1. Three-Tier (or three-layer) Architecture

Software architecture is the description of the subsystems and computational components of a software system and the relationships among themselves. It is the outcome of the software architectural design activity. For this activity, design patterns are commonly used; each one of them is suitable for a type of problem and has specific features.

Tier design (or style) pattern has the great advantage of facilitating both changeability (that is to say: extensibility, portability, maintainability and restructurability) and testing. As a disadvantage, we point out that it makes difficult to obtain the optimum efficiency since it adds unnecessary or redundant work.

The calls (and return) in a tier pattern must comply with the following constraints:
- The components (objects and classes) are grouped into tiers.
- The communication (relationships between classes, messages between objects) only takes place between elements from the same tier of from contiguous tiers.

Three-Tier Architecture follows this diagram:

The presentation tier knows how to present the data to the user, but it ignores which transformations have to be made in order to reply to the requests of the user.

1. It is connected to users: by receiving events and by presenting them replies and results.
2. It is connected to the domain tier: by transferring the external events (calls to actions) and queries to it, and by receiving the replies and results.
3. It is in charge of:
   - Finding out about the users’ requests.
   - Ordering the execution of actions.
   - Communicating the results of the actions to the users.
Dealing with windows, buttons, dialogues, menus and lists.

The domain tier knows how to satisfy the requests of the user, but it ignores where the data are stored and how they are presented to the user.

1. It is connected to the presentation tier: by transferring the replies and results to it and by receiving the external events (calls to actions) and queries.
2. It is connected to the data management tier: by transferring query and data modification operations and by receiving replies and results.
3. It is in charge of:
   - Finding out about the events.
   - Controlling its validity.
   - Changing the state of the domain.
   - Executing the assigned actions.
   - Finding out about the queries.
   - Obtaining the result.
   - Communicating the reply.

The data management tier knows where and how the data have been stored, but it ignores how to deal with them.

1. It is connected to the domain tier: by transferring the replies and the results to it, and by receiving query and data modification operations.
2. It is connected to the databases or files management system: by transferring the query operations and data modifications in the appropriate format and language, and by receiving the replies and results.
3. It is in charge of:
   - Allowing the domain to ignore where the data are.
   - Making possible that certain objects of the domain are persistent.

Through this structure you can achieve:

- That a change in the persistent data representation (for example, a change in the databases or files management system), normally, only affects the data management tier (we say normally because the suitable structure for a solution with certain database managers may affect the domain tier).
- That a change in the program interface (for example, a change in the windows system or in the peripherals used for communicating with the user) only affects the presentation tier.
- That the domain tier, which encapsulates most of the program logic, is quite independent of the platform changes, operative system changes, etc.

2. Three-tier program structure: additional concepts

- Many times, it will be advisable to have at our disposal classes whose function is bringing together other classes, coordinating the functionality (included in the methods) and unloading the functionality itself from the agglutinated classes. These classes are called controllers.

It is advisable that only the most basic operations (queries and updating functionalities, as well as other very typical functionalities such as discounts on sales) in addition to the data (attributes) appear in the model classes (or program domain), since it makes reusing easier. Therefore, we will try to put the most specific functional aspects of the application aside or, in other words, in controllers. As a result, many times the logic of the functionality/ies (use case) will be in the controllers.
If we think again about reusing, but in this case the views, separating the appearance of the view (screen or listing design) from the methods that manage them and that monitor the behaviour of the interface will be more practical. Therefore, it will be advisable to place the latter in controllers.

Consequently, controllers may appear both in the presentation tier and in the domain tier. The main program itself is usually a controller of the presentation tier that offers the functionality of the program to the user. Therefore, its structure is:

- Remember that the *links* (in associative relationships) must be implemented only in the direction where one intends to navigate.
- In order to guarantee that a homogeneous validation of the attributes is always carried out, it is advisable to centralize it, that is, that the updating will be made with an only specialized method. Moreover, this makes the maintenance of this validation easier. For this reason, hiding the attributes, by making them private (only visible from the object itself), is advisable as well as generating for each one of them the following two methods:

  ```java
  public boolean set_attribute_x (type_of_x new_value) {
    if (<filter>) {
      attribute_x = new_value; // Attention to type_of_x
      return true;
    }
    else
      return false; // if
  }
  ```

- It is better to place the filters of the values of the elements of the model (data consistency) in the structure of the model. Therefore, it is advisable that the method `set_attribute_x` checks the validity of the supplied value and returns information about this validity (in this case it returns true when the value of `new_value` is valid):

  ```java
  public boolean set_attribute_x (type_of_x new_value) {
    if (<filter>) {
      attribute_x = new_value; // Attention to type_of_x
      return true;
    }
    else
      return false; // if
  }
  ```
Any class that uses this method will use the filter (which must be a boolean function that returns true when the value of new_value can be accepted as a value of attribute_x), so it will not be necessary to check the new value of the attribute anymore.

- The persistence or disc manager usually has a data structure parallel to the structure of the model, appropriately adapted to properly store the links that implement both associations and compositions. Storing the links of an object may be responsibility either of the same operation that stores the object or there can be specific operations to store the links. Using them in order to maintain the model updated is the programmer responsibility.

3. Three-tier structure example: a CRUD

Let’s take a program that ensures the maintenance (creations, retrievals, updates and deletions, that is, the CRUD) of the passengers of a flight as an example of three-tier architecture. It could be a part of the management program of a travel agency.

Despite the most probable is that your program is ruled by events, we consider that an example where one can use the three-tier architecture together with a sequential behaviour, where the program is the one that takes the initiative of the dialogue and the control of the execution flow, will be clearer. For this reason, the program we use as an example belongs to this type. We will start from the following use case:

Let’s assume that screens have been designed to choose the maintenance option (Passengers Management use case) and to read the data of a passenger (Passenger Registration and Passenger Modification use cases). Let’s also assume that its control has been delegated to two classes VistaMenuMant and VistaMantPassatger respectively.
The presentation tier of the maintenance of passengers could be implemented in Java with a controller class `mant_passatger` like this one:

```java
public class mant_passatger // controller of the presentation tier
{
    VistaMenuMant VMM;
    VistaMantPassatger VMP;
    ControladorDominiMantPassatger CDMPS;

    public mant_passatger ()
    {
        char OPC;
        VMM = new VistaMenuMant ();
        VMM.visualitzar ();
        OPC = VMM.preguntar ("What option do you want?", '1', '2', '3', '4', '0');
        while (OPC != '0')
        {
            switch (OPC)
            {
                case '1': alta ();
                          break;
                case '2': baixa ();
                          break;
                case '3': modifica ();
                          break;
                case '4': consulta ();
                          break;
            } // switch
            VMM.visualitzar;
            OPC = VMM.preguntar ("What option do you want?", '1', '2', '3', '4', '0');
        } // while
    } // mant_passatger ()

    private void alta ()
    {
        char OPC;
        id_pass ID;
        passatger PS;
        VMP = new VistaMantPassatger ();
        CDMPS = new ControladorDominiMantPassatger ();
        VMP.visualitzar ();
        ID = VMP.LlegirId (); // it’s obtained and created (it may validate the
```
syntax of ID)
    while (ID != null)
    {
        if (CDMPS.existent(ID))              // CDMPS will use the data manager
            VMP.missatge ("Already existent passenger");
        else
            {
                PS = new passatger ();
                if (PS.set_ID (ID))
                {
                    obtenirLaRestaDeDadesDelPassatger (PS);    // PS is obtained
                    and validated
                    VMP.mostrar (PS);
                    OPC = VMP.preguntar ("Confirm (Y/N): ", 'Y', 'N');
                    if (OPC = 'Y')
                        CDMPS.afegir (PS);    // It communicates the validation to
                        // the user, it can then be added,
                        // CDMPS uses the data manager if
                        // needed
                            // if
                    } else
                        VMP.missatge ("non valid passenger code");
                        // if
                } else
                    VMP.visualitzar ();
                ID = VMP.LlegirId (); // it’s obtained and created (it may validate the
                            // syntax of ID)
            } // while
    } // alta ()

private void obtenirLaRestaDeDadesDelPassatger (passatger PAS)    // very simple
    {
        String nom, adreça, ...;
        ...;
        boolean valid;
        while (!valid)
        {
            nom = VMP.read_nom ();
            valid = PAS.set_nom (nom);
            adreça = VMP.read_adreça ();
            valid = PAS.set_adreça (adreça) && valid;
            ...
        } // while
        // obtenirLaRestaDeDadesDelPassatger ()
    }

4. The power of generalization: a generic CRUD

When we make another CRUD, for example, of flights or of customer orders, a big part of the logic
from the previous one will be the same. Mainly, it is the name and the structure of the objects to be
maintained that change. It is all about taking advantage of the characteristics of an object-oriented lan-
guage in order to program just once what is common to all CRUDs.

Particularly, the critical point is that the class of objects to be maintained must be a parameter or an ar-
argument of the classes that implement the generic CRUD.

- There will be a generic controller class for all the maintenances (mant) from which all the main-
tenance controllers will inherit. It will have a parameter which will be the class of domain entity
to be maintained. According to the logic of the use case, all the domain entities to be maintained must have some kind of identifier.

- It will be necessary to modify the class \texttt{VistaMenuMant} to be able to transfer a text (\textit{string}) with the name of the class of objects of the model (subclass of \texttt{EntitatDomini}) we want to maintain in every case.

- An abstract class which will generalize \texttt{VistaMantPassatger} from \texttt{Passenger} to any domain entity will be necessary too. Ought to the fact that the data to be read and presented depend on the class of entity itself, \texttt{VistaMantPassatger} will have to inherit from an abstract and deferred class.

- Finally, we will have to do the same with the identifier, with the maintenance controller of the domain tier, and with the persistence manager: generalizing them so that we will be able to use them for other entities of the domain.
We could obtain a possible implementation from Java version 1.5 by modifying what we had previously made in this way (we have highlighted what has to be deleted and what has to be added in this way in order to help you follow it):

```java
public abstract class mant_passatger<T extends EntitatDomini> {
    VistaMenuMant VMM;
    VistaMantPassatgerED VMED;
    ControladorDominiMantPassatgerED CDMED;
    T ed;
    String nomEntitat;
    Identificador ID;

    public mant_passatger(VistaMantED ve, T e, ControladorDominiMantED cd,
                          String nomE, Identificador ide) {
        char OPC;
        nomEntitat = nomE;
        VMED = ve;
        ed = e;
        CDMED = cd;
        ID = ide;
        VMM = new VistaMenuMant(nomEntitat);
        VMM.visualitzar();
        OPC = VMM.preguntar("What option do you want?", '1', '2', '3', '4', '0');
        while (OPC != '0') {
            switch (OPC) {
            case '1': alta ();
                break;
            case '2': baixa ();
                break;
            case '3': modifica ();
                break;
            case '4': consulta ();
                break;
            } // switch
            VMM.visualitzar;
            OPC = VMM.preguntar("What option do you want?", '1', '2', '3', '4',
                                 '0');
        } // while
    } // mant_passatger()

    private protected void alta () {
        char OPC;
        id_pass_ID;
        passatger PS;
        VMP = new VistaMantPassatger();
        CDMPS = new ControladorDominiMantPassatger();
        VMED.visualitzar();
        ID = VMED.LlegirId (); // it’s obtained and created (it may validate the syntax of ID)
        while (ID != null) {
            if (CDMPS.ed.existent(ID)) // CDMPS will use the data manager
                VMED.missatge("Already existent passatger" + nomEntitat);
            else {
                PS = new passatger();
                if (PS.ed.set_ID (ID))
                    { obtenirLaRestaDeDadesPassatger (ed); // PS ed is obtained and validated
```
abstract void obtenirLaRestaDeDades (T ed1)
{}

That is, it would be like this:

```java
public abstract class mant<T extends EntitatDomini>
{
    VistaMenuMant VMM;
    VistaMantED VMED;
    ControladorDominiMantED CDMED;
    T ed;
    String nomEntitat;
    Indentificador ID;

    public mant (VistaMantED ve, T e, ControladorDominiMantED cd, String nomE, Indentificador ide)
    {
        char OPC;
        nomEntitat = nomE;
        VMED = ve;
        ed = e;
        CDMED = cd;
        ID = ide;
        VMM = new VistaMenuMant (nomEntitat);
        VMM.visualitzar ();
        OPC = VMM.preguntar ("What option do you want?", '1', '2', '3', '4', '0');
        while (OPC != '0')
        {
            switch (OPC)
            {
                case '1': alta ();
                    break;
                case '2': baixa ();
                    break;
                case '3': modifica ();
                    break;
                case '4': consulta ();
                    break;
            } // switch
            VMM.visualitzar;
            OPC = VMM.preguntar ("What option do you want?", '1', '2', '3', '4', '0');
        }
    }
```
protected void alta ()
{
    char OPC;
    VMED.visualitzar ();
    ID = VMED.LlegirId (); // it’s obtained and created (it may validate the syntax of ID)
    while (ID != null)
    {
        if (CDMED.existent(ID)) // CDMED will use the data manager
            VMED.missatge ("Already existent" + nomEntitat);
        else
        {
            if (ed.set_ID (ID)
            {
                obtenirLaRestaDeDades(ed); // obtains and validates ed
                OPC = VMED.preguntar ("Confirm (Y/N): " , 'Y', 'N');
                if (OPC = 'Y')
                    CDMED.confirmat (ed); // It communicates the validation
                    // to the user, it can then be
                    // added, CDMED uses the data
                    // manager if needed
            } // if
            else
                VMED.missatge ("non valid " + nomEntitat + " code");
        } // if
    } // while
} // alta()

abstract void obtenirLaRestaDeDades (T ed1)
{;
} // mant

Therefore the passenger maintenance controller would be like:

public class mant_passatger extends mant<passatger>
{
    public mant_passatger ()
    {
        super (new VistaMantPassatger (), new passatger (),
        new ControladorDominiMantPassatger (), "passenger",
        new id_pass ());
    } // mant_passatger ()

    ...
5. Another step towards generalization: views with dynamic fields

A view such as VistaMantED will be deferred because there are two parts of its functionalities that depend on the entity to be maintained (generic parameter): the fields that appear on the screen and the data to be returned to the domain through the operations read_attribute.

In fact, when VistaMantED is specified for a particular entity of the domain (such as Passenger), what has to be made for every field to be maintained is:

1. Adding a line to the service in charge of presenting the form which displays on the screen the name of the field and (in case of updates) its current value.
2. Adding a line to the service responsible for reading the inputs of the user to read the field.
3. Define a service read_attribute so that the functionality obtenerLaRestaDeDades can obtain the read value.

This makes that modifying a view can be tedious, since any change in the set of fields of the entity makes changing at least three points of the class compulsory. Next, we show you a solution to get a change in the field to be maintained to be made only in one point of the view class.

Every field that has to be maintained by the view is represented through an object of the class Dada. Typically, the aggregation of Dada to VistaMantED is usually implemented with a table (array) since in this way data manipulation is easier. The different types of data can be obtained through inheritance.

The code of the view is simplified because

1. Presenting a form is limited to go through (by means of an iteration) the list of aggregated Dada and to display them on the screen.
2. Reading the data from a form can be made in the same way through an iteration.
3. The pass of data between the View and the Domain is limited to the transfer of an object of the domain entity.
In this case, the service `obtenirLaRestaDeDades` would also be implemented according to the following:

```java
private void obtenirLaRestaDeDades (EntitatDomini ed)
{
    VMED.agafarDades (ed);  // It must be deferred (abstract) in VistaMantED
} // obtenirLaRestaDeDades ()
```

The method `agafarDades` is an iteration that takes every datum from `VMED` and makes `sets` in the entity `ed`, so that it includes the validation and, therefore, this fact makes necessary to treat the exceptions and notify them to the user. In this case, when either updates or deletions are made, the sequence will be different:

```java
ID = VMED.llegirId ();
ed = CDMED.recuperar (ID);
VMED.mostrar (ed);
VMED.modificaEntitat (ed);
```

In this last section only one behaviour scheme is described, we are not talking about the detailed algorithm, for this reason we have neither specified all the necessary parameters and their implications nor checked if the ID matches an existing passenger.

### 6. A few notions

- The controllers of the presentation tier are linked to the use cases, there is one per case at the most. Use cases from a “family” are tried to be assembled in a controller. For example, the CRUD (creation, retrieval, update and deletion) of a class.
- Persistence managers (PM) or disc managers are linked to the persistent classes of the model. Usually, there will be one per every persistent class.
- The views are linked either to the menus or to the use cases. We’ll try to use the views for more than one use case.
- A controller of the domain tier will aggregate as many model classes (together with their respective PMs) as necessary to implement their use cases.
- Moreover, it is always advisable to generalize the common characteristics of the classes. For example, the specific persistence manager of every class may inherit all the characteristics which are common to all the persistence managers from an abstract manager.
- In the presentation tier, at the most, the syntactical validity of the data the user introduces will be controlled.

### 7. The controllers of the domain tier

The task of the domain controllers is to unload the model classes of those responsibilities (methods or attributes) that are specifically needed for this program and that, therefore, are not a part of their nature. We want to take them out from the model classes since they would make their reuse difficult, and they cannot be transferred to the presentation controllers because they exclusively have to do with transfor-
mations of the domain and not with the dialogue with the user. Usually these responsibilities are the necessary methods for implementing the use cases, in addition to the data manager class itself in case of being persistent. The aggregation to the model class (domain tier) of the class that manages its persistence (data management tier) would load it, furthermore, with the specific technological characteristics of this program’s environment.

If we continue with the same management program of a travel agency we could suggest the implementation of plane ticket booking and sale. At least, the passenger, flight, and ticket model classes would be involved. A diagram of the implementation of this use case, containing at least the most important elements, could be as follows: