UNIVERSIDAD POLITECNICA DE VALENCIA

Seminar

Seminar 3.

System Architecture: Business Logic API (Controller)

Software Engineering

Computer Science School

DSIC – UPV

Introduction

We have adopted a 3-layered architecture to develop the proposed system, **Emergency Calls service**. In the previous session we implemented the domain objects in the business logic layer. Next we will design part of the API offered by the business logic layer to the presentation layer. This boundary is often called the Control API.

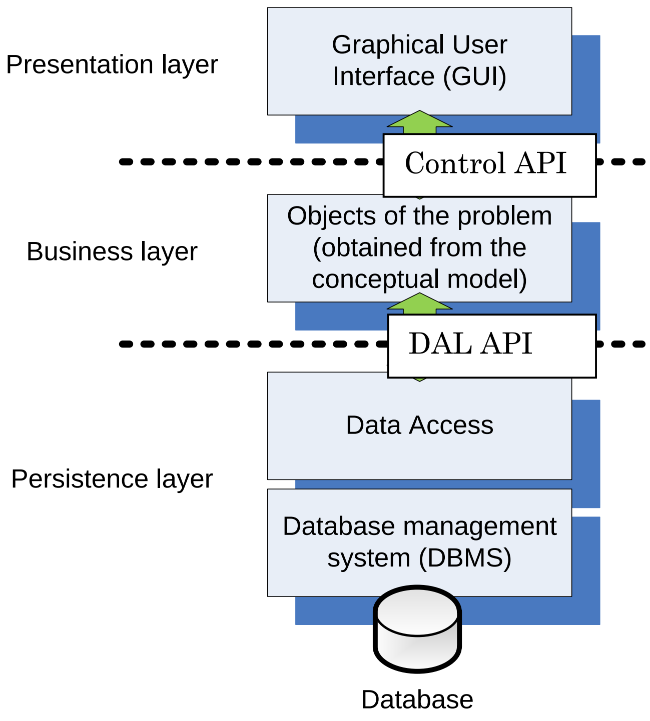
Presentation layer
Graphical User Interface (GUI)
Control API
Business layer
Objects of the problem (obtained from the conceptual model)
DAL API
Persistence layer
Data Access
Database management system (DBMS)
Database

Figure 1. Generic three-layer architecture + API

Figure 1 shows the basic 3-layered architecture with the Control and Data Access Layer APIs.

The Control API is in charge of communicating the presentation and the business logic layers. Given that the interaction with users is described in the use case models we may use them as a reference. However, to implement it, the graphical models are not enough and detailed interaction steps must be described using text templates or, alternatively, sequence diagrams.

Use Cases Description

As an example we will create the control interface for the use cases *CreatePatient* and *ListPatients* which are described in the following text templates:

ID	1
Use Case	Create Patient
Actors	Operator (Initiator)
Goal	Insert a new Patient
Summary	1. The operator provides the patient information: id, first name, last name, address, phone number, age and gender. 2. The system creates a patient and stores all the information.
Precond	--
Postcond	The patient is created and stored

ID	2
Use Case	List Patients
Actors	Operator (Initiator)
Goal	List all patients
Summary	1. The system obtains the list of all patients and provides it to the operator displaying the following information: id, first name, last name, address, phone number, age and gender.
Precond	--
Postcond	--

Implementing the use cases in the controller class

Control class

In our system, the control tasks in the business logic layer will be carried out by our *EmergencyCallService* class. This control class will offer public methods to obtain the collection of patients and to create a patient. This class will be in charge of maintaining "alive" objects in memory and enforce their integrity.

Singleton
To enforce the integrity and prevent unnecessary copies of objects in the business logic layer there must be a single instance of the *EmergencyCallService* class. This can be achieved by using the Singleton design pattern.

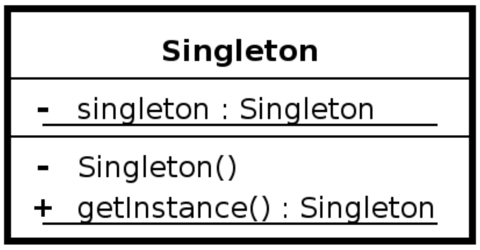

Figure 2. UML representation of the Singleton pattern

This pattern applied to our example would result in the following Java code.

```
public final class EmergencyCallService {
   // single instance
   private static EmergencyCallService INSTANCE = new EmergencyCallService();

   // private constructor
   private EmergencyCallService () {
      // load pre-existing data
      systemLoad();
   }

   // method to obtain the instance class
   public static EmergencyCallService getEmergencyCallService () {
       return INSTANCE;
   }

   public void createPatient(Patient patient) throws BusinessLogicException {
      // search patient
      if (searchPatient(patient.getDni()) == null) {

          // add patient if not found
          // Patients collection defined as a HashMap

          patients.put(patient.getDni(), patient);
      } else
          throw new BusinessLogicException("The patient already exists.");
   }

   public Patient searchPatient(String dni) {
      // Search patient in local collection
      Patient patient = patients.get(dni);
      return patient;
   }

   public List<Patient> listPatients(){
      // obtain patients in the local collection and returns an ArrayList
      return new ArrayList<Patient>(patients.values());
   }

}
```

Exceptions

In the previous code a semantic based on exceptions has been selected to inform about errors in the business logic layer when invoking services of the controller. In particular, we have created a class called `BusinessLogicException` to capture exceptions in the business logic layer.

Loading pre-existing data

Some data which is pre-existing will be pre-loaded in memory when the controlled object is created. In our example, hospitals and ambulances will be created in this phase by implementing a method called *systemLoad()*. At this moment, because we have no persistence layer, we will create some example objects manually.

```
private void systemLoad() {
   // This method would load pre-existing objects from the database.
   // Now we create them manually because there is no persistence layer
   // implemented

   // We create two hospitals
   Hospital h1 = new Hospital("Costa del Azafrán", "Avenida de la Paella, 1",
45.0, 15.0);
   Hospital h2 = new Hospital("Hospital Psiquiátrico El Cuco", "Calle del
Limbo, 33", 46.0, 14.0);
   // We create a private ambulance
   Private a1 = new Private("12345 KIS", "UCI Movil", 44.0, 13.9, "Ambulancias
Privadas S.L.");
   // We create a hospital based ambulance
   HospitalBased a2 = new HospitalBased("6789 DRY", "Sin Equipamiento", 37.0,
14.5, h1);

   hospitals.add(h1);
   hospitals.add(h2);
   ambulances.add(a1);
   ambulances.add(a2);
}
```

Testing:

In order to test the methods of the controller class we will implement a tester class that will use the methods that will be invoked in the future by the presentation layer. This tester class will display the results on the console.

```
public class BusinessLogicTester {

public static void main(String[] args) {
   try {
      // The controller object is created
EmergencyCallService ecs= EmergencyCallService.getEmergencyCallService();


      // A patient is added
      ecs.createPatient(new Patient("10123456A", "Juan", "Martinez
Gandia","Calle Santiago, 4 Valencia", 123453250, 50, 'H'));

      List<Patient> patients = ecs.listPatients();
      for (Patient pat : patients)
         System.out.println(" DNI: " + pat.getDni() + " First Name: " +
pat.getFirstName() + " Last Name: "    + pat.getLastName() + " Address: " +
pat.getAddress() + " Phone: " + pat.getPhone()+ " Age: " + pat.getAge() + "
Gender: " + pat.getGender());
}
```

IMPORTANT NOTE: The control class MUST NOT display any message on the screen. All the messages are displayed on the console by the *BusinessLogicTester* class.