Fundamentals of multilayer architecture

Intelligent Infrastructure Design for the Internet of Things

Antonio Navarro

Index

- References
- Introduction
- Single-layer architecture
  - Features
  - Advantages and disadvantages
- Two-layer architecture
  - Features
  - Advantages and disadvantages
  - Related patterns

Index

- Multilayer architecture
  - Features
  - Advantages and disadvantages
  - Related patterns
- Front controller pattern
- Application Controller Pattern
- MVC Pattern

Index

- Transfer pattern
- *Data Access Object* Pattern
- Application service pattern
- Pattern subject of business
- Domain store pattern
- Conclusions

References

- Alur, D., Malks, D., Crupi. J. *Core J2EE Design Patterns: Best Practices and Design Strategies. 2nd Edition.* Prentice Hall, 2003
- Fowler, M. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002
- Java Platform Enterprise Edition 7 Tutorial https://docs.oracle.com/javaee/7/tutorial/index.html

Introduction

- In this topic we will look at the fundamentals of *multilayer architecture.*
- It is a basic architecture for the design of applications with important integration needs.

Introduction

- We will not worry about obtaining good designs at the component level of each layer.
- This is dealt with by disciplines such as:
  - Human-computer interaction
  - Programming
  - Design patterns
  - Databases

Introduction

- We will take care of quality from an *architectural point of view*.
- *Architecture** is the fundamental organization of a system, expressed in its components, their relationships with each other, and in the environment and principles that guide its design and evolution.

IEEE Std 1471-2000 IEEE Recommended Practice for Architectural Description of Software-Intensive Systems

Introduction

- In particular, we will look at the multilayer architecture and its fundamental patterns.
  - The catalog contains 21 patterns
  - We will see a basic multilayer with 4 patterns
- Although these patterns are drawn from web engineering*, they are also applicable to non-web applications.

*http://www.corej2eepatterns.com/index.htm

Introduction

- According to Christopher Alexander, "a *pattern* describes a problem that occurs over and over again in our environment, as well as the solution to that problem in such a way that you can apply this solution a million times, without doing the same thing twice".

Introduction

- Although Alexander was referring to patterns in cities and buildings, what he says is also valid for OO design patterns.
- We can say that the design patterns:

  - They are simple and elegant solutions to specific OO software design problems.

  - They represent solutions that have been developed and have evolved over time.

Introduction

- Design patterns do not take into account issues such as:
  - Data structures
  - Domain-specific designs
- They are descriptions of classes and related objects that are particularized to solve a general design problem in a given context.

Introduction

- Each design pattern identifies:
  - Participating classes and instances
  - Roles and collaborations of these classes and instances
  - The distribution of responsibilities

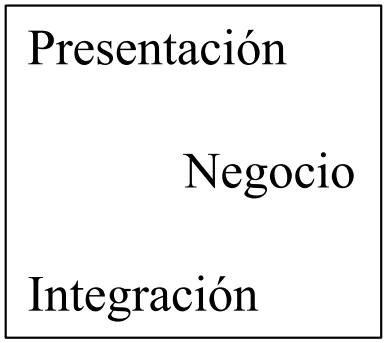
Introduction

- Some pattern sources:
  - GRASP*, by Craig Larman
  - *Gang of Four* (GoF) patterns, by Eric Gamma et al.
  - *Core J2EE patterns*, by Alur et al.
  - *Patterns of Enterprise Application Architecture*, by Fowler et al.
  - *SOA Design Patterns*, by Thomas Erl

General *Responsibility Assignment Software Patterns*

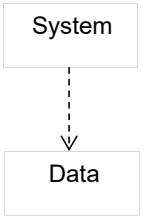
Single-layer architecture
Features

- The single-layer architecture does not divide the system into presentation, business and integration.



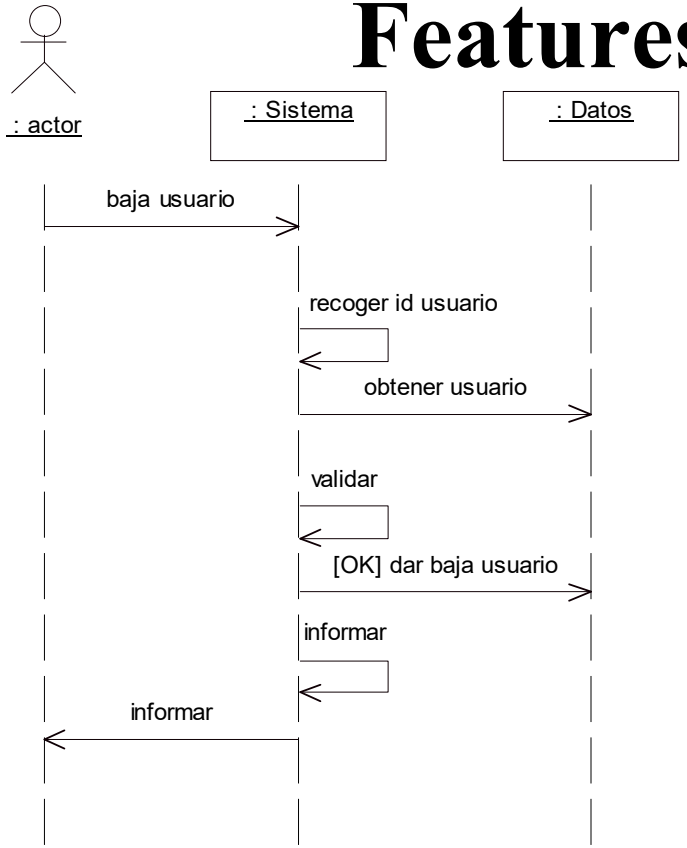
Arquitectura de una capa

Single-layer architecture
Features



System classes

Single-layer architecture Features



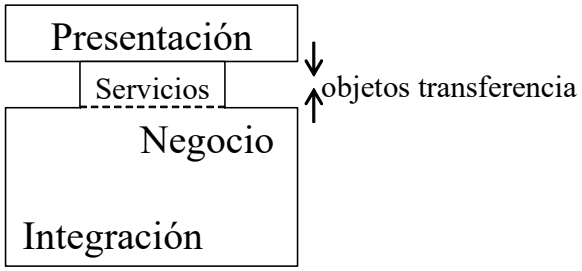
System behavior

Single-layer architecture
Advantages and disadvantages

- Advantages
  - Conceptual simplicity
- Inconveniences
  - Neither the user interface, the business logic nor the data representation can be modified without affecting the other layers.
  - Factual complication

Two-layer architecture
Features

- The two-layer architecture differentiates between the presentation layer and the rest of the system.
- Does not differentiate integration business

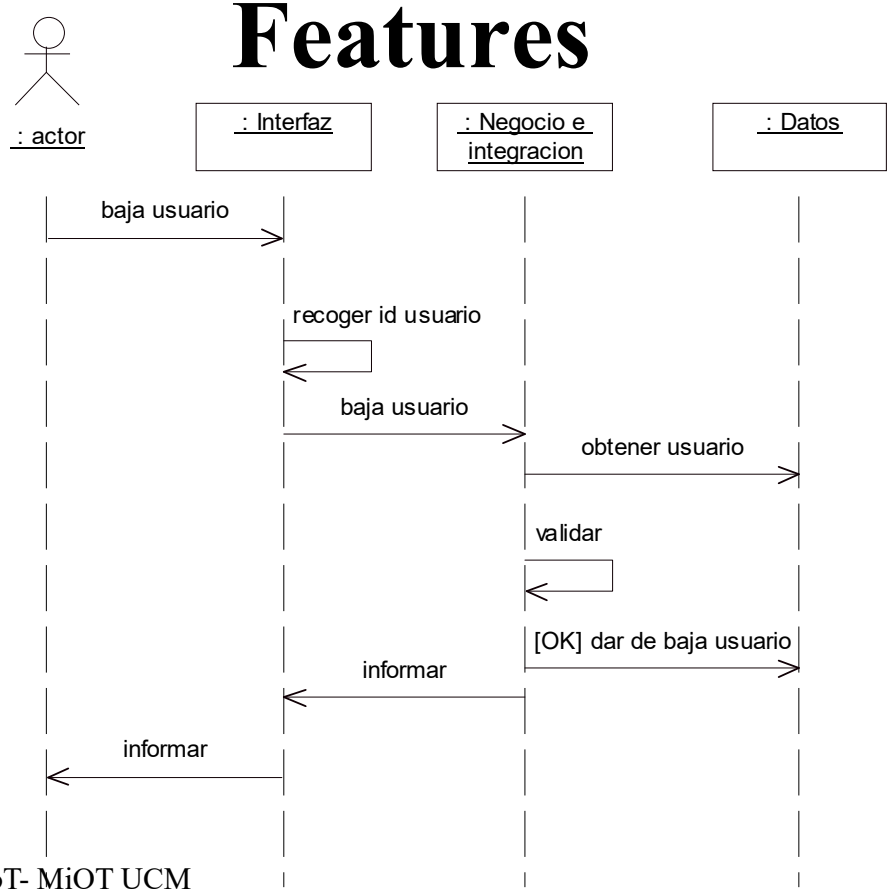


Arquitectura de dos capas

Two-layer architecture Features

System classes

Two-layer architecture Features



System behavior

Two-layer architecture
Advantages and disadvantages

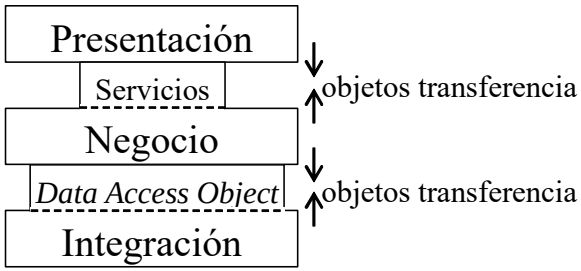
- Advantages
  - Allows changes to the user interface or to the rest of the system without mutual interference
  - Factual simplicity
- Inconveniences
  - More architectural complication than single-layer architecture
  - Cannot change business logic or data representation without mutual interference

Two-layer architecture
Related patterns

- Although not strictly necessary, it is often used:
  - MVC

Multilayer architecture Features

- The multilayer architecture considers a presentation layer, a business layer, and an integration layer.



Arquitectura multicapa

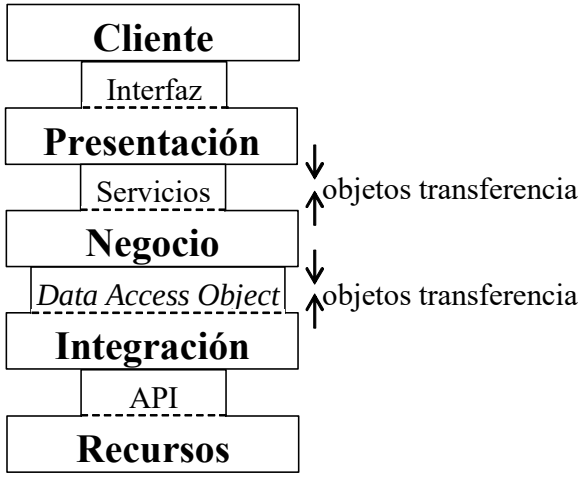
Multilayer architecture
Features

- The *presentation layer* encapsulates all the presentation logic required to serve clients accessing the system.
- The *business layer* provides the system services
- The *integration layer* is responsible for communication with external resources and systems.

Multilayer architecture
Features

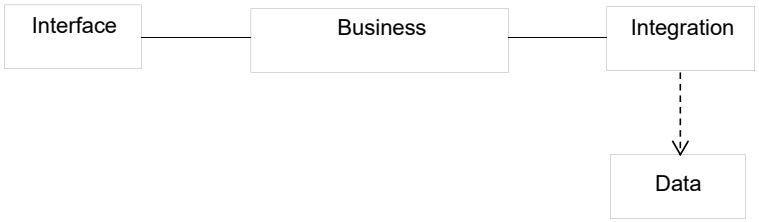
- In reality, the architecture is *five-tiered*, as it includes the client and resource layers
- The *client layer* represents all the devices or clients of the system that access the system. It is above the presentation layer
- The *resource layer* contains the business data and external resources. It is under the integration layer

Multilayer architecture Features



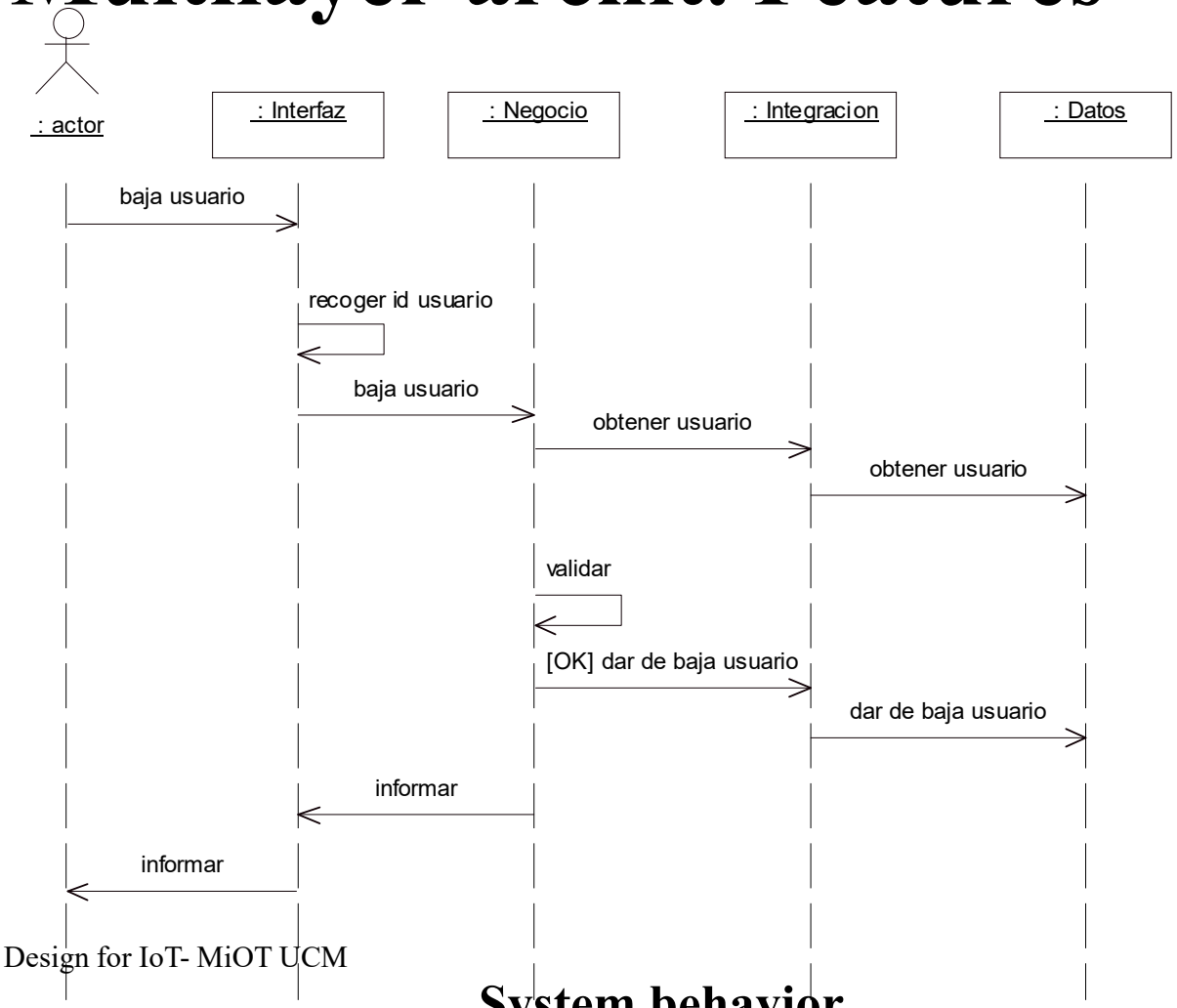
Arquitectura multicapa

Multilayer architecture Features



System classes

Multilayer archit. Features



System behavior

Multilayer architecture
Features

- Note that these are *logical* layers
- *Physical* layers are another matter
- Thus, the web presentation layer and the business logic could be on the same machine or on different machines.

Multilayer architecture Features

- Advantages
  - Any layer can be modified without affecting the others.
  - Factual simplicity?
- Inconveniences
  - Increased architectural complexity

Multilayer architecture Features

- Advantages:
  - Integration and reusability
  - Encapsulation
  - Distribution
  - Partitioning
  - Scalability
  - Improved performance
  - Improved reliability

Multilayer architecture Features

- Manageability
- Increased consistency and flexibility
- Support for multiple customers
- Independent development
- Rapid development
- Packaging
- Configurability

Multilayer architecture Features

- Disadvantages:
  - Possible loss of performance and scalability
  - Security risks
  - Component management

Multilayer architecture
Related patterns

- Related patterns:
  - Presentation
    - Front controller
    - Application Controller
    - MVC
  - Business
    - Transfer
    - Application service

Multilayer architecture
Related patterns

- Object of business

– Integration

- *Data Access Object* (DAO)
- Domain store

Front controller pattern

- Purpose
  - Provides an access point for handling requests from the presentation layer
- Also known as
  - *Front controller*

Front controller pattern

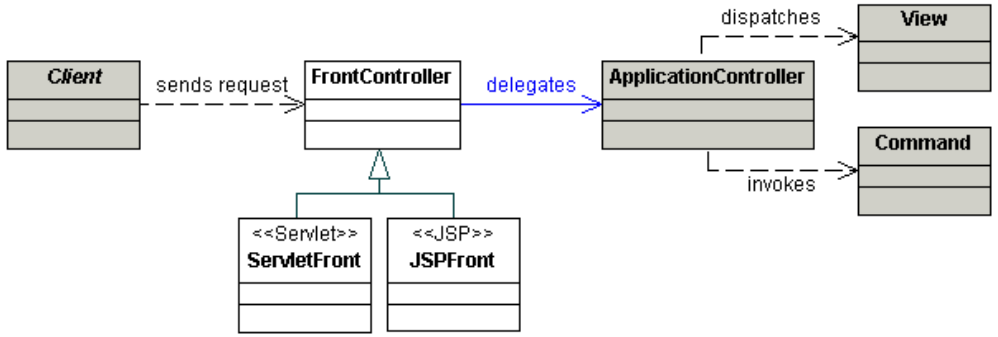
- Motivation
  - Avoiding duplicate control logic is desired
  - You want to apply a common logic to different requests
  - You want to separate the processing logic of the system from the view
  - It is desired to have centralized and controlled access points to the system.

Front controller pattern

- It must be applied when
  - You want to have an initial point of contact for handling requests, centralizing control logic and managing request handling activities.

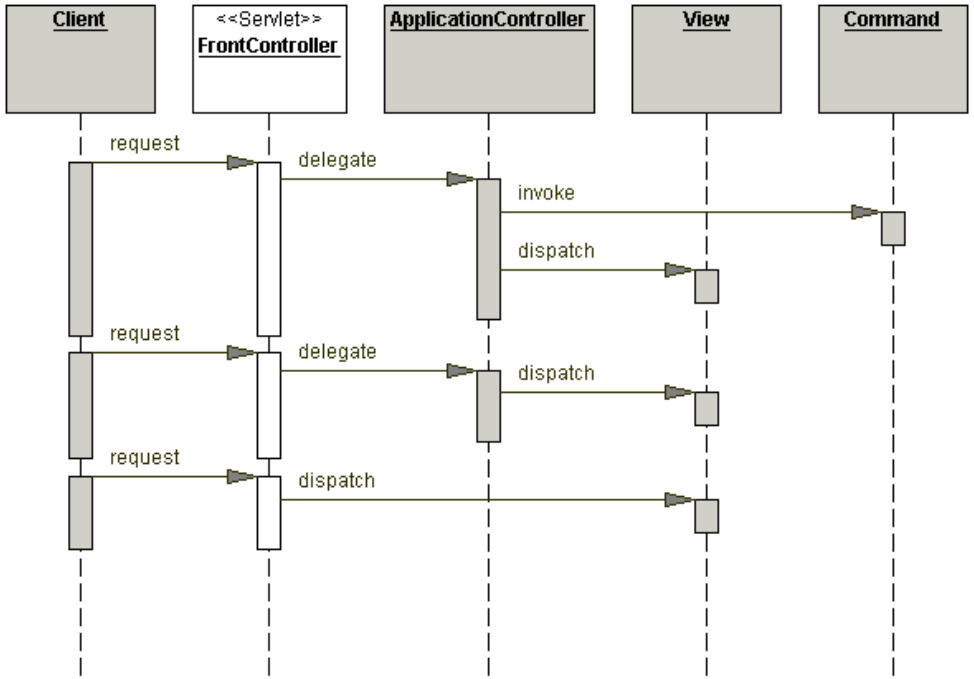
Front controller pattern

- Structure



Structure of the front controller pattern

Front controller pattern



Interaction of related objects by the front controller

Front controller pattern

- Consequences
  - Advantages:
    - Centralizes control
    - Improved application management
    - Improved reuse
    - Improved separation of roles
  - Inconveniences
    - In large applications it can grow very large

Front controller pattern

- Example code

```
public class FrontController extends HttpServlet {
  protected void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,
java.io.IOException {

      processRequest(request, response);

    }
```

Front controller pattern

```
protected void doPost(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, java.io.IOException {

      processRequest(request, response);
    }
```

Front controller pattern

```
protected void processRequest(HttpServletRequest request,
HttpServletResponse response) throws ServletException,
java.io.IOException {
      String page;
      ApplicationResources resource =
ApplicationResources.getInstance();
      try {
        RequestContext requestContext =
            new RequestContext(request, response);
```

Front controller pattern

```
ApplicationController applicationController = new
            ApplicationControllerImpl();
ResponseContext responseContext =
applicationController.handleRequest(requestContext);
applicationController.handleResponse(
             requestContext, responseContext);
      } catch (Exception e) {
LogManager.logMessage("FrontController:exception : " +
            e.getMessage());
request.setAttribute(resource.getMessageAttr(),
            "Exception occurred : " + e.getMessage());
         page = resource.getErrorPage(e);
```

Front controller pattern

```
dispatch(request, response, page);
      }
    }
// only this function is used if there is an error.
protected void dispatch(HttpServletRequest request,
HttpServletResponse response, String page)
throws javax.servlet.ServletException, java.io.IOException {
 RequestDispatcher dispatcher = this.getServletContext().
          getRequestDispatcher(page);
  dispatcher.forward(request, response);
    }
}
```

Application Controller Pattern

- Purpose
  - We want to centralize and modularize the management of actions and views.
- Also known as
  - *Application controller*

Application Controller Pattern

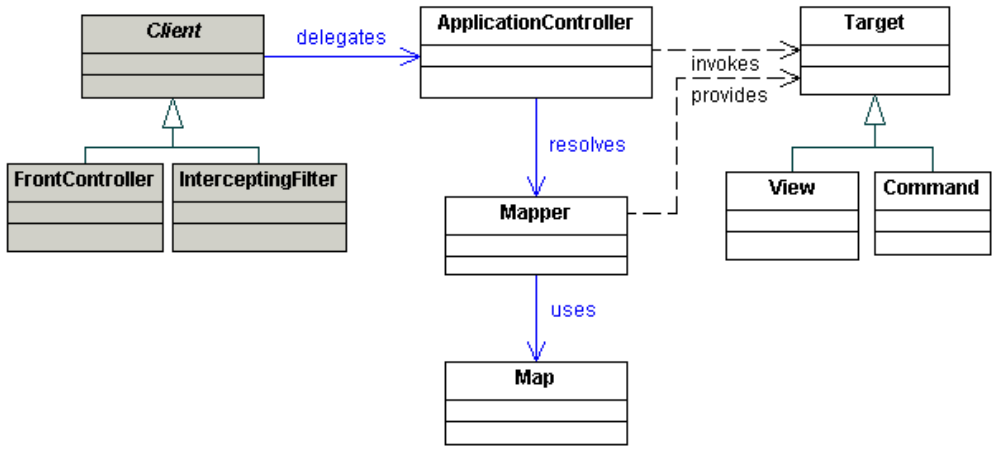
- Motivation
  - You want to reuse the view and action management code.
  - You want to improve the extensibility of request handling (e.g. add use cases to an application incrementally).
  - Improved code modularity and maintainability is desired, facilitating application extension and testing of request handling code independently of the web container.

Application Controller Pattern

- It must be applied when
  - You want to centralize the retrieval and invocation of request processing components, such as commands and views.

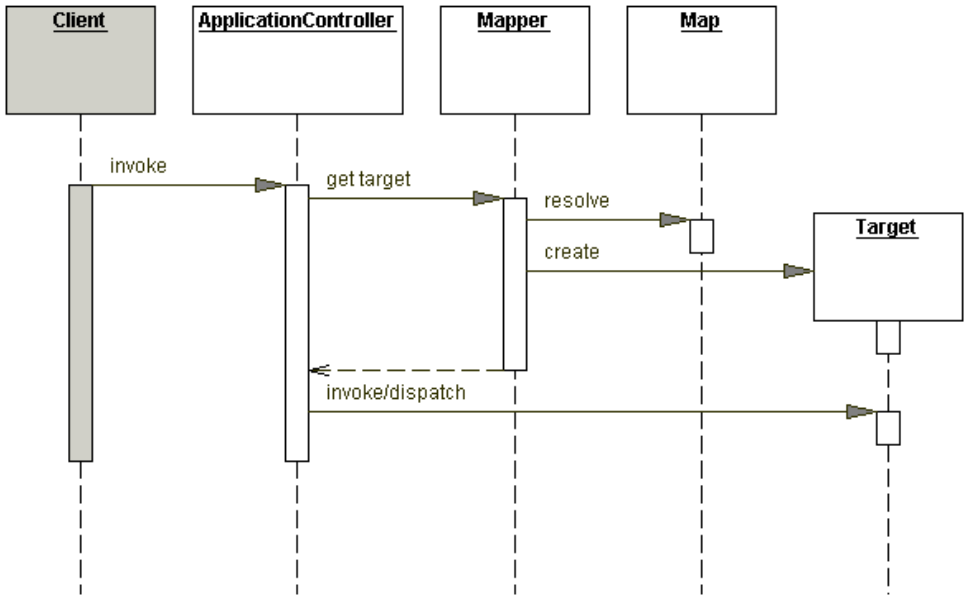
Application Controller Pattern

- Structure:



Structure of the application controller pattern

Application Controller Pattern



Interaction between related objects by application controller

Application Controller Pattern

- Consequences
  - Advantages
    - Improved modularity
    - Improved reuse
    - Improved extensibility
  - Inconveniences
    - Increases the number of objects involved
    - In large applications it can grow very large

Application Controller Pattern

- Example code

```
interface ApplicationController
{
      ResponseContext handleRequest(RequestContext requestContext);

      void handleResponse(RequestContext requestContext,
ResponseContext responseContext);
}
```

Application Controller Pattern

```
class WebApplicationController implements ApplicationController
{

public ResponseContext handleRequest(RequestContext requestContext)
{
       ResponseContext responseContext = null;

try {
      String commandName = requestContext.getCommandName();
```

Application Controller Pattern

```
CommandFactory commandFactory = CommandFactory.getInstance();
Command command = commandFactory.getCommand(commandName);
CommandProcessor commandProcessor = new CommandProcessor();
responseContext = commandProcessor.invoke(command,
            requestContext);
      } catch (java.lang.InstantiationException e) {
      } catch (java.lang.IllegalAccessException e) {
      }
      return responseContext; }
    . . . }
```

MVC Pattern

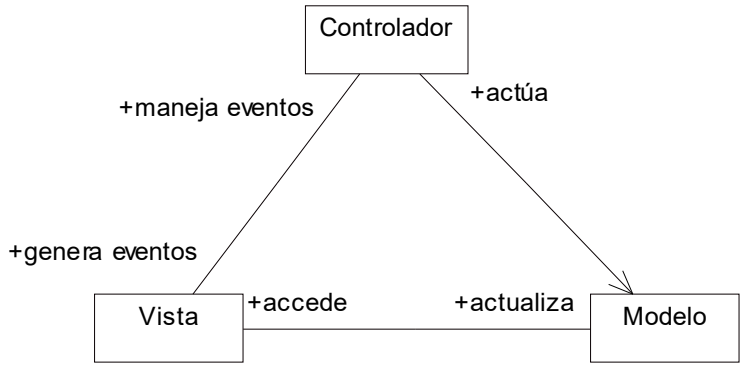
- In some frameworks the front and application controller are merged.
- Let's look at a more *classic* MVC
- The *Model View Controller MVC* pattern/architecture divides an interactive application into three components:
  - The *model* contains the basic functionality and data.
  - The *views* display/gather information to/from the user.
  - *Controllers* mediate between views and model

MVC Pattern

- The MVC pattern has two variants:
  - Active model
  - Passive model

MVC Pattern

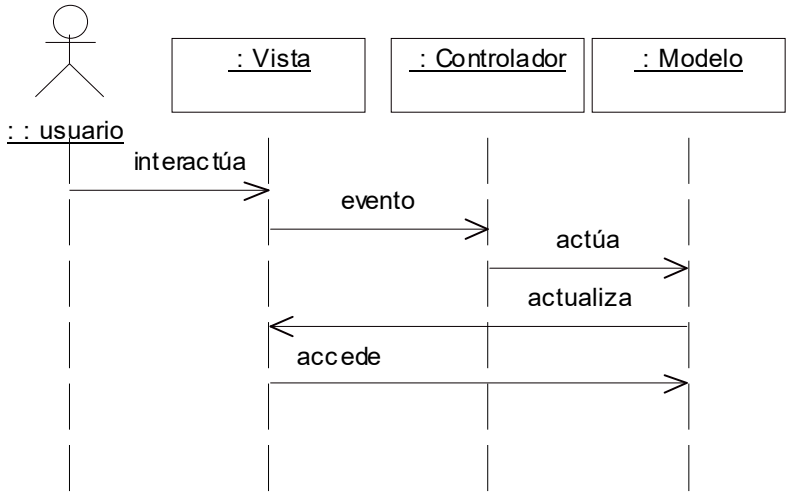
- Participants in MVC. Active model:



Participants in MVC. Active model

MVC Pattern

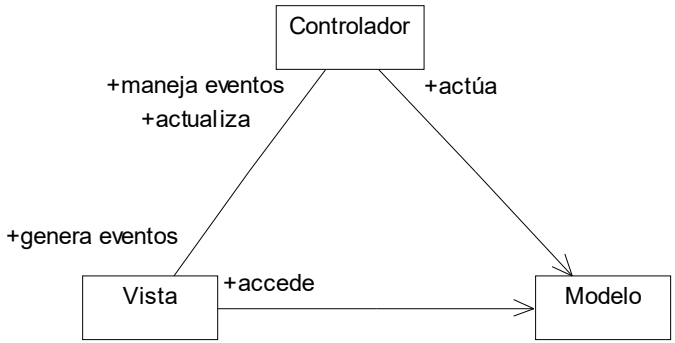
- Interaction in MVC. Active model:



Interaction in MVC. Active model

MVC Pattern

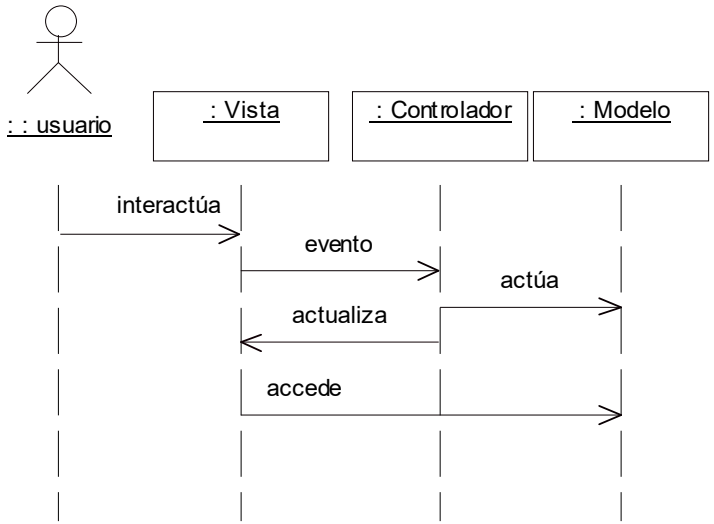
- Participants in MVC. Passive model:



Participants in MVC. Passive model

MVC Pattern

- Interaction in MVC. Passive model:



Interaction in MVC. Passive model

MVC Pattern

- Advantages:
  - Model independent of output representation and input behavior
  - There can be multiple views for the same model
  - Independent interface/logic changes
- Disadvantages:
  - Complexity

MVC Pattern

- Example code*:

```
class Vista extends JFrame implements IVista {
   JTextField value;
   JButton sum;
   public Vista(Model model)
       {……….

       sum= new JButton ("+");
       ActionListener controller=
                          new Controller(model);
       sumar.addActionListener(controller);
       } ……}
```

Active model without using `java.util.Observer`

MVC Pattern

```
class Controller implements ActionListener{
.............................
      public void actionPerformed (ActionEvent e)
   {
      model.sum();
   }
}
```

MVC Pattern

```
class Model {
   int value;
   IVista vista;
   ..................
   void sum()
       { value++;
       view.update(this); }
   int getValue()
       { return value; }
}
```

MVC Pattern

```
public interface IVista {
   void update(Object updated);
}
```

MVC Pattern

```
class Vista extends JFrame implements IVista {
.........................
public void update (Object o){
  Model model= (Model) or;
  Integer i= new Integer(model.getValue());
  value.setText(i.toString()); }
.........................
}
```

MVC Pattern

- In the previous example:
  - The view that sends events to the controller is the same view that receives model/controller updates.
  - Match the interface event handler and the business event handler
- In general, this is unreasonable (e.g. web applications).

MVC Pattern

- Regarding the number of controllers, there may be:
  - One per event
  - One per functionality/stimulus set
  - One per application

MVC Pattern

- Example code*

```
public class GUIAltaUser extends JFrame {
.....
      public GUIAltaUser()
  { setTitle("User registration");
      JPanel panel= new JPanel();
      JLabel lName= new JLabel("Name:");
      final JTextField tName= new JTextField(20);
      JLabel lEMail= new JLabel("e-mail:");
      final JTextField tName= new JTextField(20);
      final JTextField tEMail= new JTextField(20);
      JButton accept= new JButton("Accept");
      JButton cancel= new JButton("Cancel");
```

Single controller, passive model

MVC Pattern

```
panel.add(lName);
panel.add(tName);
panel.add(lEMail);
panel.add(tEmail);
panel.add(accept);
panel.add(cancel);
getContentPane().add(panel);

pack();
```

MVC Pattern

```
accept.addActionListener(new ActionListener()
  { public void actionPerformed(ActionEvent e)
      { setVisible(false);
          String name= tName.getText();
          String eMail= tEMail.getText();
          TUsuario tU= new TUsuario(nombre , eMail);
          Controller.getInstance().
          action(Events.ALTA_USER, tU);
      }
                                                  });
.....
}
```

MVC Pattern

```
public class Event {

   public static final int ALTA_USER= 101;
   public static final int BAJA_USUARIO= 102;
   public static final int SHOW_USER= 103;
   ……….
   public static final RES_ALTA_USER_OK= 401;
   public static final RES_ALTA_USER_KO= 402;
   ……….
}
```

MVC Pattern

```
public class Controller {
   ……..
   //this is a controller access option
   //a services and GUI
   //there may be more advanced ones
   private SAUsUser saUser;
   private IGUI gui;
```

MVC Pattern

```
//naif implementation of a controller table
//according to the singleton, it should be in the subclass
public void action(int event, Object data)
{ switch (event){
       case EVENT.ALTA_USER: {
      TUser tUser= (TUser) data;
      int res= saUser.high(tUser);
      if (res>0) gui.update(Event.RES_ALTA_USER_OK,
            new Integer(res));
      else
      gui.update(Event.RES_ALTA_USER_KO, null);
      break; }
   case Event.UNSUBSCRIBE_USER: { ..... }
```

.....

MVC Pattern

```
public interface IGUI {
  // we do not use java.util.Observer
 //because it forces the data to be observable

 void update(int event, Object data);

}
```

MVC Pattern

```
public class GUIBiblioteca extends JFrame implements IGUI {

   private static GUIBiblioteca guiLibrary guiLibrary;
   private IGUIUser guiUser;
   private IGUIPublication guiPublication;
   private IGUIPrestamo guiPrestamo;
   private Controller controller;
```

MVC Pattern

```
public void update(int event, Object data)
  { switch (event)
      {
  case Event.SHOW_GUI_LIBRARY:
  { setVisible(true); break; }
  case Event.HIDE_GUI_GUIDE_LIBRARY: { setVisible(false); break; }
```

MVC Pattern

```
case EventGUI.RES_ALTA_USER_OK:
{ Integer id= (Integer) data;
  JOptionPane.showMessageDialog(null,
      "User created with ID: "+id.intValue()); setVisible(true);
  break; }

case EventGUI.RES_ALTA_USER_KO:
{ JOptionPane.showMessageDialog(null,
      "User could not be created"); setVisible(true); break; }
.....
}
}
```

MVC Pattern

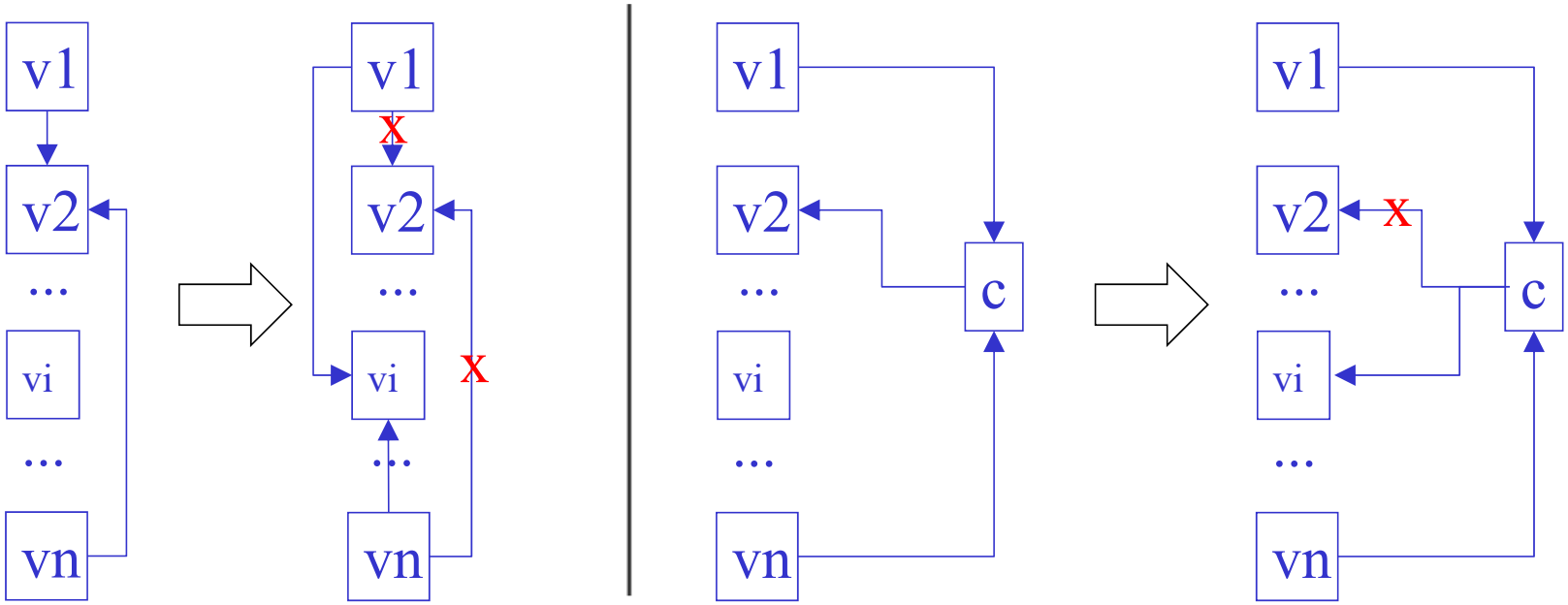
- Comments:

  - The MVC pattern is a particular case of the *observer* pattern.

  - Views in MVC can be nested. In this way a view would be a particular case of the *composite* pattern

  - By centralizing the interaction between views, and between views and model, it is easier to change both view selection and model selection.

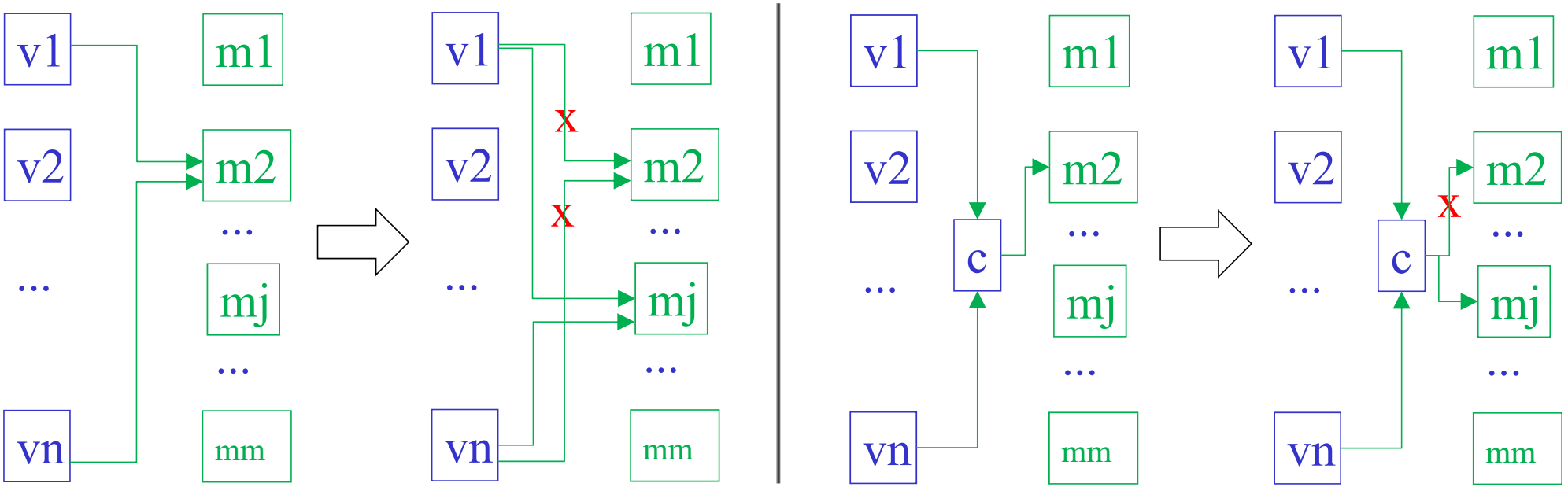
MVC Pattern



Changes without controller

Changes with controller

MVC Pattern



Changes without controller

Changes with controller

Transfer pattern

- Purpose
  - Independence of data exchange between layers
- Also known as
  - *Transfer*

Transfer pattern

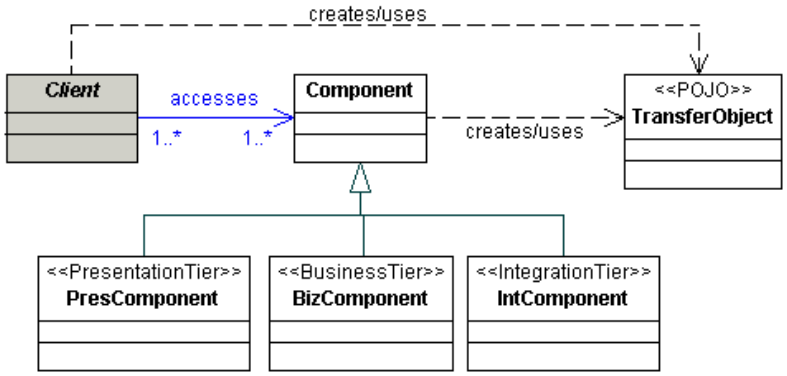
- Motivation
  - If we want to make the layers independent, they cannot have knowledge of the representation of the entities of our system within each layer.
  - For example, if we access relational databases, clients should be aware of the existence of *columns* in the data.

Transfer pattern

- It must be applied when
  - You do not want to know the internal representation of an entity within a layer.
- Note
  - As a communication mechanism between layers, they are serializable objects.

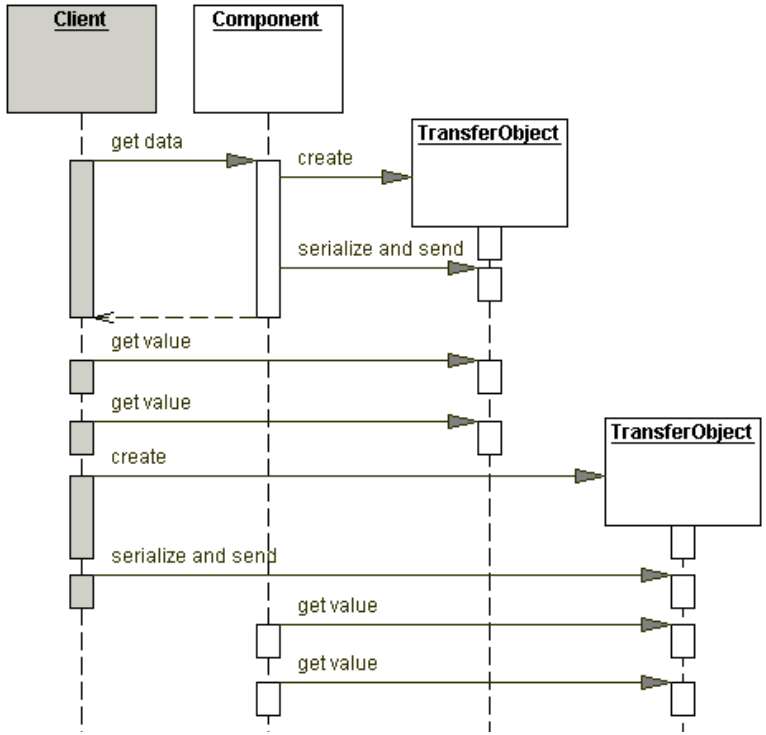
Transfer pattern

- Structure



Structure of the transfer pattern

Transfer pattern



Interaction between objects related by the transfer pattern

Transfer pattern

- Consequences
  - Advantages
    - Helps to make layers independent
  - Inconveniences
    - Significantly increases the number of objects in the system

Transfer pattern

- Example code

```
public TUsuario implements Serializable {
   public int id;
   public String name;
   public String eMail;
   public boolean active;

   public TUsuario(String nombre, String eMail)
   { this.id=0; this.name= name;
      this.eMail= eMail; this.active= true; }

   public TUsuario(int id, String nombre, String
                     eMail, boolean active)
   { this.id= id; this.name= name;
      this.eMail= eMail; this.asset= asset; }
```

Transfer pattern

```
public int getId()
{ return id; }

public String getName()
{ return name; }

public String getEMail()
{ return eMail; }

public boolean getActive()
{ return active; }
```

Transfer pattern

```
  public setId(int id)
  { this.id= id; }

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

  public void setEMail(String eMail)
  { this.eMail= eMail; }

  public void setActivo(boolean activo)
      { this.asset= asset; }

}
```

Transfer pattern

```
public DAOUsuarioImp implements DAOUsuario {

   public TUsuario read (int id)
   {
       //database access code

       TUser user=
             new TUser(id, name, eMail, active);

       return user;
   }
   ..............
}
```

DAO pattern

- Purpose
  - Allows you to access the data layer (resources, in general), providing object-oriented representations (e.g. transfer objects) to your clients.
- Also known as
  - *Data access object*
  - Data access object

DAO pattern

- Motivation
  - Information systems (and many programs) store user data
  - This data usually has a structure, which is embodied in a representation system (e.g., relational, XML).

DAO pattern

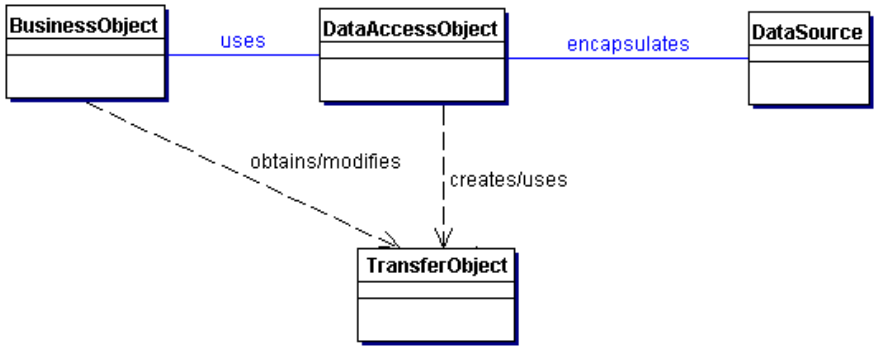
– Handling this data forces to:
  - Know the access mechanisms of the data management system (e.g., database, operating system, etc.).
  - Know the representation of data in the data management system (e.g., columns, elements, bytes, etc.).
– A business layer customer should be independent of these issues

DAO pattern

  - Thus, the data layer could be changed without affecting the business layer. Only the integration layer, which is lighter than the business layer, would have to be updated.
- It must be applied when
  - You want to make data representation and access independent of data processing.

DAO pattern

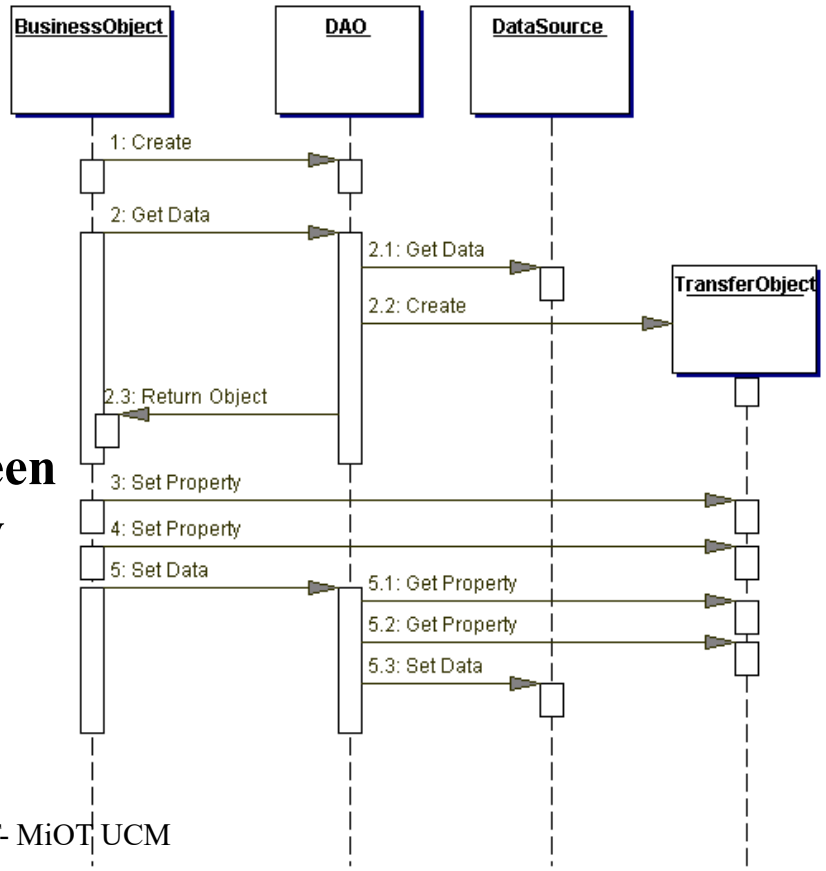
- Structure



DAO pattern structure

DAO pattern

Interaction between related objects by DAO pattern



DAO pattern

- Consequences
  - Advantages:
    - It separates the processing of data from its access and structure.
    - Allows the business layer to be separated from the data layer.
  - Inconveniences
    - Increases the number of objects in the system

DAO pattern

- Let's see a simple example of a DB connection from

- The code:
  - Connects to a database
  - Send a query
  - Process the result

DAO pattern

```
import java.sql.*

public void connectToAndQueryDatabase(String username, String
password)
{
try{
 Connection con = DriverManager.getConnection(
                    "jdbc:myDriver:myDatabase",
                     username, password);
```

DAO pattern

```
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM Table1");

while (rs.next()) {
   int x = rs.getInt("a");
   String s = rs.getString("b");
   float f = rs.getFloat("c");
  }
} finally {
            if (stmt != null) pstmt.close();
            if (con != null) con.close(); }
        }

}
```

DAO pattern

- Let's see another simple example, extracted from the same site, which makes an *update*

```
import java.sql.*;

public class UpdateCar {

    public static void UpdateCarNum(int carNo, int empNo)throws
SQLException {
```

DAO pattern

```
Connection con = null;
PreparedStatement pstmt = null;
try { con = DriverManager.getConnection(
                        "jdbc:default:connection");
      pstmt = con.prepareStatement(
                          "UPDATE EMPLOYEES " +
                          "SET CAR_NUMBER = ? " +
                          "WHERE EMPLOYEE_NUMBER = ?");

      pstmt.setInt(1, carNo);
      pstmt.setInt(2, empNo);
      pstmt.executeUpdate();
```

DAO pattern

```
}
finally {
            if (pstmt != null) pstmt.close();
            if (con != null) con.close();
        }
    }
}
```

DAO pattern

- Example code

```
public interface DAOUsuario {
   public int create(TUser tUser);
   public TUser read(int id);
   public Collection<TUser> readAll();
   public TUser readByName(String name);
   public int update(TUser tUser);
   public int delete (int id);

}
```

DAO pattern

```
public class DAOUsuarioImp implements DAOUsuario {
……
   public int create(TUser tUser)
   {
       int id= -1;
       //connection to the database
       PreparedStatement ps;

       ps = connection.prepareStatement("INSERT INTO user (name, eMail,
   active) VALUES (?,?,?,?)", Statement.RETURN_GENERATED_KEYS);
       ps.setString(1, tUser.getName());
       ps.setString(2, tUser.getEMail());
       ps.setBoolean(3, tUser.getActive());
       ps.execute();
```

DAO pattern

```
 ResultSet rs= pstmt.getGeneratedKeys();
 if (rs.next())
 {
     res= rs.getInt(1);
 }

//close connection and handle exceptions

 return id;
 }
 ……
 }
```

DAO pattern

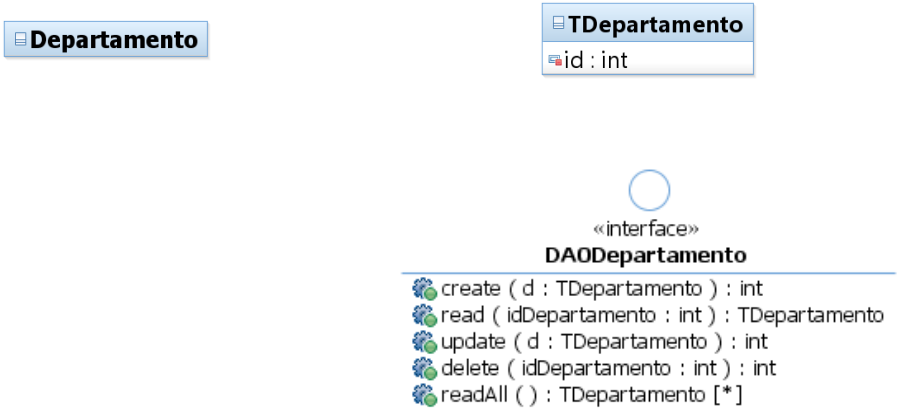
- Although it is obvious in these slides, it is essential that DAOs capture and trigger the corresponding exceptions when accessing external resources.
- Thus, the business layer will know what has happened if there has been some kind of access failure.

DAO pattern

- NOTE
  - Although, in general, DAOs should only have the CRUD operations (Create, Read, Update and Delete), it is possible that in a multi-tier architecture without business objects, we may need to enrich the DAOs to facilitate the management of the 1..n and m..n relations.

DAO pattern

– For a class:

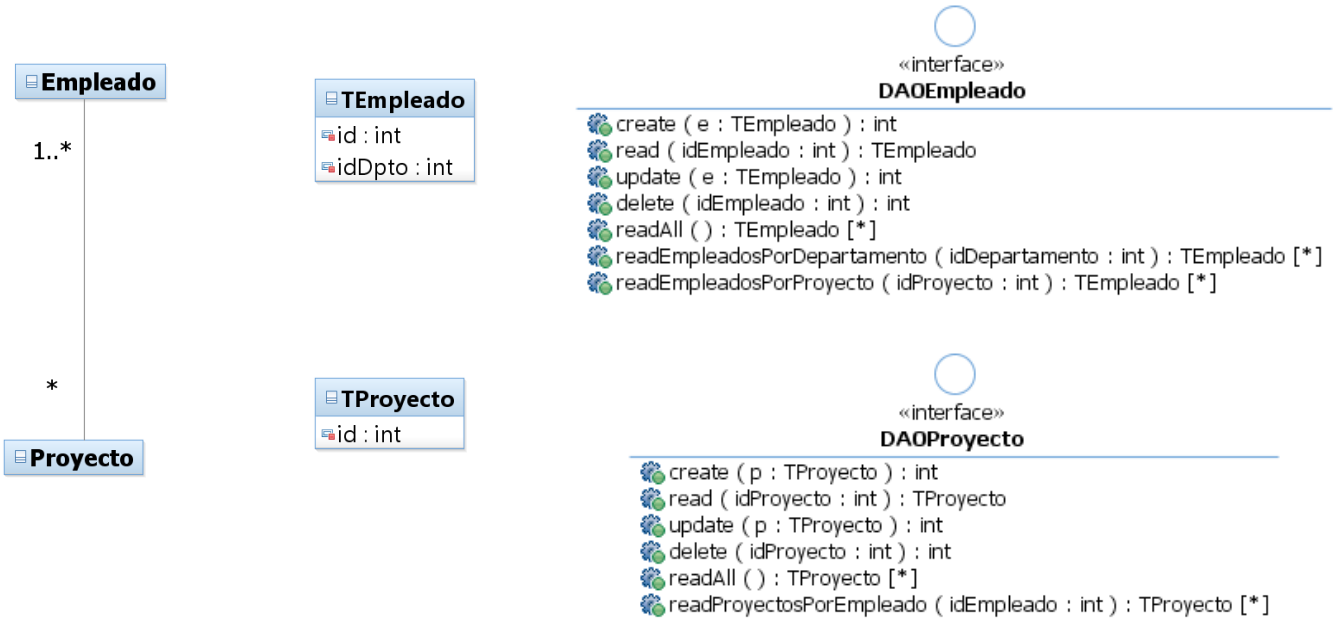


DAO pattern

– For a class, N end of a relation 1..N

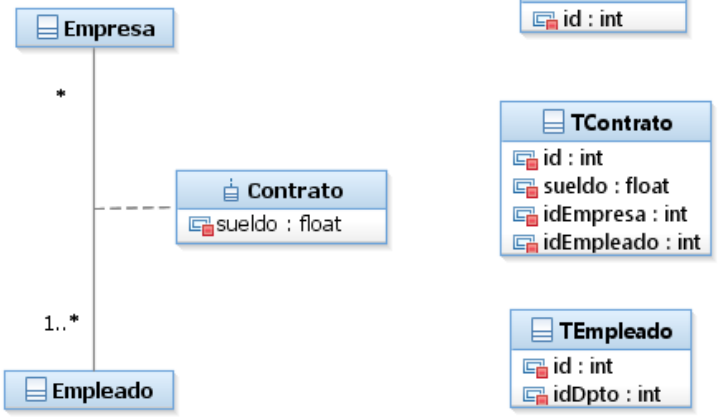
DAO pattern

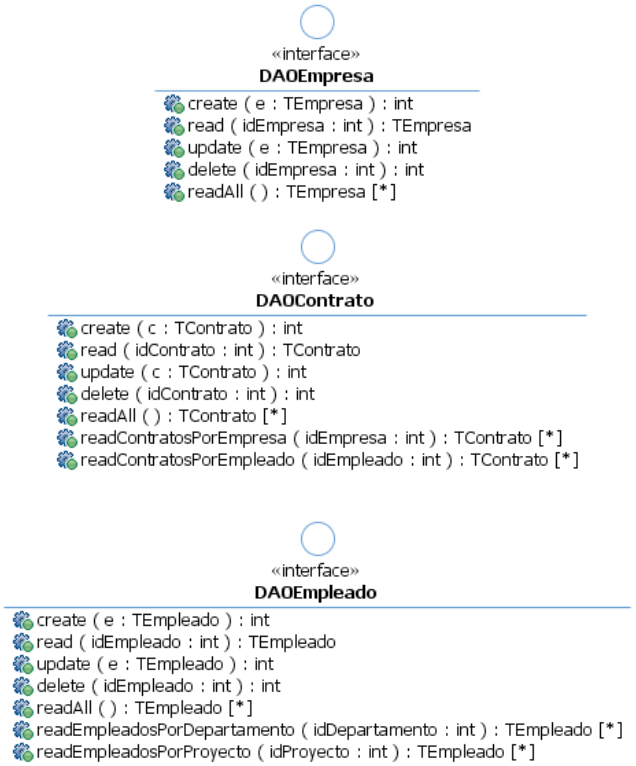
– For two extreme classes of a relation M..N



DAO pattern

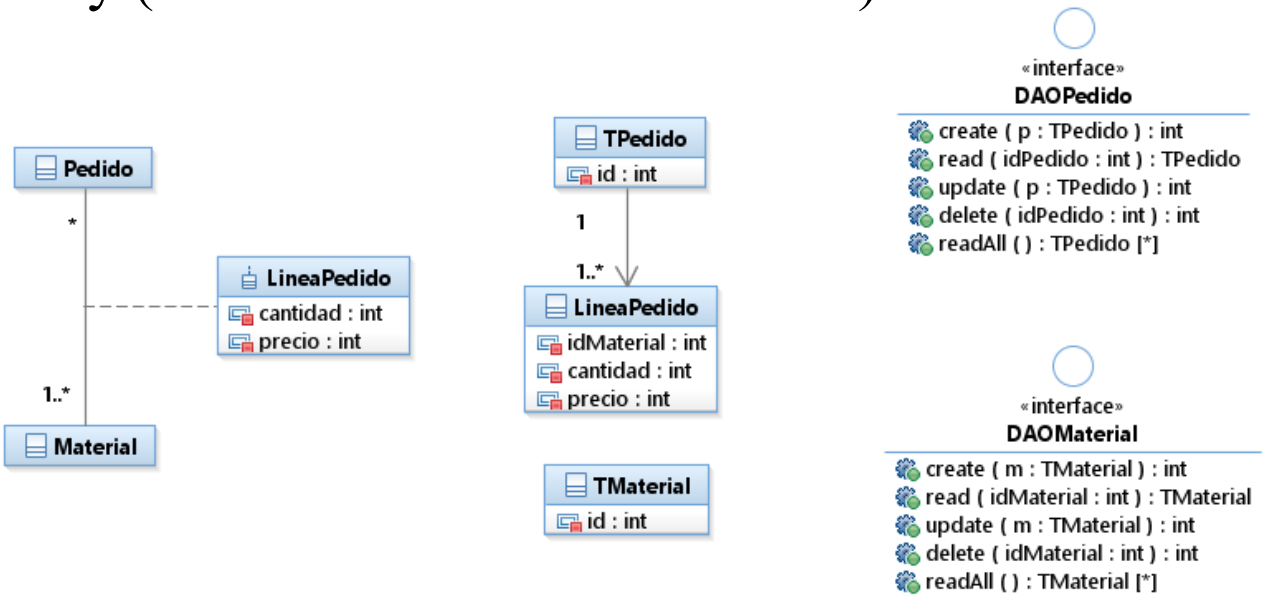
– In the case of an association class, it is like two relations 1..N:





DAO pattern

– Unless we think that the intermediate class does not have sufficient entity (i.e. does not have its own ID)

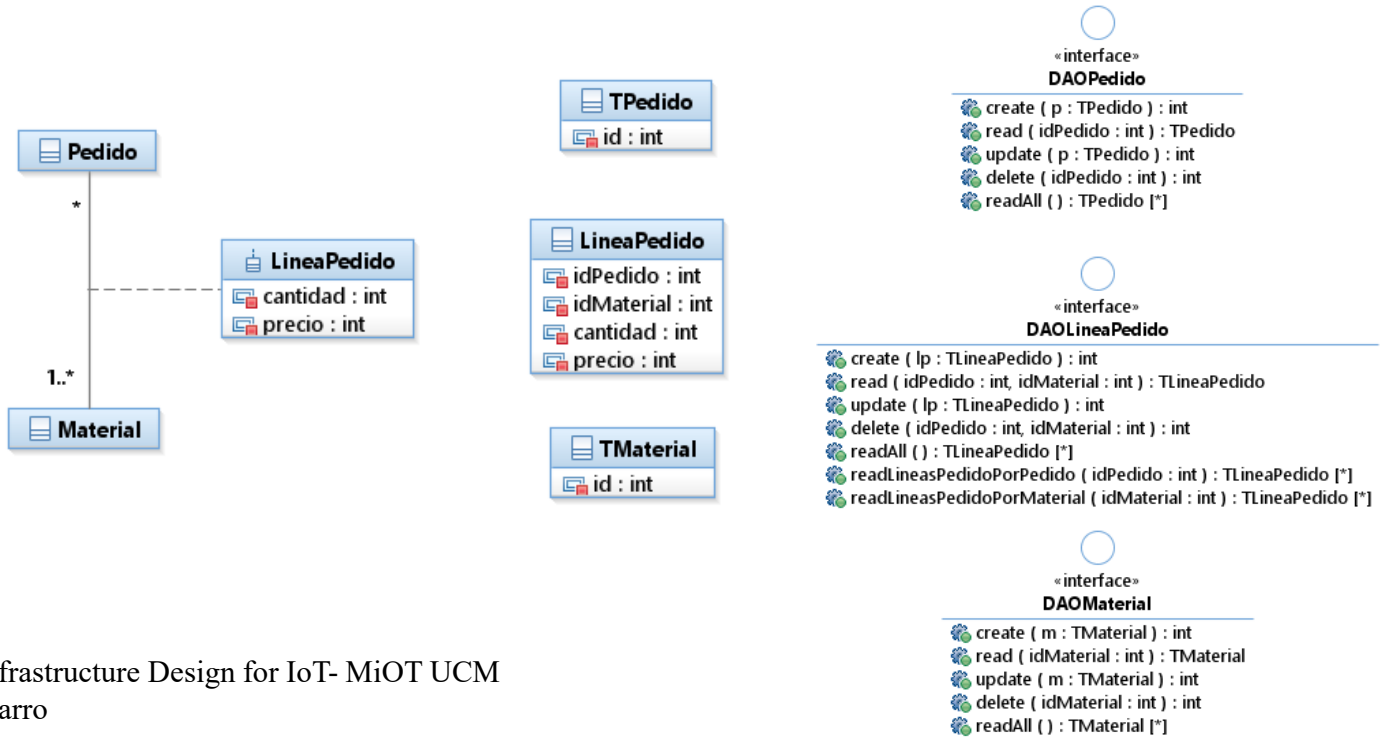


DAO pattern

- In the latter case, if the order lines, despite not being accessed outside the order transfer, are updated frequently (e.g. due to changes in orders), it might be appropriate to give them DAO, but not application service.

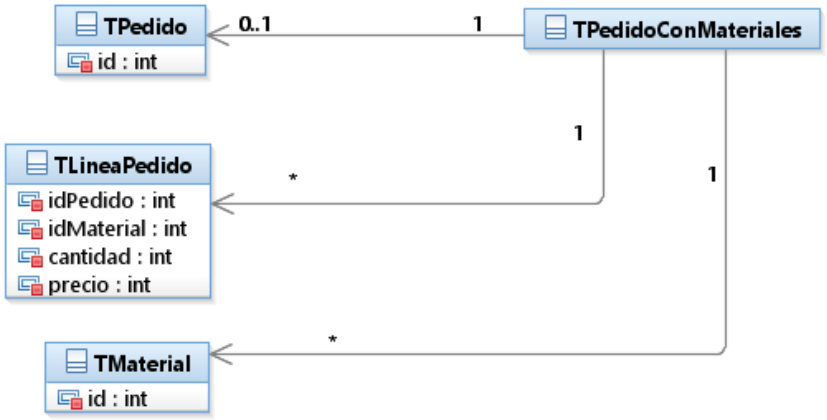
DAO pattern

– In any case, this option seems *cleaner*



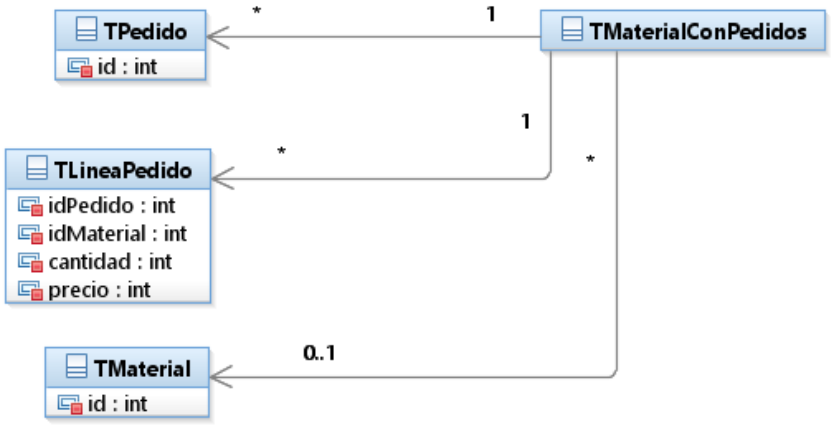
DAO pattern

- This option together with a TOA (e.g. an HS operation) would allow to read an order with its materials.



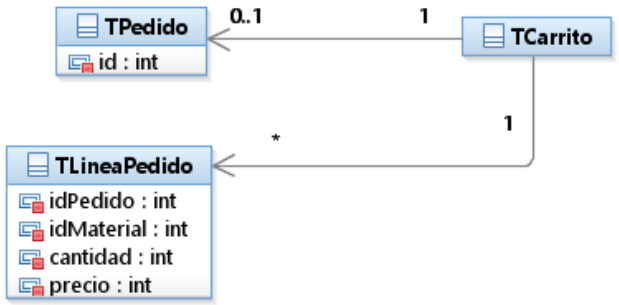
DAO pattern

- Or a material with the orders in which it appears



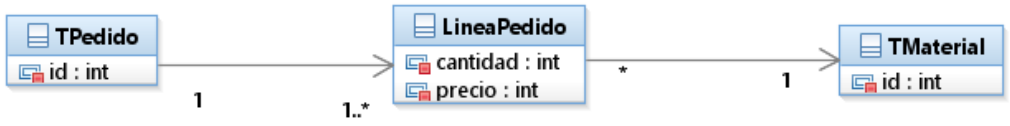
DAO pattern

- Or even a trolley



DAO pattern

– Martin Fowler is more pragmatic and would return a network of transfers created on demand from the union of use cases.
  - This option however, generates networks of *pruned* objects in order to solve both the cycling and loading problem of all referenced elements.



Application Service Pattern

- Purpose
  - Centralizes business logic.
- Also known as:
  - *Application service*

Application Service Pattern

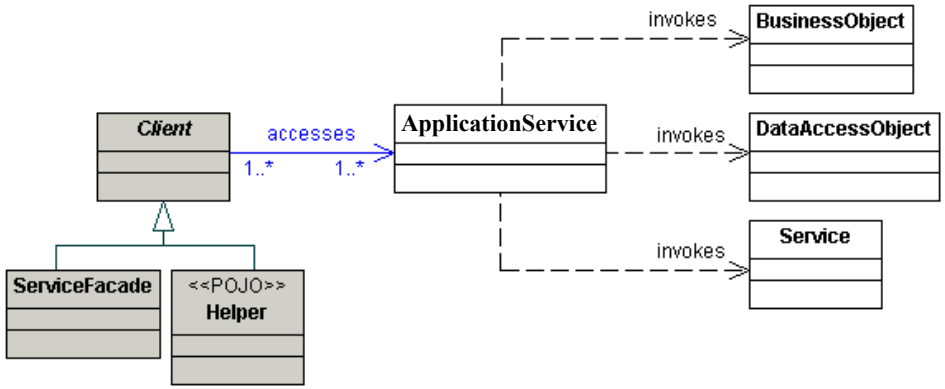
- Motivation
  - In a multi-tier architecture, the business logic must be somewhere.
  - Putting it in the controller would couple presentation and business
  - Putting it in the DAO would couple business and integration
  - That is why we include it in the application services.

Application Service Pattern

- It must be applied when
  - You want to represent a business logic that acts on different business services or *objects*
  - You want to group related functionalities
  - You want to encapsulate logic that is not represented by business objects.

Application Service Pattern

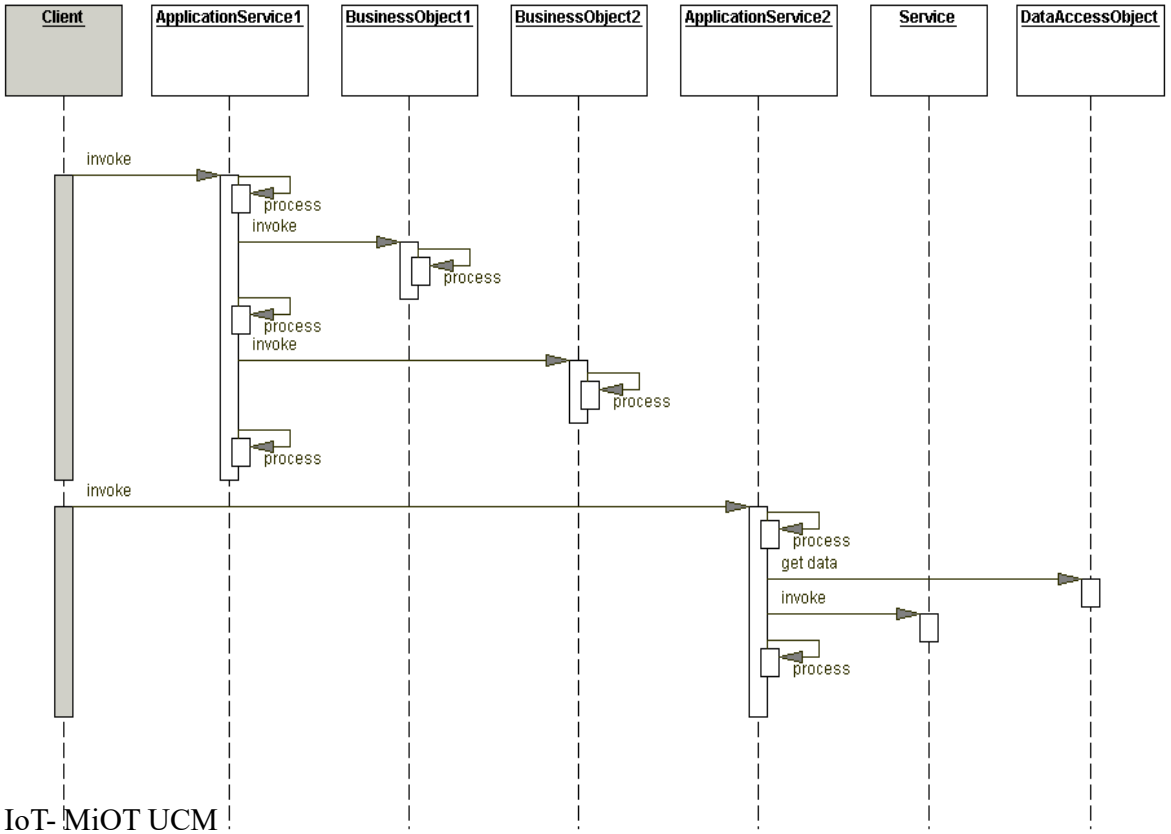
- Structure



Structure of the application service pattern

Application Service Pattern

Interaction between objects related by the application service pattern



Application Service Pattern

- Consequences
  - Advantages
    - Centralizes business logic
    - Improved code reusability
    - Avoids code duplication
    - Simplifies the implementation of facades
  - Inconveniences
    - Introduces one more level of indirection

Application Service Pattern

- Example code:

```
public interface SAUser {
   public int create(TUser tUser);
   public TUser read(int id);
   public Collection<TUser> readAll();
   public int update(TUser tUser);
   public int delete (int id);

}
```

Application Service Pattern

```
public class SAUsuarioImp implements SAUsuario {
  public int create(TUser tUser)
  { int id= -1;
      DAOUsuario daoUsuario;
      if (tUser!=null)
      { //access to DAO implementation
         tUser read=
            daoUser.readByName(tUser.getName());
         if (read==null) id= daoUser.create(tUser);
       }
      return id;
   }
.....}
```

Application Service Pattern

- Note
  - Although in the *Core J2EE Patterns* book example, application services collaborate with each other to obtain business objects (and by extension, data), this approach could complicate data consistency validations in a non-EJB multi-user environment.
  - In any case, note that the application service invoked in the example (t136), does not seem to access persistent information.

Application Service Pattern

- Note
  - Application services do not usually have attributes to make them *lighter in weight.*
  - So where are the objects that have business attributes and operations?
  - These objects are the *business objects*

Pattern subject of business

- Purpose
  - Represent business logic and domain model in object-oriented terms
- Also known as:
  - *Business object*

Pattern subject of business

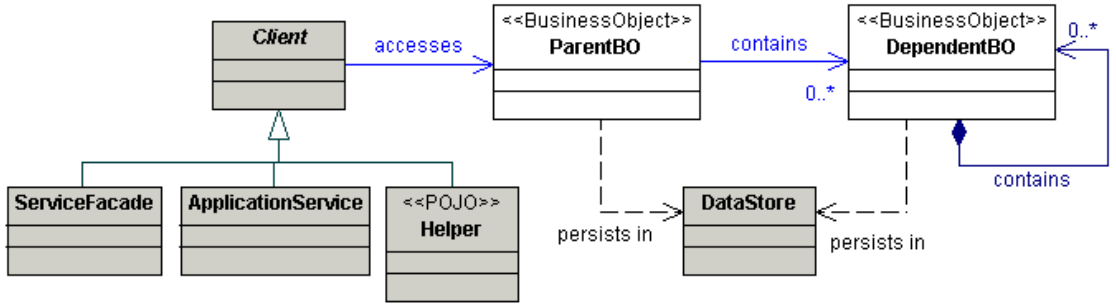
- Motivation
  - When there is little or no business logic, applications can allow clients to access the data layer directly.
  - Thus, a business layer component (e.g., `UserImpServices`) could access a DAO directly.
  - However, if there are a large number of computational processes associated with the data in the client, these processes should be encapsulated in an object representing a business object.
  -

Pattern subject of business

- Should be applied when:
  - A conceptual model with advanced validation rules and business logic is available.
  - You want to separate the business logic from the rest of the application.
  - You want to centralize the business logic
  - You want to increase the reusability of the code

Pattern subject of business

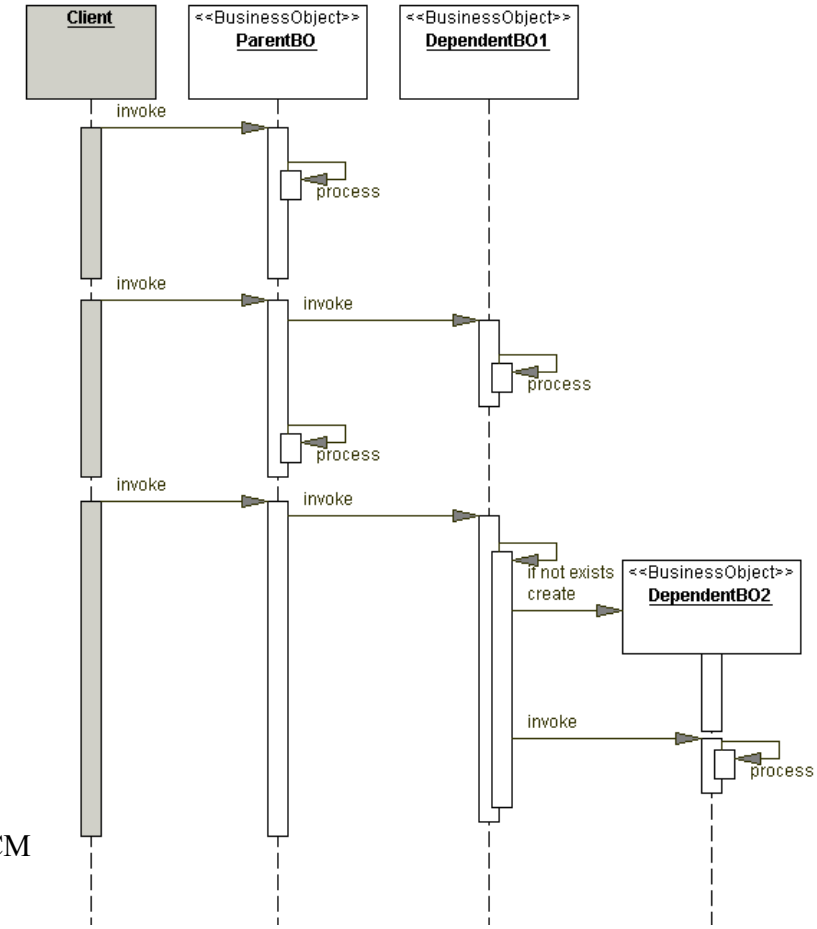
- Structure

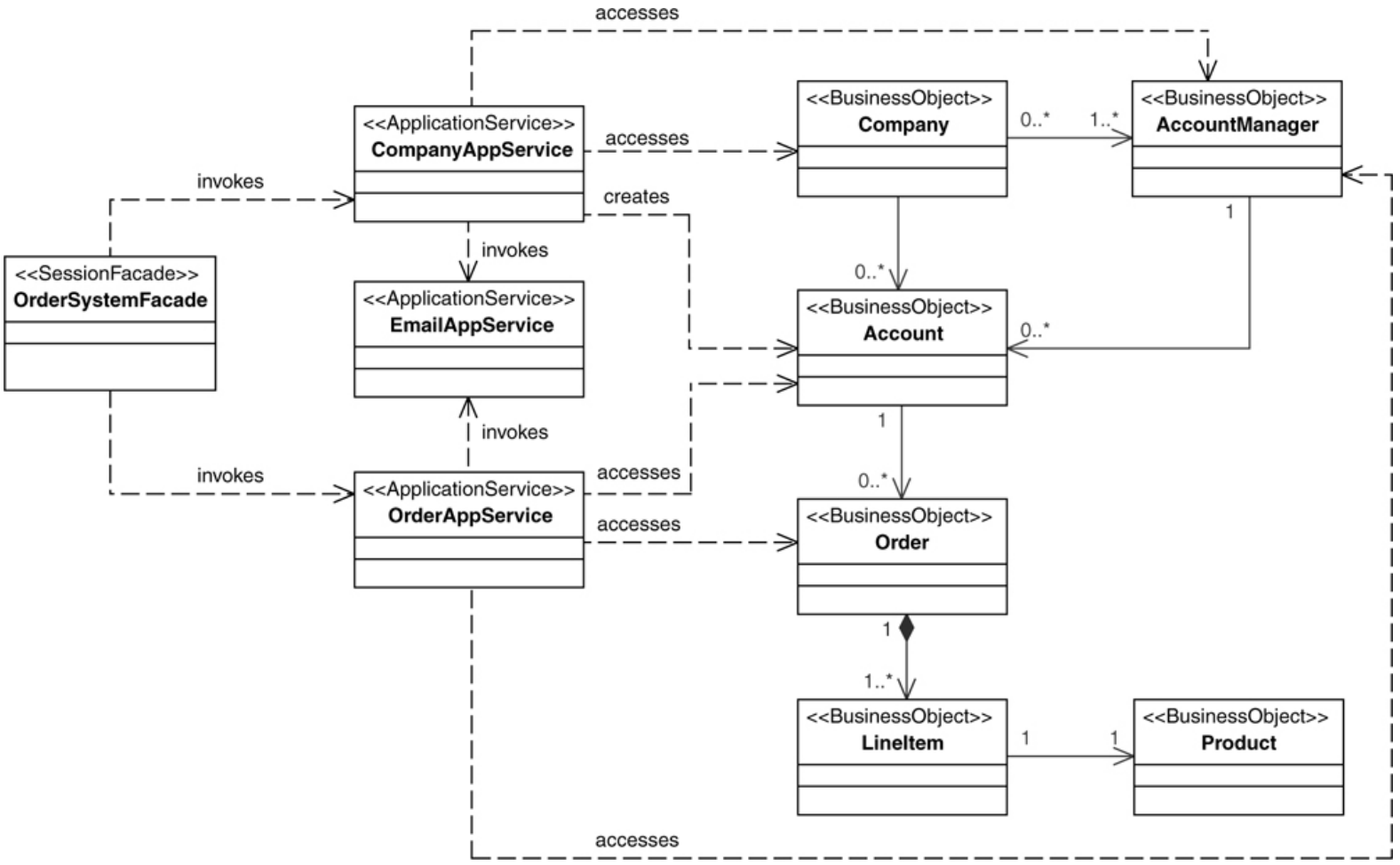


Structure of the business object pattern

Pattern subject of business

Interaction between objects related by the business object pattern





Relationship between business objects and application services

Pattern subject of business

- Consequences
  - Advantages
    - Promotes an object-oriented approach in the implementation of the business model.
    - Centralizes business behavior, promoting reusability.
    - Avoid code duplication

Pattern subject of business

- Inconveniences
  - Adds an indirection layer
  - Can produce "inflated" objects of functionality
  - Persistence of these business objects

Pattern subject of business

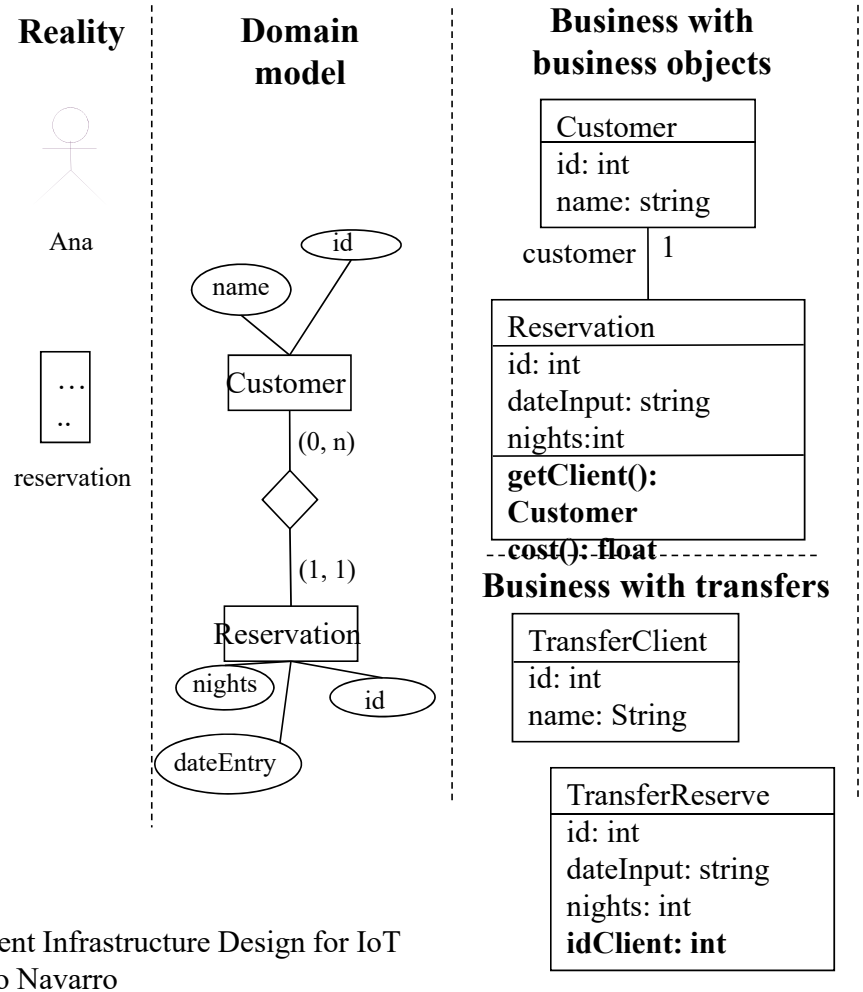
- Example code

```
public class Employee {
    private int id;
    private String name;
    private long salary;
    private Department department;
.....

    Department getDepartment()
              { return department; }
.....
```

Pattern subject of business

```
public class Department {
       private int id;
       private String name;
       private Collection<Employee> employees;

    public int getId() {
        return id; }
……
    public Collection<Employee> getEmployees() {
        return employees;}
……
}
```

Note

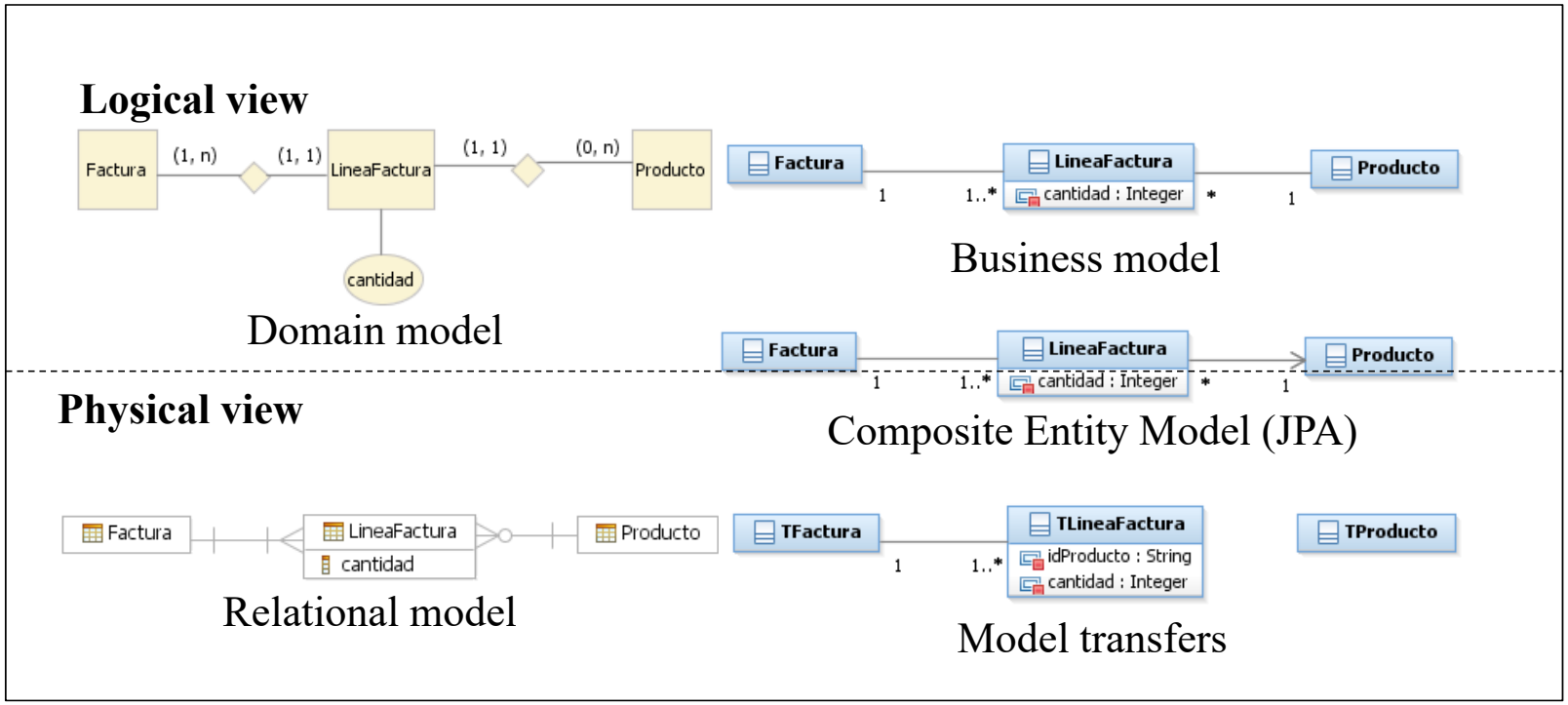


Relationships between reality, domain, business and resources

Pattern subject of business

- There is a fairly direct equivalence between:
  - Domain and business model
  - Tables and transfers
  - The composite entities would be something like an intermediate view between the domain model and the model proposed by the transfers.

Pattern subject of business



Equivalences between different models

Domain store

- Purpose
  - You want to separate the persistence from the object model.
- Also known as:
  - Domain store
  - Unit of work + Query object + Data mapper + Table data gateway + Dependent mapping + Domain model + Data transfer object + Identity map + Lazy load

Domain store

- Motivation
  - Many systems have a complex object model that requires sophisticated persistence strategies.
  - These strategies should be independent of the business objects, so that they are not coupled with a specific warehouse.

Domain store

– Thus, four simultaneous problems must be solved:
  - Persistence
  - Dynamic load
  - Transaction management
  - Concurrence

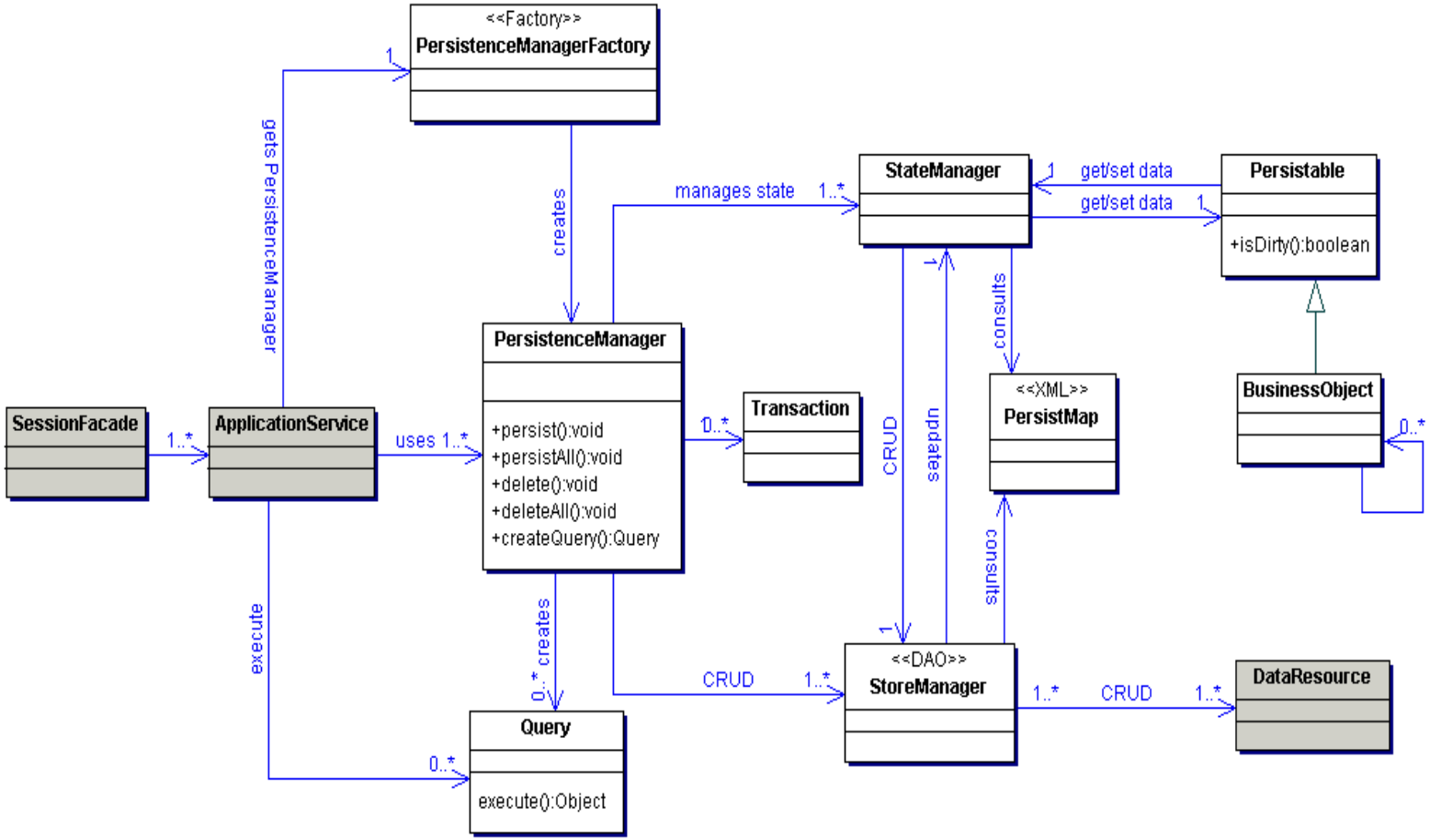
Domain store

- Context
  - You want to omit persistence details in the business objects.
  - The application could run in a web container
  - The object model uses inheritance and complex relationships

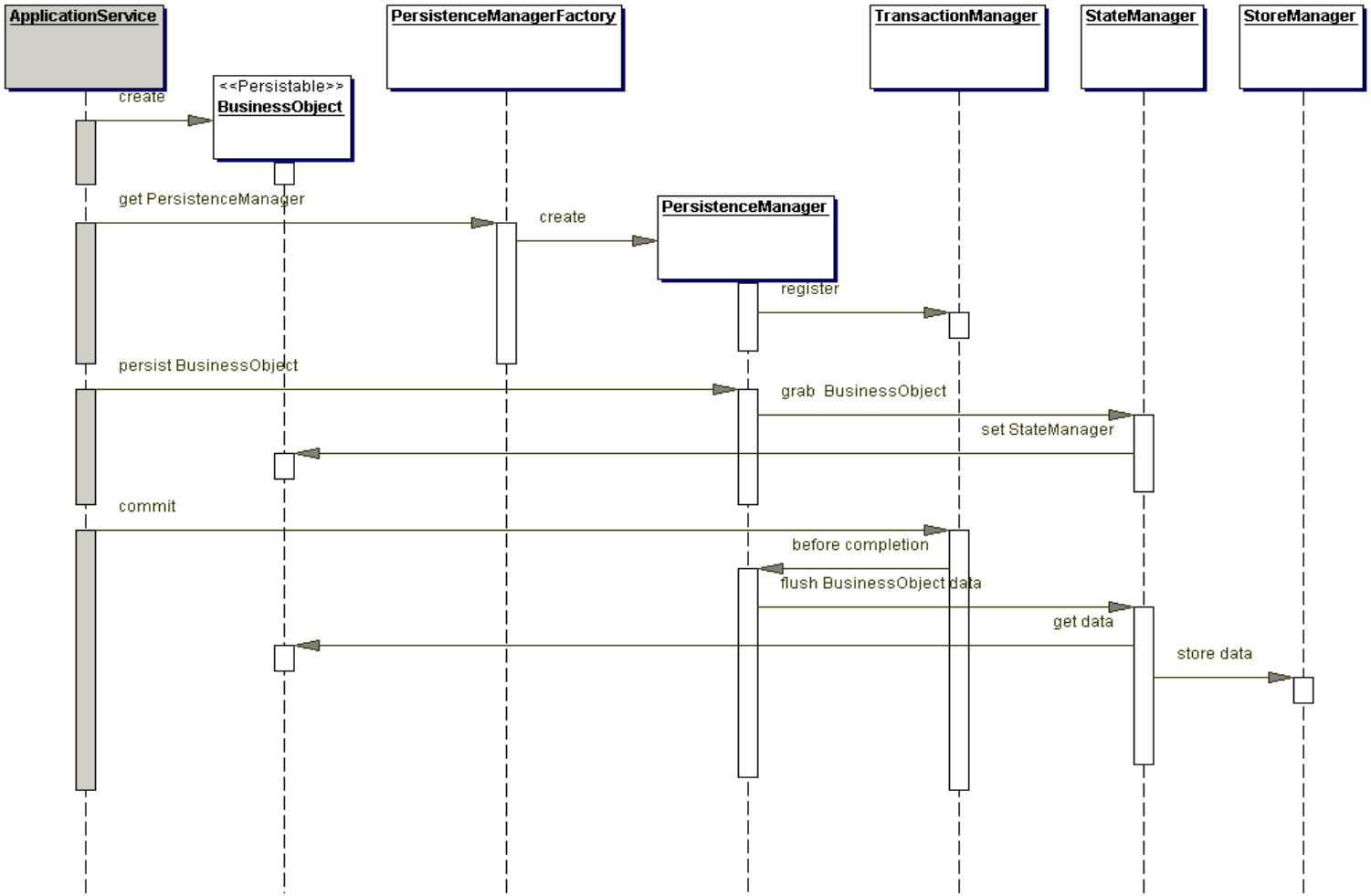
Domain store

- Solution
  - Using a domain store to transparently persist an object model

- Description

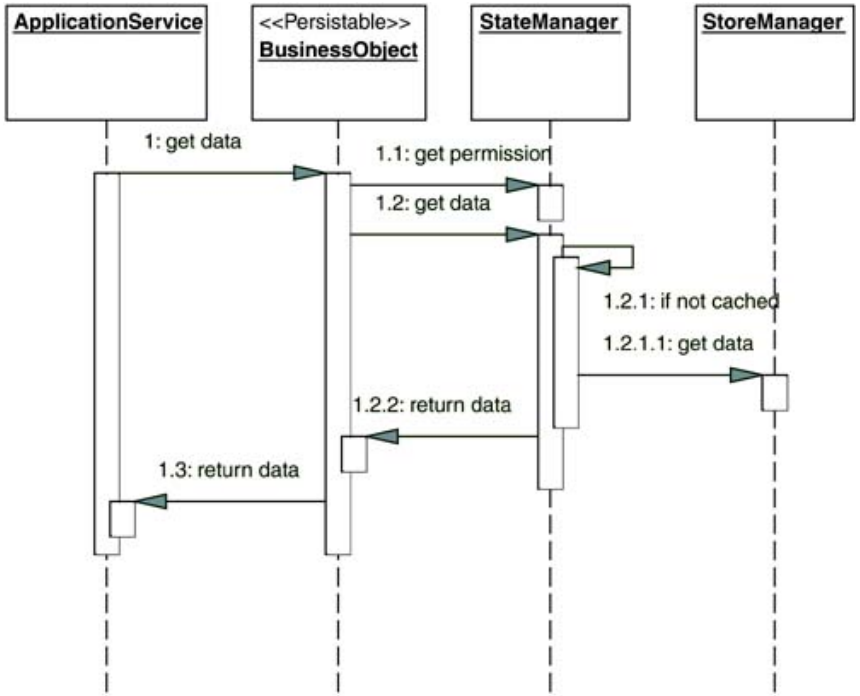


Structure of the domain store pattern

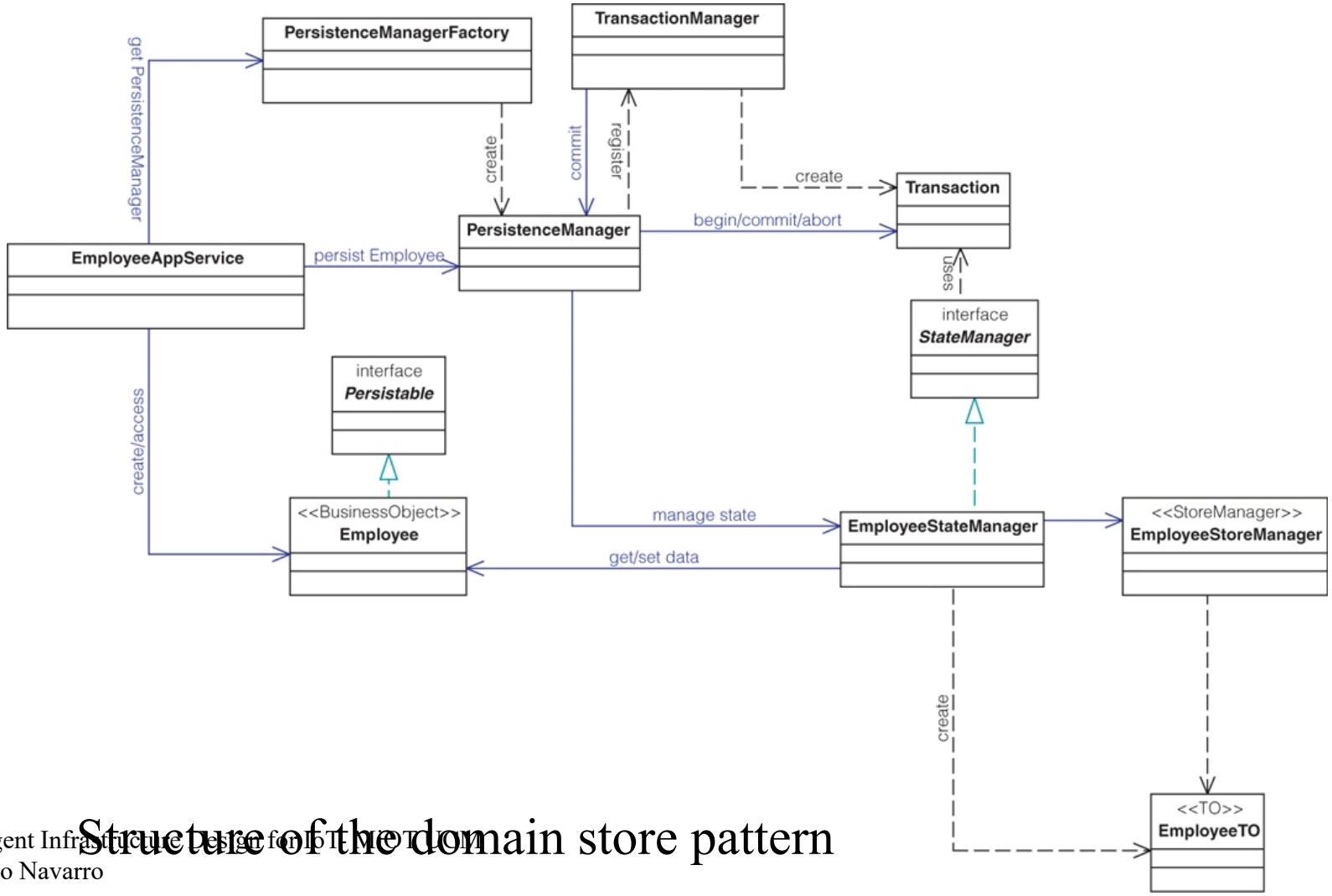


Persistence of a business object

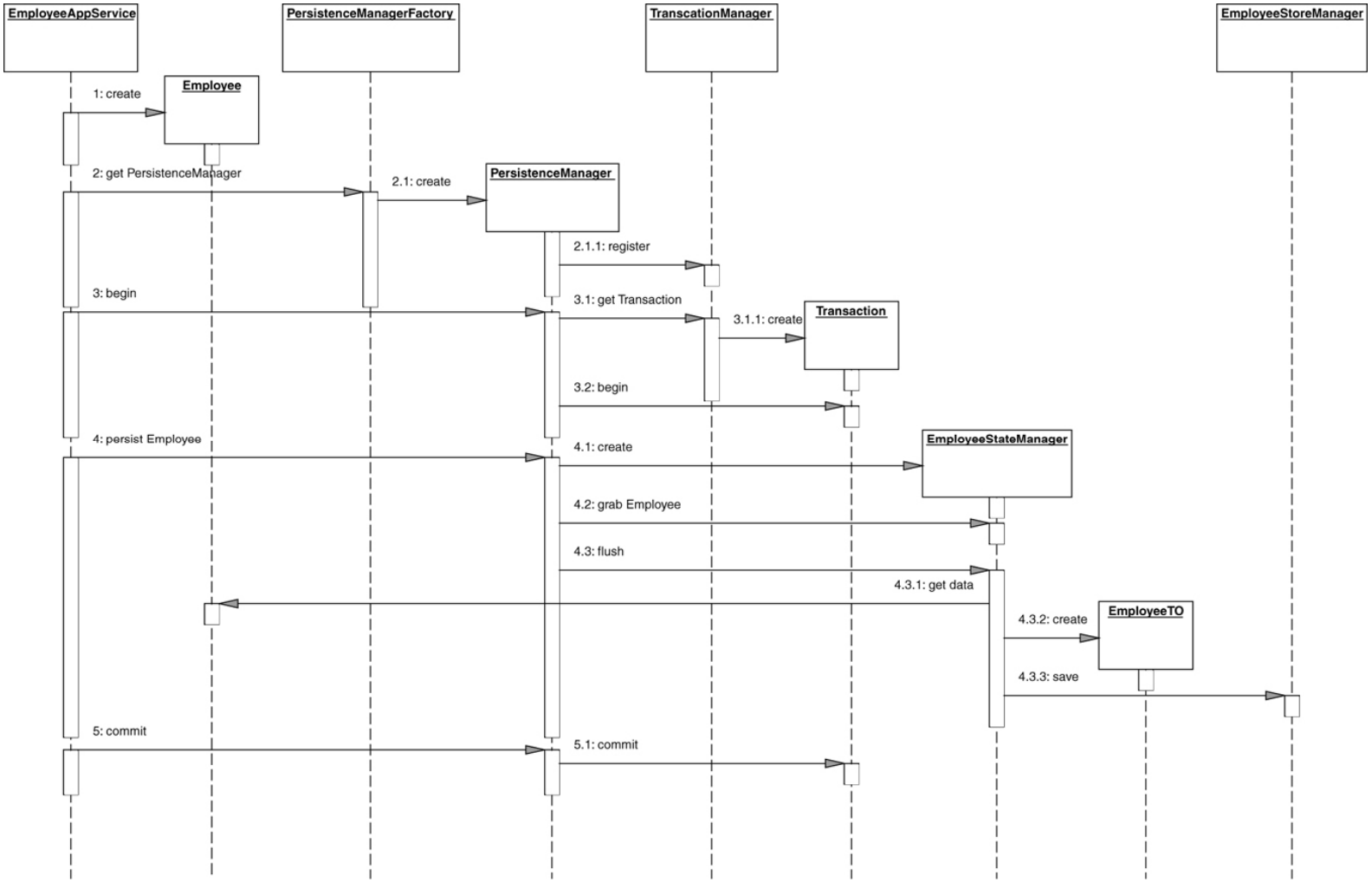
Domain store



Access to simple attributes of a business object



Structure of the domain store pattern



Interaction in the domain warehouse pattern

Note

- Double package structure:
  - Layers
  - Modules

	presentation	business	integration
users			
copies			
loans			
searches			

- Design subsystems/code packages: layer-driven, with replicated modules
  - presentation
    - users
    - copies
    - loans
    - searches
  - business
    - users
    - copies
    - loans
    - searches
  - integration
    - users
    - copies
    - loans
    - searches

Conclusions

- This topic introduces the basic patterns of multilayer architecture.
- We have seen a VERY basic multilayer. Missing:
  - Business objects
  - Dynamic load
  - Concurrence
  - Transactional