Sequence Diagrams and Collaboration Diagrams

Object Oriented Design

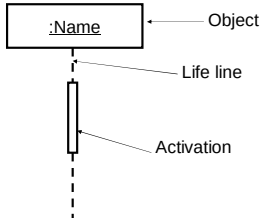
- Design consists of the following steps :
  - Refine the class diagram.
  - Draw the interaction diagrams for the system.
    - Sequence Diagram
    - Collaboration Diagram
  - If objects go through complex state transitions – statechart diagrams
  - Do the above steps iteratively as needed.

Sequence Diagram

- Shows how objects communicate with each other over time.
  - That is, sequence diagrams are used to model object interactions arranged in time sequence and to distribute use case behavior to classes.
  - They can also be used to illustrate all the paths a particular use case can ultimately produce.
- The sequence diagram consists of *Active Objects*, *Messages* represented as solid-line arrows, and *Time* represented as a vertical progression.

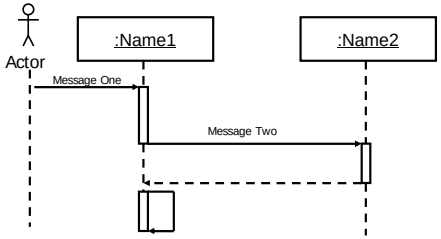
Sequence Diagram - Objects

- A life line illustrates what is happening to an object in a chronological fashion.



Sequence Diagram – Time & Messages

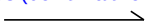
- Messages are used to illustrate communication between different active objects of a sequence diagram.



Types of Messages

- Synchronous (flow interrupt until the message has completed.

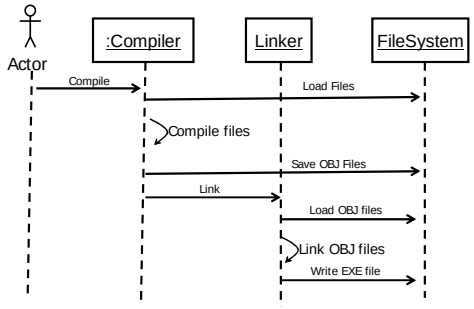
- Asynchronous (don't wait for response)



- Flat – no distinction between sysn/async

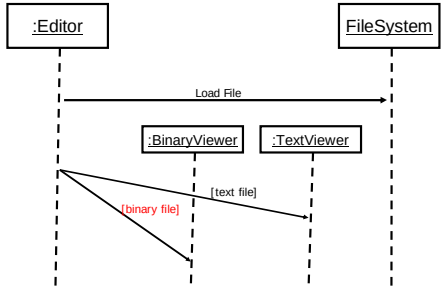
- Return – control flow has returned to the caller.

Sequence Diagram – Compilation



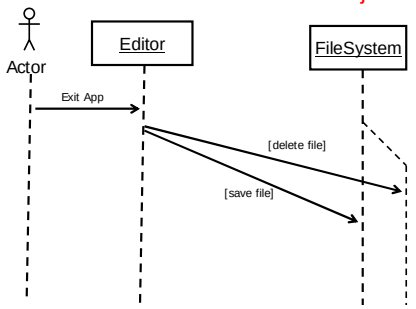
7

Branching Flow: flow goes to different objects [if condition is met]



8

Alternative Flow: flow changes to alternative lifeline branch of the same object



9

Sequence diagram -example

- Use case
  - Add Subject Use Case to URS (University Record System):
- Scenario
  - Scenario 1 : Subject gets added successfully.
  - Scenario 2 : Adding the subject fails because the subject is already in the database.

10

System Design Principles

- System input can take different forms. E.g.
  - From a graphical user interface
  - From a command file
- URS system should be designed such that the functionality can be re-used.
- Command reading and functionality implementation have to be separated.

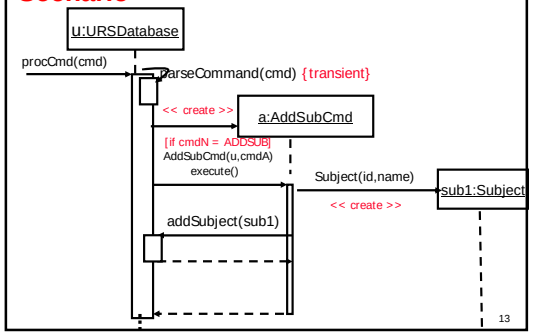
11

Reading from a command file - example

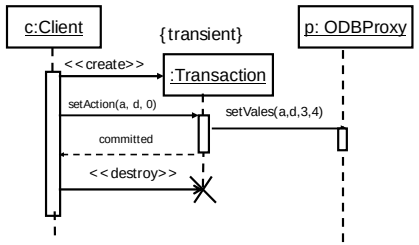
```
class URS{
    public static void main(String[] args){
        URSDatabase u = new URSDatabase();
        //Read command from file;
        while ( not end of file) {
            u.procCommand(cmd);
            //Read next commad;
        }
        //Close file
    }
}
```

12

Sequence Diagram – URS Add Subject Scenario



Creating and Deleting objects

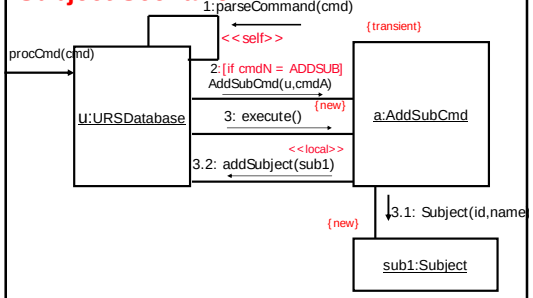


Collaboration Diagrams

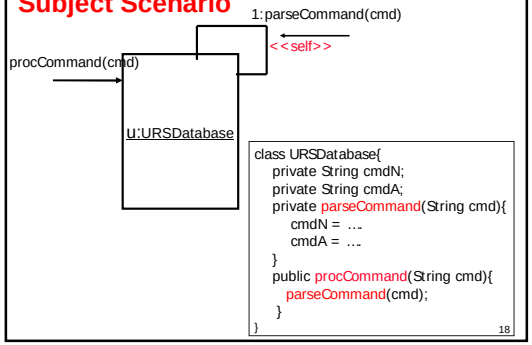
Collaboration Diagrams

- Class diagrams indicates what classes are part of our system, what they offer, how they relate, but they don't tell us how they communicate.
- Collaboration diagrams show (used to model) how objects interact and their roles.
- They are very similar to sequence diagrams
- *Sequence Diagrams* are arranged according to Time.
- *Collaboration Diagrams* represent the structural organization of object
- [Both sequence and collaboration diagrams are called interaction diagrams]

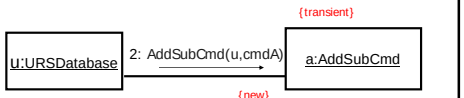
Collaboration Diagram – URS Add Subject Scenario



Collaboration Diagram – URS Add Subject Scenario



Collaboration Diagram – URS Add Subject Scenario

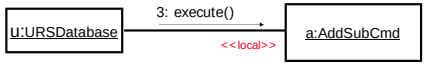



```
class URSDatabase{
  private String cmdN;
  private String cmdA;

  public procCommand(String cmd){
      parseCommand(cmd);
      if (cmdN == ADDSUB){
          AddSubCmd a = new AddSubCmd(u,cmdA);
      }
    }
}
```


Collaboration Diagram – URS Add Subject Scenario

```
class abstract Command {
    protected String cmd;
    protected URSDatabase u;
    public abstract void execute();
}

class AddSubCmd extends Command{
  public AddSubCmd(URSDatabase urs, String cmd){
          u = urs;
          // parse command and set the arguments
  }
  public  void execute(){
     // implement here
  }
}
```

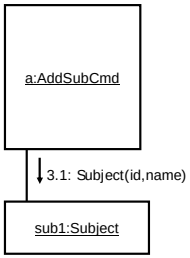

Collaboration Diagram – URS Add Subject Scenario



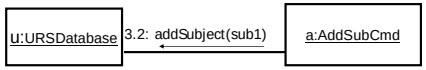

```
class URSDatabase{
  private String cmd;
  public procCommand(String cmd){
      parseCommand(0);
      if (cmd == ADDSUB){
          AddSubcmd a = new AddSubCmd(……);
      }
      a.execute();

    }
}
```


Collaboration Diagram – URS Add Subject Scenario

```
class AddSubCmd{
   URSDatabase u;

  public execute(){


      subject sub1 = new Subject(id,name);

    }
}
```



Collaboration Diagram – URS Add Subject Scenario

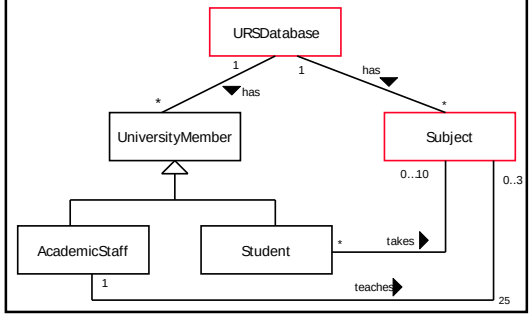



```
class AddSubCmd{
  URSDatabse u;
  public execute(){


      subject sub1 = new Subject(……);
      u.addSubject(sub1);
    }
}
```


Collaboration Diagram – URS Add Subject Scenario

```
class URSDatabase{
  private String cmd;
  private Hashtable subjectHash = new HashTable();
  public procCommand(String cmd){
      parseCommand(0);
      if (cmd == ADDSUB){
          AddSubcmd a = new AddSubCmd(……);
      }
      a.execute();
  }
  public addSubject(Subject sub);
  {
          subjectHash.put(sub.getKey(), sub);
  }

}
```

URS -High Level Class Diagram



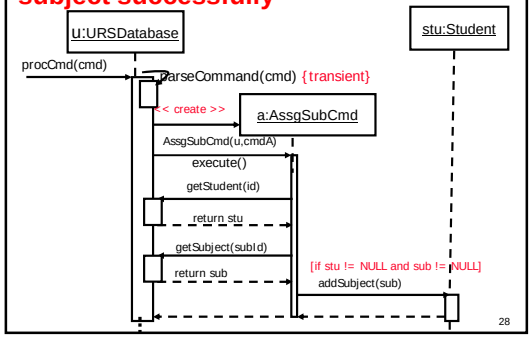
Collaboration Diagrams

- Collaborations Diagrams show transient links that exists between objects.
  - <<self>> - A message from object to itself
  - << local>> - A message sent due to the object begin defined as a local variable in the method.
  - <<parameter>> - The object reference was sent as a parameter to the method.
  - <<global>> The object is global.

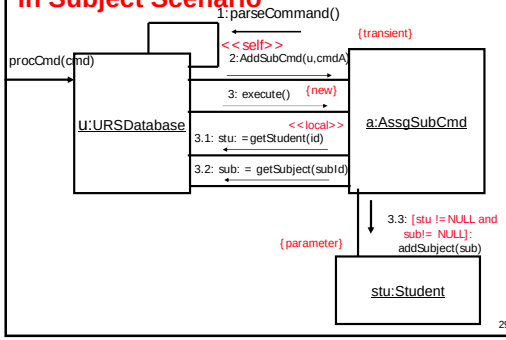
Use Case Vs Scenarios

- Use case
  - Enroll Subject Use Case:
- Scenario
  - Scenario 1 : Student is enrolled for the subject.
  - Scenario 2 : Enrollment fails since the student is already enrolled in 10 subjects.

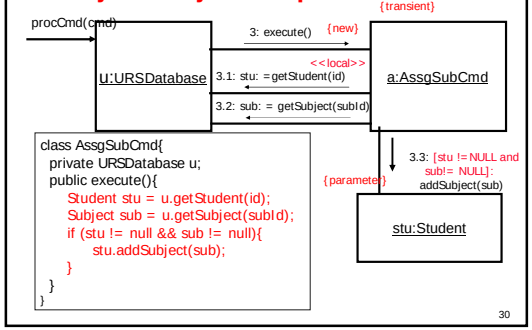
Sequence Diagram – Enroll Student for subject successfully



Collaboration Diagram – Enroll Student in Subject Scenario



Collaboration Diagram – Enroll Student in Subject subject - implementation



Sequence Diagram – Enroll Student for subject - Failure