Interaction Modelling: Sequence Diagrams

Fabrizio Maria Maggi
Institute of Computer Science

(these slides are derived from the book “Object-oriented modeling and design with UML”)
Software Development Methodology

Domain (Class) Model → Interaction Modelling → Application (Class) Model

Code Generation
Software Development Methodology

Instrument for identifying the right interfaces/operations

Domain (Class) Model → Interaction Modelling → Application (Class) Model

Code Generation
Interaction Modelling
Interaction Modelling

How do objects interact?
Interaction Modelling
Software Development Methodology

Domain (Class) Model → Interaction Modelling → Application (Class) Model

Code Generation
Software Development

Domain (Class) Model → Interaction Modelling → Application (Class) Model

Domain Classes; Application Classes (e.g., Patterns); Attributes; Relations; Operations

Code
Application (Class) Model

<table>
<thead>
<tr>
<th>carry</th>
<th>come</th>
<th>drink</th>
<th>go</th>
<th>lead</th>
</tr>
</thead>
<tbody>
<tr>
<td>ask</td>
<td>catch</td>
<td>cook</td>
<td>drive</td>
<td>hit</td>
</tr>
<tr>
<td>bake</td>
<td>clap</td>
<td>cry</td>
<td>eat</td>
<td>hop</td>
</tr>
<tr>
<td>bite</td>
<td>clean</td>
<td>cut</td>
<td>float</td>
<td>juggle</td>
</tr>
<tr>
<td>bounce</td>
<td>climb</td>
<td>dance</td>
<td>fly</td>
<td>jump</td>
</tr>
<tr>
<td>brush</td>
<td>close</td>
<td>dig</td>
<td>fold</td>
<td>kick</td>
</tr>
<tr>
<td>build</td>
<td>color</td>
<td>draw</td>
<td>follow</td>
<td>knock</td>
</tr>
<tr>
<td>call</td>
<td>comb</td>
<td>dream</td>
<td>give</td>
<td>laugh</td>
</tr>
</tbody>
</table>
Application (Class) Model
Application (Class) Model
Application (Class) Model
Interaction Modelling: Detailing Use Cases with Scenarios

**Use Case:** Buy a beverage

**Summary:** The vending machine delivers a beverage after a customer selects and pays for it.

**Actors:** Customer

**Preconditions:** The machine is waiting for money to be inserted.

**Description:** The machine starts in the waiting state in which it displays the message “Enter coins.” A customer inserts coins into the machine. The machine displays the total value of money entered and lights up the buttons for the items that can be purchased for the money inserted. The customer pushes a button. The machine dispenses the corresponding item and makes change, if the cost of the item is less than the money inserted.

**Exceptions:**

*Canceled:* If the customer presses the cancel button before an item has been selected, the customer’s money is returned and the machine resets to the waiting state.

*Out of stock:* If the customer presses a button for an out-of-stock item, the message “That item is out of stock” is displayed. The machine continues to accept coins or a selection.

*Insufficient money:* If the customer presses a button for an item that costs more than the money inserted, the message “You must insert $nn.nn more for that item” is displayed, where $nn.nn is the amount of additional money needed. The machine continues to accept coins or a selection.

*No change:* If the customer has inserted enough money to buy the item but the machine cannot make the correct change, the message “Cannot make correct change” is displayed and the machine continues to accept coins or a selection.

**Postconditions:** The machine is waiting for money to be inserted.

---

*Figure 3.2: Use case description. A use case brings together all of the behavior relevant to a slice of system functionality.*

Interaction Modelling

![Diagram of interaction modelling between a customer and a vending machine. The diagram shows steps such as inserting coins, totaling amount, and choosing beverage. The vending machine has options like buying a beverage, performing scheduled maintenance, making repairs, and loading items. Other roles like repair technician and stock clerk are also depicted.]
The creation of a sequence diagram should be driven by the objective of writing interfaces for the classes of a domain model by reasoning on their interactions in the implementation of the required functionalities.
Sequence Diagrams as a support to define and document interfaces

- A sequence diagram is used to support the definition of the interfaces through the identification of the operations that the classes need to expose for implementing the required functionalities.
- A sequence diagram is used to document where the identified operations come into play in the implementation of the required functionalities.
Sequence Diagrams

Sequence diagrams show operation calls.

1: getAirlines()
2: airports + getAirports()
3: createAirline()
4: getFlights()
5: flights
6: addAirline(airline)
7: airline
8: airlineAdded
9: airlineAdded
10: airlines

UI
City
airport: Airport
flight: Flight
The period of time of an object’s execution is a thin rectangle called **activation** or **focus of control**.

An activation shows the time period during which a call of an operation is processed including the time when the called operation invoke other operations.
Sequence Diagrams

An activation has a call arrow coming into its top and a return arrow leaving its bottom.
The body of the operation is made of all the interactions that occur between the call arrow and the return arrow.
Sequence Diagrams

The period of time when an object exists but is not active is shown as a dashed line.
The entire period of time when an object exists is called **lifeline**
Sequence Diagrams

The notation for a call is an arrow from the calling activation to the activation created by the call.

IN MagicDraw: Message
A return of a call is a dashed arrow from the bottom of the called activation to the calling activation.

**IN MagicDraw: Reply Message**
If an object does not exist at the beginning of the sequence diagram, it must be created. UML shows creation by placing the object symbol at the head of the dashed arrow representing the call that creates the object.
An object can call its own operations (self calls)
Sequence Diagrams

Loops
Advanced Sequence Diagrams: alt

```
bank: Bank  |  theCheck: Check  |  account: CheckingAccount

getAmount()  |  

amount:

getBalance()  |  

balance:

[Balance >= amount]

addDebitTransaction (check: Number, amount)

storePhotoOfCheck (theCheck)

[else]

addInsufficientFundsFee ()

noteReturnedCheck (theCheck)

returnCheck (theCheck)
```
Advanced Sequence Diagrams: opt

`register : RegisterOffice`

`ar : AccountsReceivable`

`drama : Class`

getPastDueBalance (studentId)

pastDueBalance

[pastDueBalance = 0]

addStudent (studentId)

getCostOfClass ()

classCost

chargeForClass ()
Useful material about sequence diagrams can be found on the course website:

https://courses.cs.ut.ee/MTAT.03.083/2016_fall/uploads/Main/MaterialSD
The Entity-Control-Boundary Pattern

**Entities**
Objects representing system data, often from the domain model.

**Boundaries**
Objects that interface with system actors (e.g. a user or external service). Windows, screens and menus are examples of boundaries that interface with users.

**Controls**
Objects that mediate between boundaries and entities. These serve as the glue between boundary elements and entity elements, implementing the logic required to manage the various elements and their interactions. It is important to understand that you may decide to implement controllers within your design as something other than objects – many controllers are simple enough to be implemented as a method of an entity or boundary class for example.

**Four rules apply to their communication:**

1. Actors can only talk to boundary objects.
2. Boundary objects can only talk to controllers and actors.
3. Entity objects can only talk to controllers.
4. Controllers can talk to boundary objects and entity objects, and to other controllers, but not to actors

**Communication allowed:**

<table>
<thead>
<tr>
<th></th>
<th>Entity</th>
<th>Boundary</th>
<th>Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Boundary</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Control</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
The Entity-Control-Boundary Pattern

- Actors interact with boundary objects.
- Boundary objects issue commands to controller objects.
- Controller objects may send queries back to the boundary objects to get more information from the actors.
- Controllers then update entities.
The Entity-Control-Boundary Pattern
The Entity-Control-Boundary Pattern

Functionality to be implemented
The Entity-Control-Boundary Pattern

Most of the calls come from the controller that plays the role of the “orchestrator” in the implementation of the functionality.
The Data Access Object Pattern

- Data Access Object Pattern or DAO pattern is used to separate low level data accessing API or operations from high level business services. Following are the participants in Data Access Object Pattern.
  - Data Access Object Interface - This interface defines the standard operations to be performed on a model object(s).
  - Data Access Object concrete class - This class implements above interface. This class is responsible to get data from a data source which can be database / xml or any other storage mechanism.
  - Model Object or Value Object - This object is simple POJO containing get/set methods to store data retrieved using DAO class.
The Data Access Object Pattern

Step 1

Create Value Object.

*Student.java*

```java
public class Student {
    private String name;
    private int rollNo;

    Student(String name, int rollNo) {
        this.name = name;
        this.rollNo = rollNo;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getRollNo() {
        return rollNo;
    }

    public void setRollNo(int rollNo) {
        this.rollNo = rollNo;
    }
}
```
The Data Access Object Pattern

Step 2

Create Data Access Object Interface.

*StudentDao.java*

```java
import java.util.List;

public interface StudentDao {
    public List<Student> getAllStudents();
    public Student getStudent(int rollNo);
    public void updateStudent(Student student);
    public void deleteStudent(Student student);
}
```
The Data Access Object Pattern

Create concrete class implementing above interface.

StudentDaoImpl.java

```java
import java.util.ArrayList;
import java.util.List;

public class StudentDaoImpl implements StudentDao {

    //list is working as a database
    List<Student> students;

    public StudentDaoImpl(){
        students = new ArrayList<Student>();
        Student student1 = new Student("Robert",0);
        Student student2 = new Student("John",1);
        students.add(student1);
        students.add(student2);
    }

    @Override
    public void deletestudent(Student student) {
        students.remove(student.getRollNo());
        System.out.println("Student: Roll No " + student.getRollNo() + ", deleted from database");
    }

    //retrive list of students from the database
    @Override
    public List<Student> getallStudents() {
        return students;
    }

    @Override
    public Student getStudent(int rollNo) {
        return students.get(rollNo);
    }

    @Override
    public void updatestudent(Student student) {
        students.get(student.getRollNo()).setName(student.getName());
        System.out.println("Student: Roll No " + student.getRollNo() + " , updated in the database");
    }
}
```
The Data Access Object Pattern

Step 4

Use the `StudentDao` to demonstrate Data Access Object pattern usage.

```java
public class DaoPatternDemo {
    public static void main(String[] args) {
        StudentDao studentDao = new StudentDaoImpl();

        // print all students
        for (Student student : studentDao.getAllStudents()) {
            System.out.println("Student: [RollNo : " + student.getRollNo() + ", Name : " + student.getName() + "]");
        }

        // update student
        Student student = studentDao.getAllStudents().get(0);
        student.setName("Michael");
        studentDao.updateStudent(student);

        // get the student
        studentDao.getStudent(0);
        System.out.println("Student: [RollNo : " + student.getRollNo() + ", Name : " + student.getName() + "]");
    }
}
```
From a Domain model to an Application model
From a Domain model to an Application model

Diagram showing class relationships and a sequence diagram for a financial application.
## Application (Class) Model

<table>
<thead>
<tr>
<th>Operations/Interfaces</th>
<th>carry</th>
<th>come</th>
<th>drink</th>
<th>go</th>
<th>lead</th>
</tr>
</thead>
<tbody>
<tr>
<td>ask</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>catch</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cook</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>drive</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lift</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bake</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>clap</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cry</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>eat</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hop</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lock</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bite</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>clean</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cut</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>float</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>juggles</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>look</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bounce</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>climb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dance</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fly</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jump</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>march</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>brush</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>close</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dig</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fold</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>kick</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mix</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>build</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>comb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>color</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>draw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>follow</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>knock</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mop</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>call</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>comb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dream</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>give</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>laugh</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>open</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Invoked on an object of type Bank -> it should be provided in the interface of class Bank

Invoked on an object of type AccountLedger -> it should be provided in the interface of class AccountLedger

Invoked on an object of type CheckingAccount -> it should be provided in the interface of class CheckingAccount
Application (Class) Model
From a Domain model to an Application model
For creating an application model, you start from a domain model where the operations of the classes are not specified and you only have classes representing entities of the domain. Then, through interaction modeling you understand what operations are needed to implement a certain functionality and what additional application classes (outside the domain) are needed to implement it (e.g., Boundary and Controllers).

The application model is the class model obtained by adding operations and application classes to the domain model.
Application Model

• It is possible to add redundant associations to access the data more efficiently
• It is possible to specify the direction of certain associations
Exercise 1: Interfaces?

- A secretary of the CEO office of a company can create a new company in the CompanyManager system with all its parts. Each company is identified by a name. A company consists of departments. Existing departments can be added to the company. When a department is created, it is identified by a name. Each department is located in one or more offices. Each office is in (identified by) a certain address. Each department has a manager and a set of employees. Each manager and employee has a (unique) name and a title. The secretary can add an office, a manager and employees to an existing department. A human resource officer can add employees and a manager to a department (and create them) and promote an employee to a manager. All users can get all the employees with a certain title from a department located at a certain address.
Exercise 2: Interfaces?

- A SchoolManagement system contains the list of all the students of a school. A student is associated to a name, a (unique) student ID and a date of birth. Each student can take at most 6 courses. Each course has a title and one professor. Each professor has a (unique) name, an office number and a consultation time. A professor can teach several courses. The system should provide a functionality to first level administrative officers that given a student returns the list of his/her professors with name, office and consultation time. The same functionality should be provided to a student to retrieve the list of his/her professors. The system should also provide a functionality to first level administrative officers and professors to get all the students registered for a certain course. Together with all the functionalities available for first level administrative officers, the system should allow the second level administrative officers to enroll a student, to register an enrolled student for an existing course, to create a professor, to create a new course assigning it to a professor.
Exercise 3: Interfaces?

- A CinemaBooking system should store seat bookings for multiple theatres. A theatre has a name and an address. Each theatre has seats arranged in rows. A customer can check the availability of seats for a certain show (i.e., the screening of a given movie at a certain time) in one of the theaters. Shows are at an assigned date and time, and scheduled in the theatre where they are screened. Customers can reserve seats for a show and are given a row number and seat number. They may request bookings of several seats. The system stores the customers telephone numbers. Customers can cancel a booking up to 4 hours before the show. Finally, a theater responsible can ask the system administrator to create a new theater and get the telephone number of a customer that has booked a certain seat.
A paper reviewing system can be used to manage several conferences. All the users need to have a registered profile identified by a (unique) name and an affiliation. Each conference has a title and a year and is managed by a chair. A chair can create a conference and add committee members to it. The chair can open the submissions where a deadline for submitting a paper is set. An author can submit a paper with a title before the submission deadline (a paper ID is automatically generated). A conference has several submitted papers, but a paper can be submitted to only one conference. After the submission deadline, the chair can assign a paper to 3 reviewers taken from the committee members. A reviewer can set his/her decision about a paper by changing its status from under review to accepted or rejected. Given a paper ID, the chair and the committee members can retrieve the paper title and the list of authors with their names and affiliations. In order to send a reminder to reviewers, the system should provide a functionality to the chair to get the e-mails of all the reviewers of the papers under review. When the chair closes the reviews the status of a paper becomes accept if the paper has at least 2 reviewers giving the score accept to it or reject if the paper has at least 2 reviewers giving the score reject to it. In order to notify the authors about the final decision on a paper the system should provide a functionality to the chair to get the emails of all the authors of the accepted papers and of all the authors of the rejected papers separately.