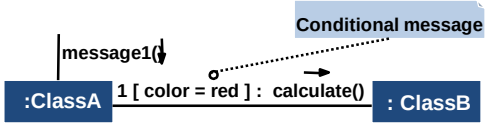
Communication Diagrams: Creation of Instances

- Any message can be used to create an instance, but there is a convention in UML to use a message named *create* for this purpose (some use *new*)
- If another message name is used, the message may be annotated with a stereotype, like so: *«create»*
- The *create* message may include parameters, indicating the passing of initial values. This indicates a constructor call with parameters.



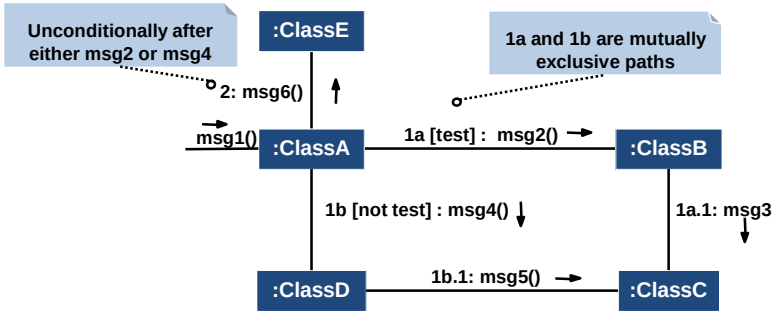
Communication Diagrams: Conditional Messages

- The message is only sent if the clause evaluates to true

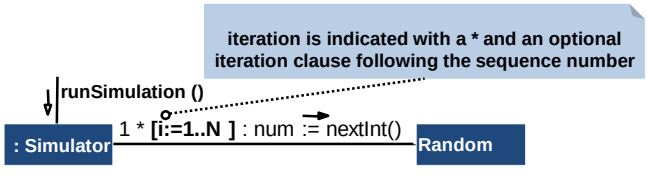


Communication Diagrams: Mutually Exclusive Paths

- Message flows between objects may follow different paths according to some conditions
- In the example, there are two paths according to condition "test" : a or b

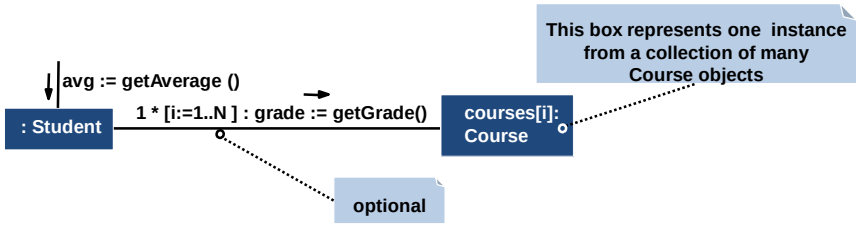


Communication Diagrams: Iteration or Looping



Communication Diagrams: Iteration Over a Collection (Multiobject)

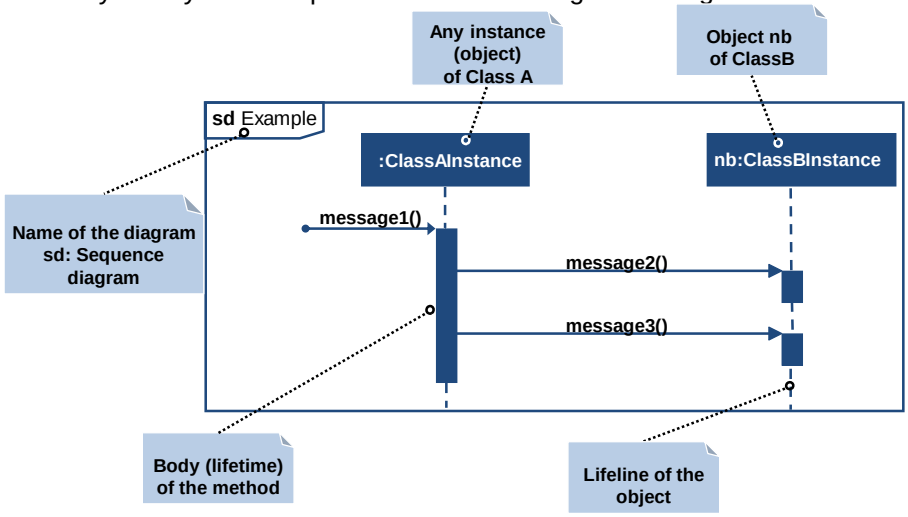
- A common algorithm is to iterate over all members of a collection (such as a list or map), sending a message to each
- In UML, the term "**multiobject**" is used to denote a set of instances



27

Sequence Diagrams

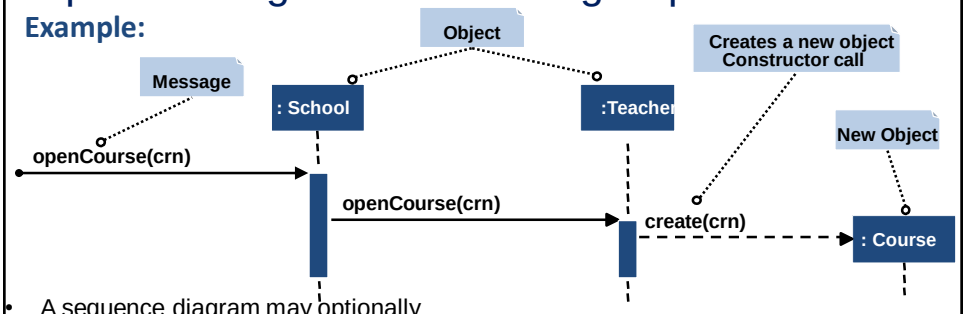
- Sequence diagrams also illustrate the interactions between objects
- They clearly show sequence or time ordering of messages



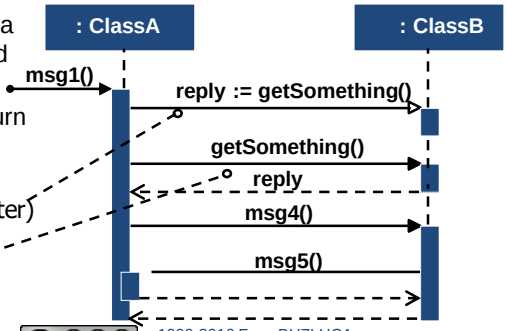
28

Sequence Diagrams: Illustrating Replies or Returns

Example:

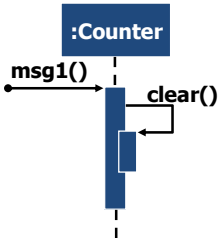


- A sequence diagram may optionally show the return from a message as a dashed open-arrowed line at the end of an activation box
- There are two ways to show the return result from a message:
  1) Using the message syntax: returnVar := message (parameter)
  2) Using a reply (return) message line



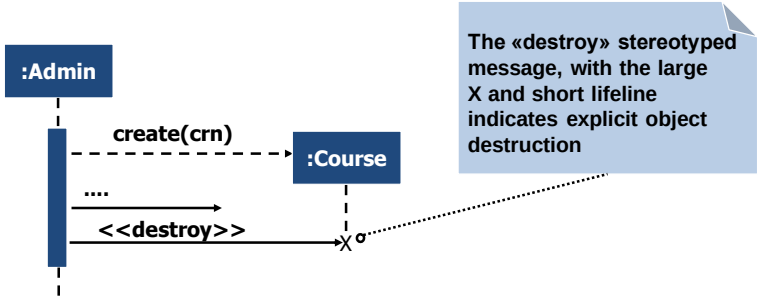
Sequence Diagrams: Messages to "Self" or "This"

- A message can be sent from an object to itself



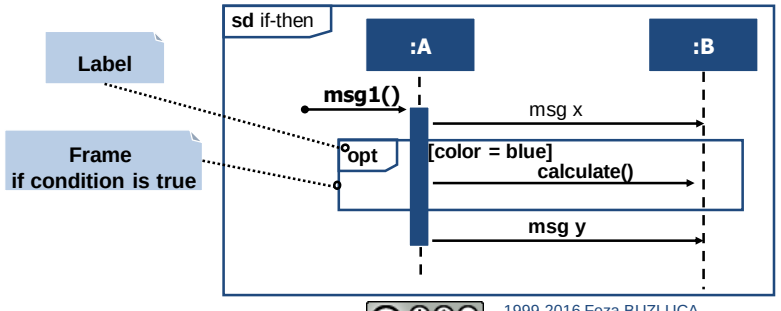
Sequence Diagrams: Object Destruction

- In some circumstances, it is desirable to show explicit destruction of an object (as in C++, which does not have garbage collection)
- In this case, the `delete` operator is used, and the destructor of the target object is called



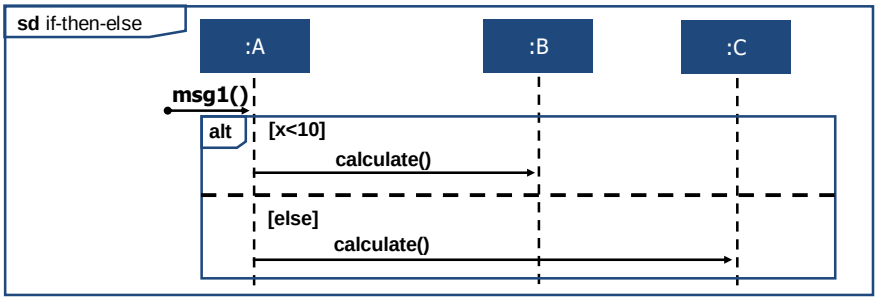
Sequence Diagrams: Conditional Messages

- To support conditional and looping constructs, UML uses frames
- Frames
  - are regions or fragments of the diagrams
  - have an operator or label (such as loop or opt) and a guard (conditional clause)
- In order to illustrate conditional messages, an opt frame is placed around one or more messages

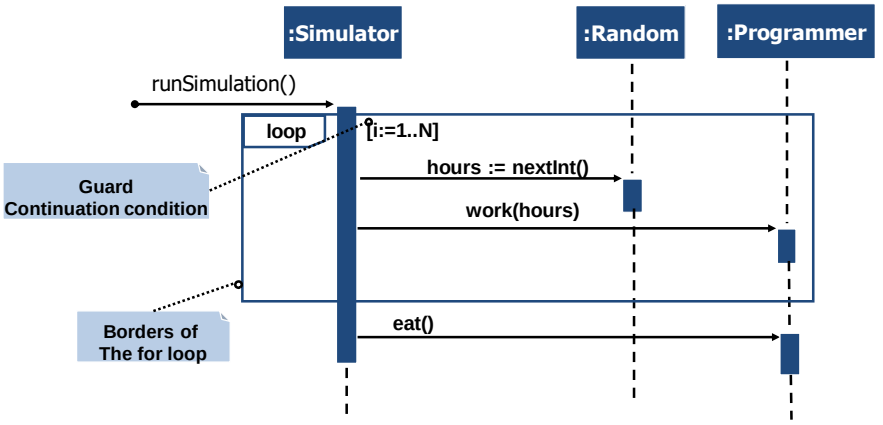


Sequence Diagrams: Mutually Exclusive Conditional Messages

- An alt frame is placed around the mutually exclusive alternatives

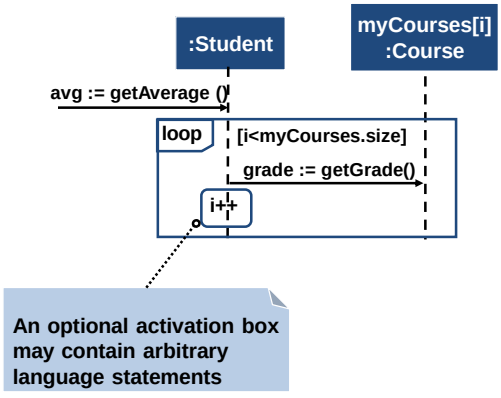


Sequence Diagrams: Looping



Sequence Diagrams: Iteration Over a Collection (Multiobject)

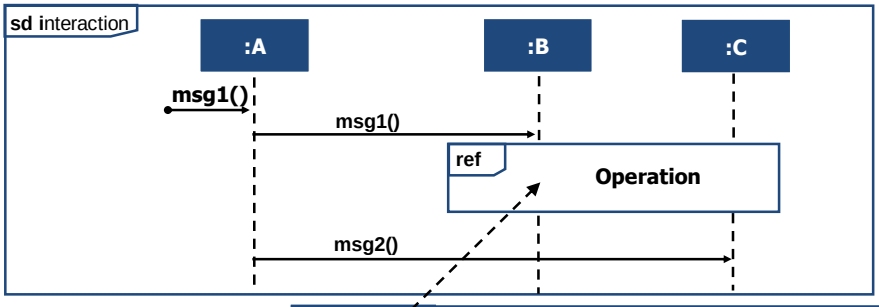
- A common algorithm is to iterate over all members of a collection (such as a list or map), sending a message to each
- In UML, the term "**multiobject**" is used to denote a set of instances



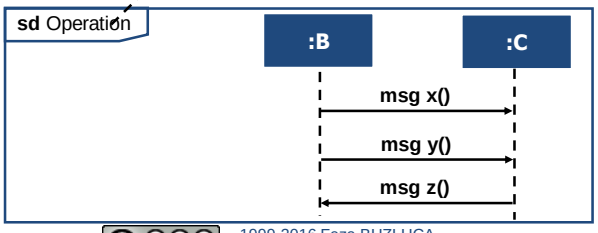
35

Interaction of Diagrams

- Reference frames are used
  - to simplify a diagram and factor out a portion into another diagram, or
  - if there is a reusable interaction occurrence



- It is like calling subroutines



36