Arquitectura Software

3.2 Domain Layer Design

Introduction to Domain Layer Design

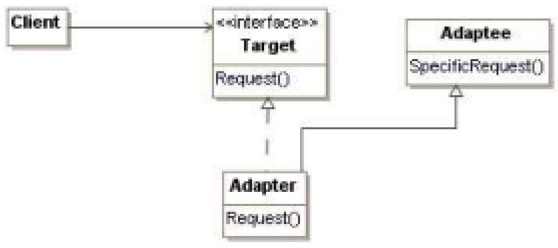
General purpose-patterns that may be applied to the domain layer.

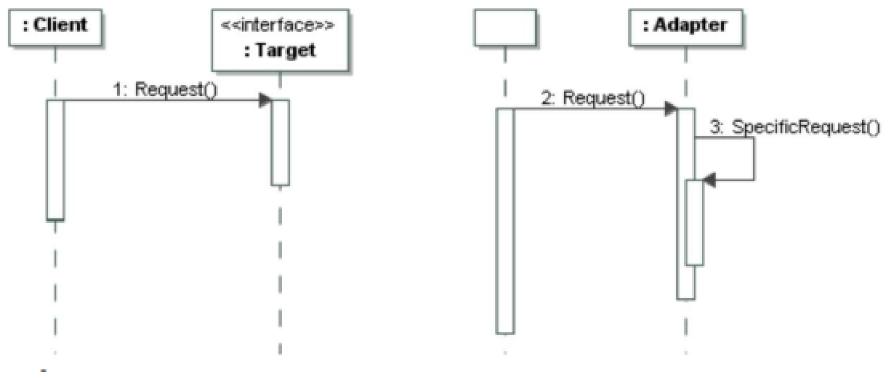
Patterns for Domain Layer

- **Adapter Pattern**:

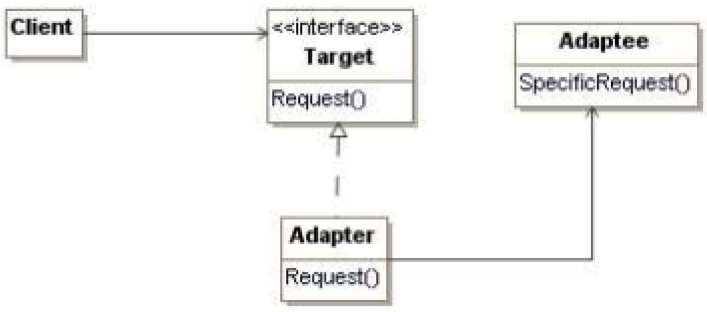
The interface of an existing class does not match the one needed. Solves the problem of having incompatible interfaces. Converts the interface of a class (*Adaptee class*) into another interface (*Adapter class*) clients (*Client class*) expects. There are two options:

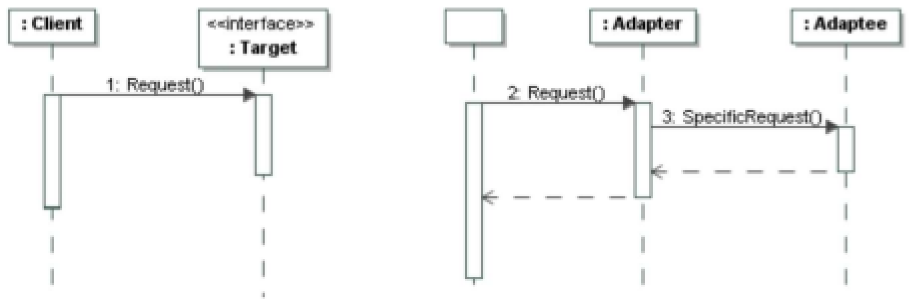
1. Class adapter:





2. Object adapter:

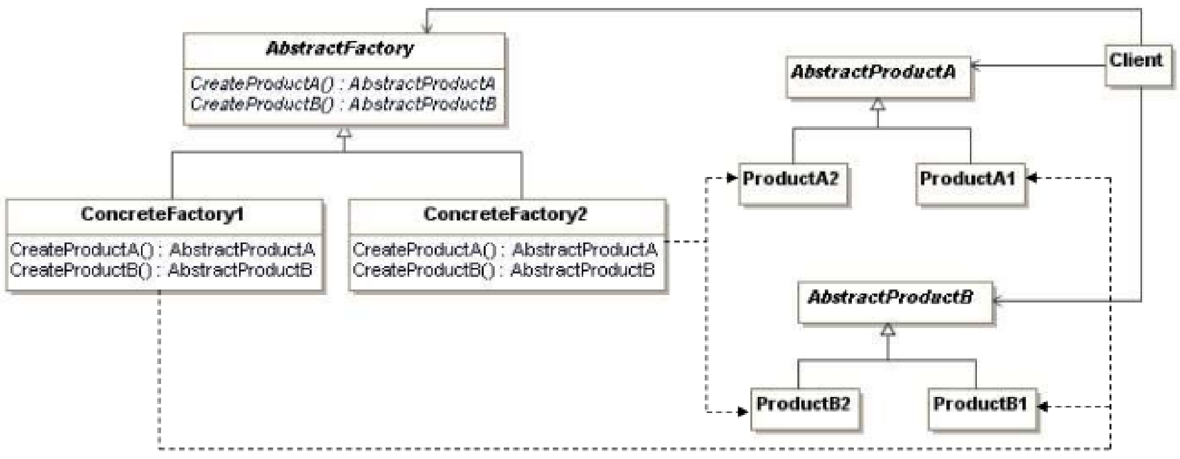


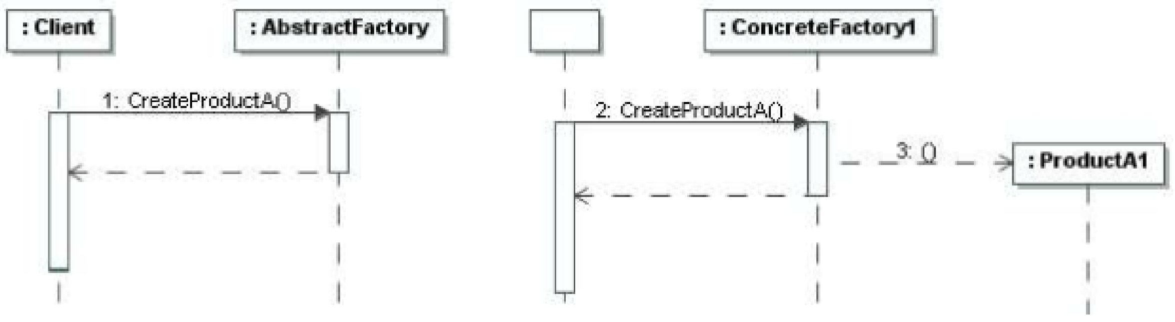


- **Abstract Factory:**

Systems that create, represent and compose a family of products that should be used together and that we do not want to reveal their implementations. Who should be responsible for creating objects when there are special considerations, such as a family of related or dependent objects?

Abstract factory provides an interface for creating families of related or dependent objects without specifying their concrete classes.


AbstractFactory
CreateProductA() : AbstractProductA
CreateProductB() : AbstractProductB
ConcreteFactory1
CreateProductA() : AbstractProductA
CreateProductB() : AbstractProductB
ConcreteFactory2
CreateProductA() : AbstractProductA
CreateProductB() : AbstractProductB
AbstractProductA
ProductA2
ProductA1
AbstractProductB
ProductB2
ProductB1
Client

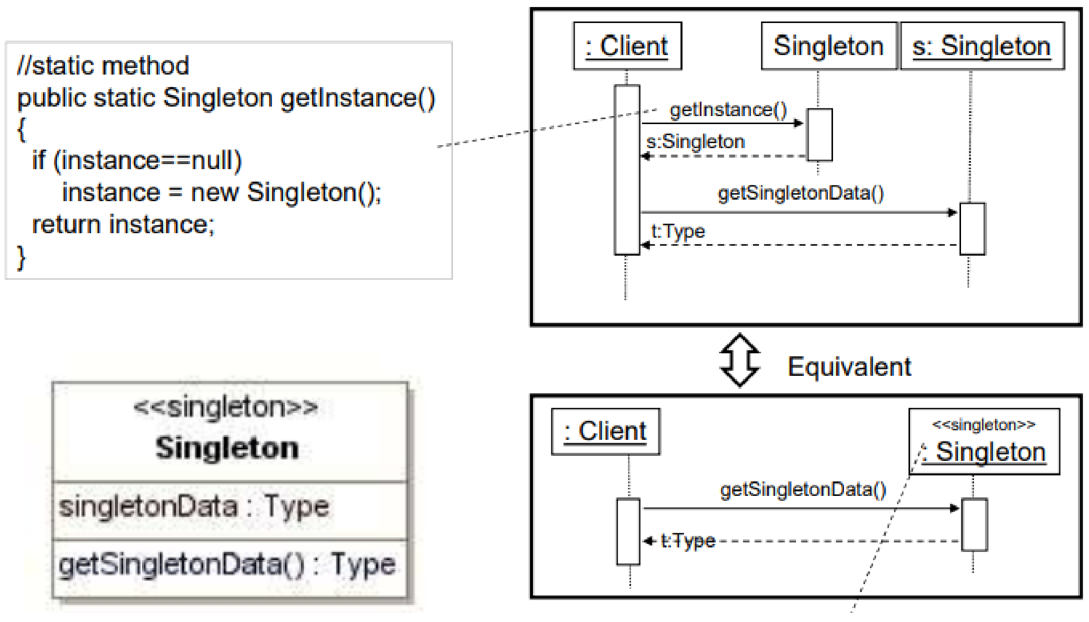


Simple or Concrete Factory is a variation of Abstract Factory Patterns where n object called Factory is the responsible for creating objects with a complex creation logic or for a better cohesion.

- **Singleton:**

Systems that haves classes with exactly one instance that must be accessible. There must be exactly one instance of a class, and it must be accessible to clients from a well-known point. A global variable makes an object accessible, but it doesn't keep you from instantiating multiple objects.

Define a class operation of the class that returns the singleton.

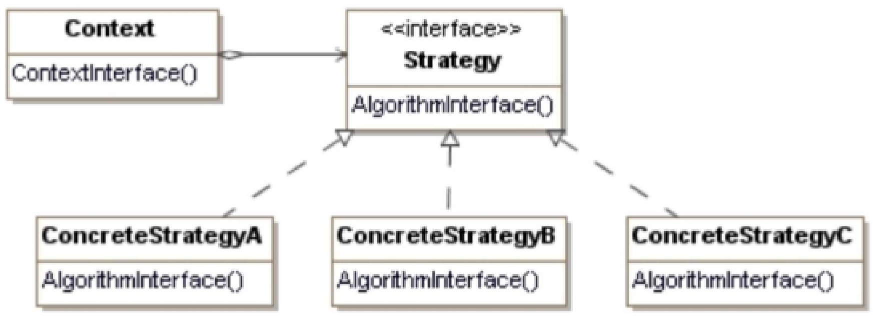


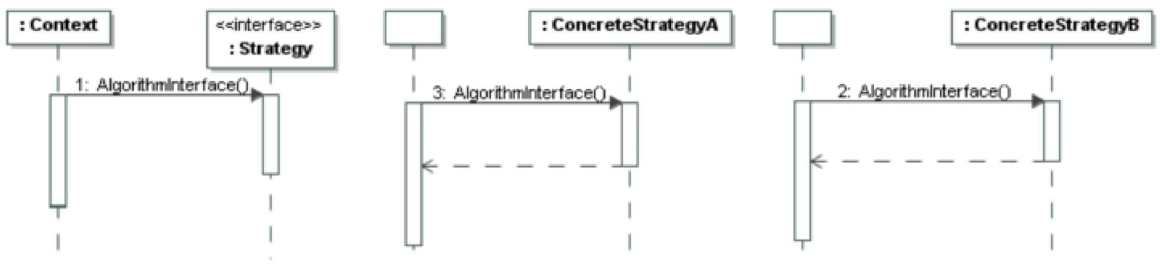
```
//static method
public static Singleton getInstance()
{
  if (instance==null)
     instance = new Singleton();
  return instance;
}
```

- **Strategy:**

Systems that have related classes differing only in their behaviour. How to design for varying, but related, algorithms or policies? How to design for the ability to change these algorithms or policies. Including these algorithms in the clients makes them bigger, harder to maintain and extend.

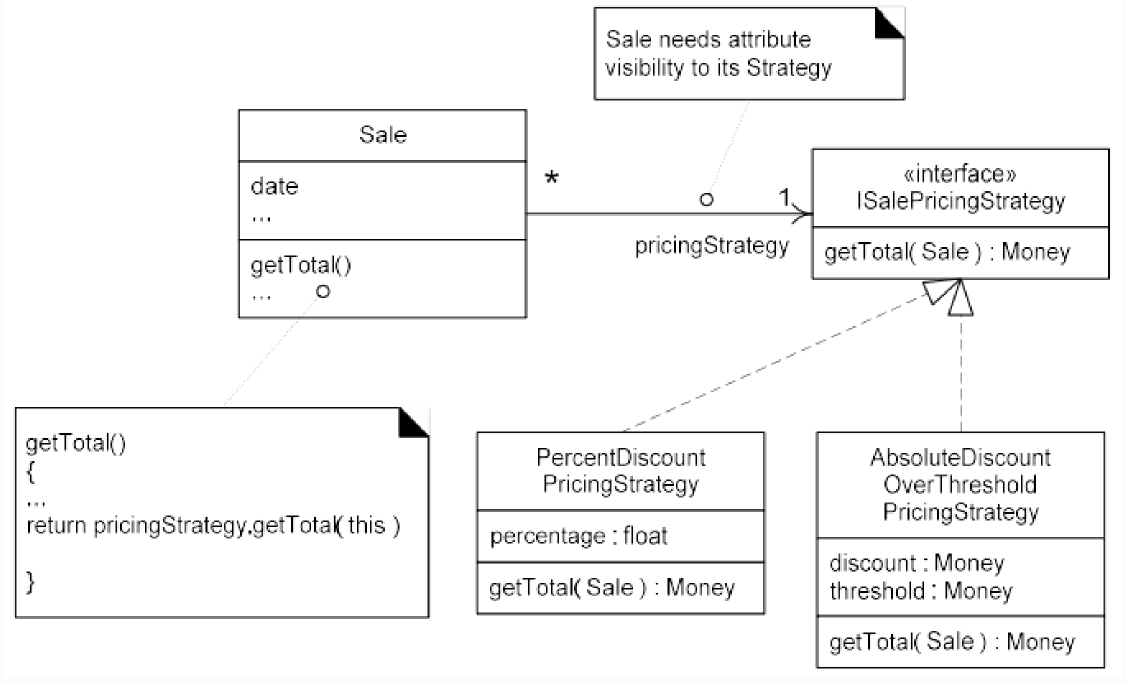
Define classes that encapsulate different algorithms. An algorithm that is encapsulated in this way is called a strategy.

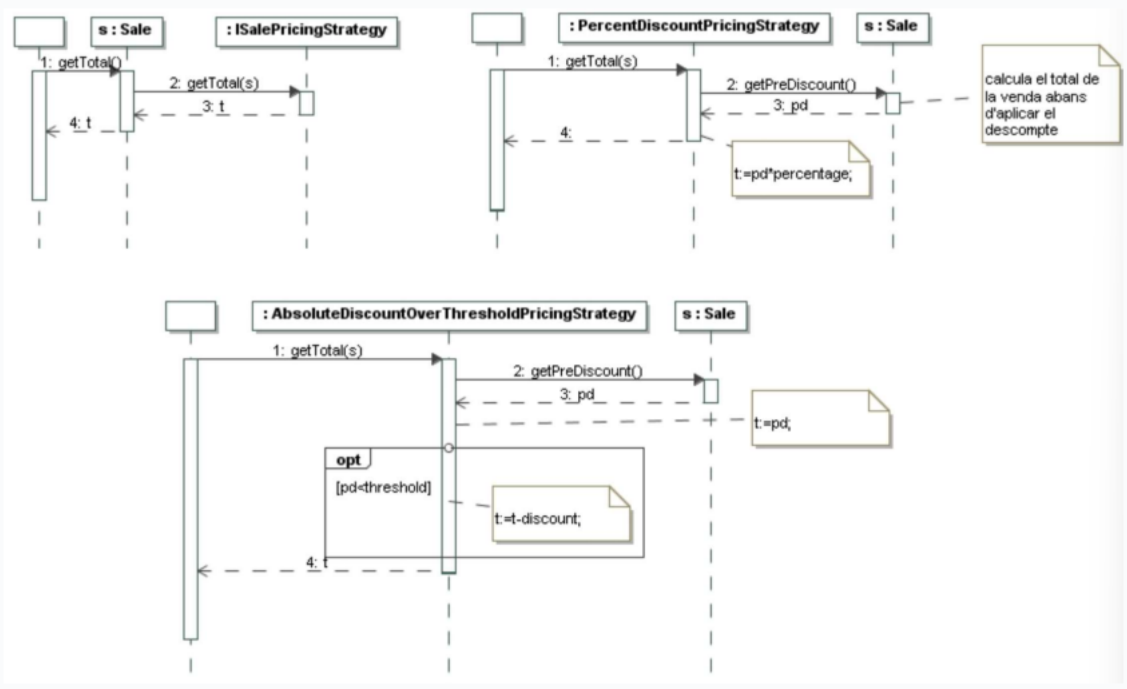




The *AlgorithmInterface* method is different for each ConcreteStrategy class. A similar sequence diagram for the ConcreteStrategyC.

Example:



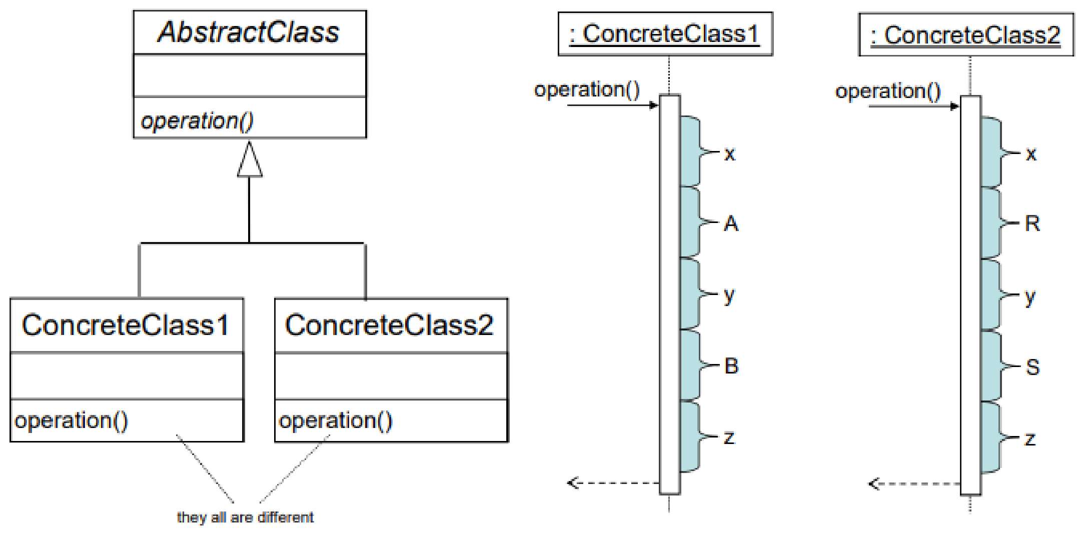


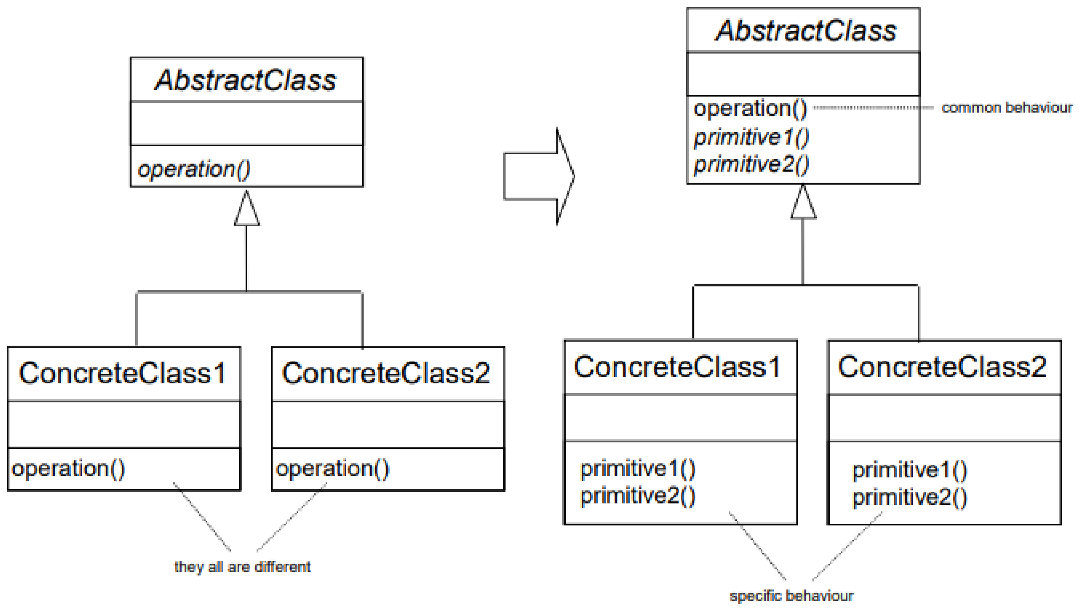
- **Template Method**:

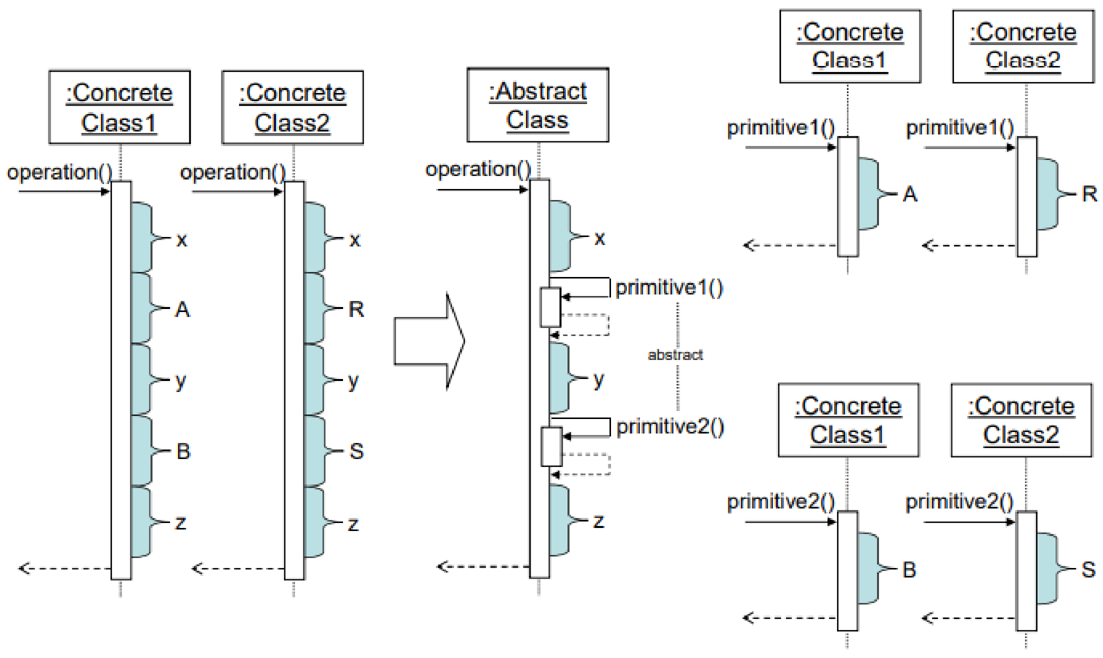
The definition of an operation in a hierarchy has some common behaviour to all subclasses but algo some specific behaviour for each of them. Replication the common behaviour in all subclasses requires code duplication and therefore a more costly maintenance.

To define the operation in the superclass, invoking abstract operations that are implemented as methods in the subclasses. The concrete operation is called *template* and the new operations primitives.

The operation at the superclass defines the common behaviour whilst the abstract operations identify the specific behaviour, that is described in each subclass.

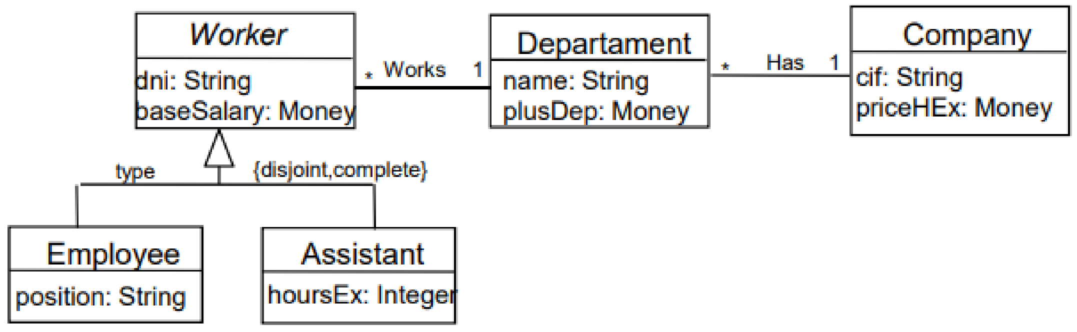






Example

Specification data conceptual model:

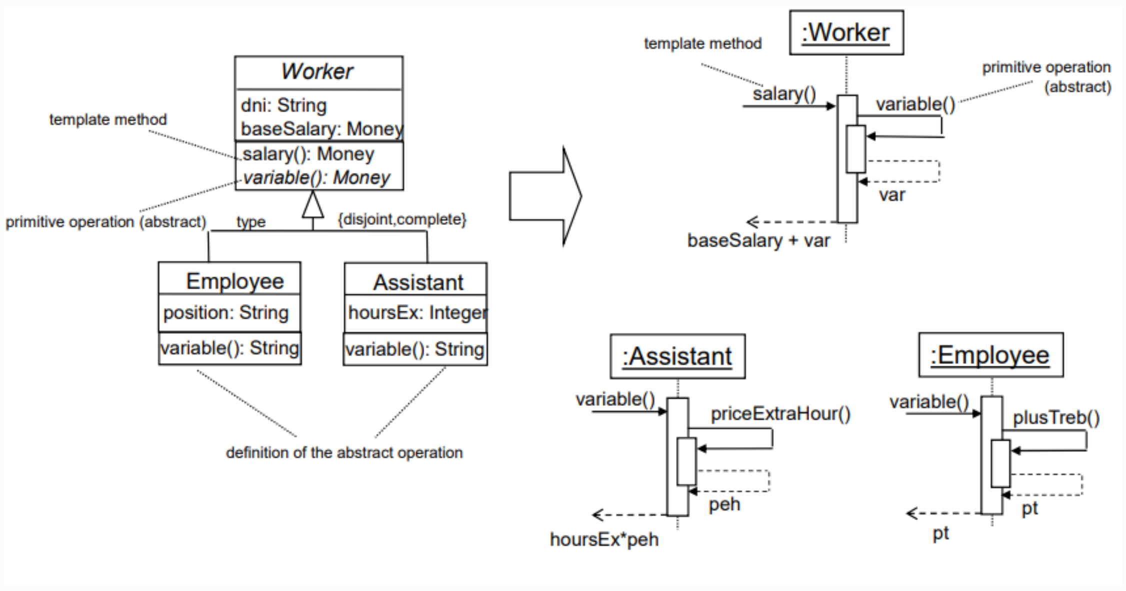


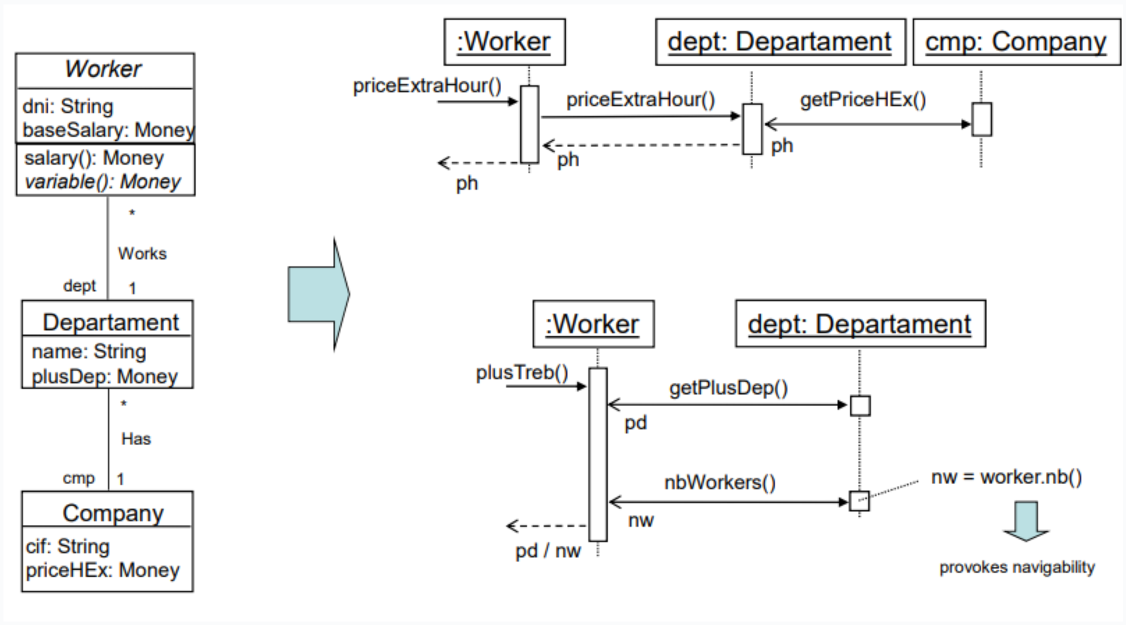
Textual Integrity Constraints:
Identifiers: (Worker, dni); (Company, cif)
There cannot exist two departments with the same name in the same company

We want to design an operation in class *Worker* to compute the salary that has to be paid to employees working in different companies:

$$salaryAssistant = baseSalaryWorker + hoursEx * priceHEx$$

$$salaryEmployee = baseSalaryWorker + plusDep/numberWorkers$$





3.3 Presentation Layer Design

Introduction

The design of the Presentation Layer includes two clearly differentiated tasks:

- External Design: definition of the interaction of the user with the software system. Designing the elements that the user will see, feel and touch when interacting with the system. Design of the *GUI*.
- Internal Design: definition of the interaction between the GUI and the Domain Layer.

External Design of the Presentation Layer

Is the definition of:

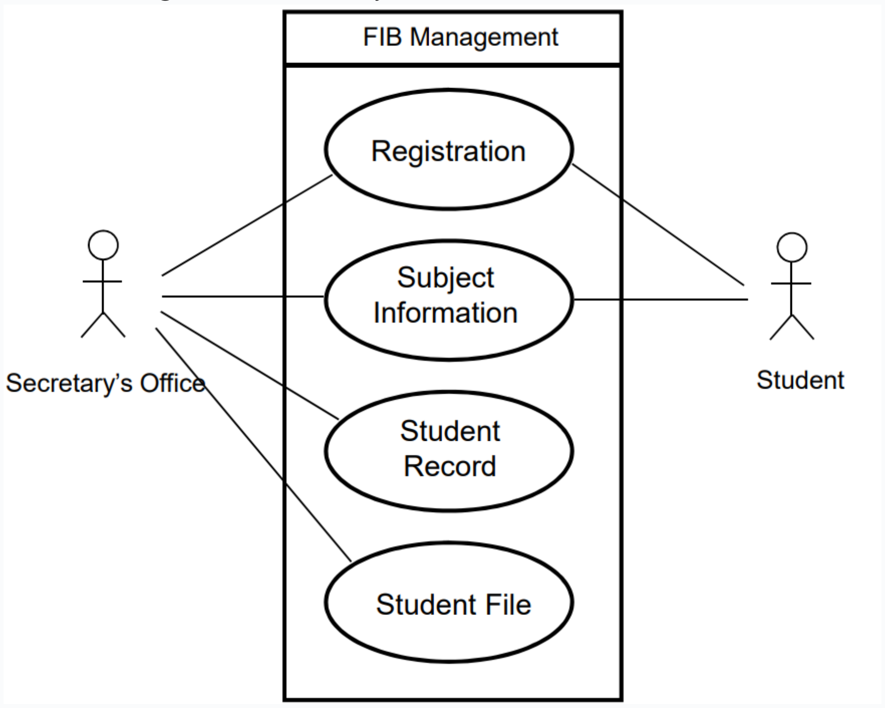
- Mechanisms that will allow the user to make requests to the system **->** Interaction mechanisms (*e.g.*, command language, function keys, pointing menus with mouse or touch-sensitive screen, ...).
- Ways to show the user the results of his requests **->** Mechanisms for the presentation of information (*e.g.*, graphic formats, images, textual, videos, ...).

There are three golden rules behind the principles of the user interface design:

- The user is in control: interaction modes, flexibility, cancelations, modifications, customization, ...
- Minimize memorization: reduce short term memory, default options, shortcuts, hierarchies, ...
- Maintain a consistent interface: inform about the context, consistency between windows, consistency between product families, ...

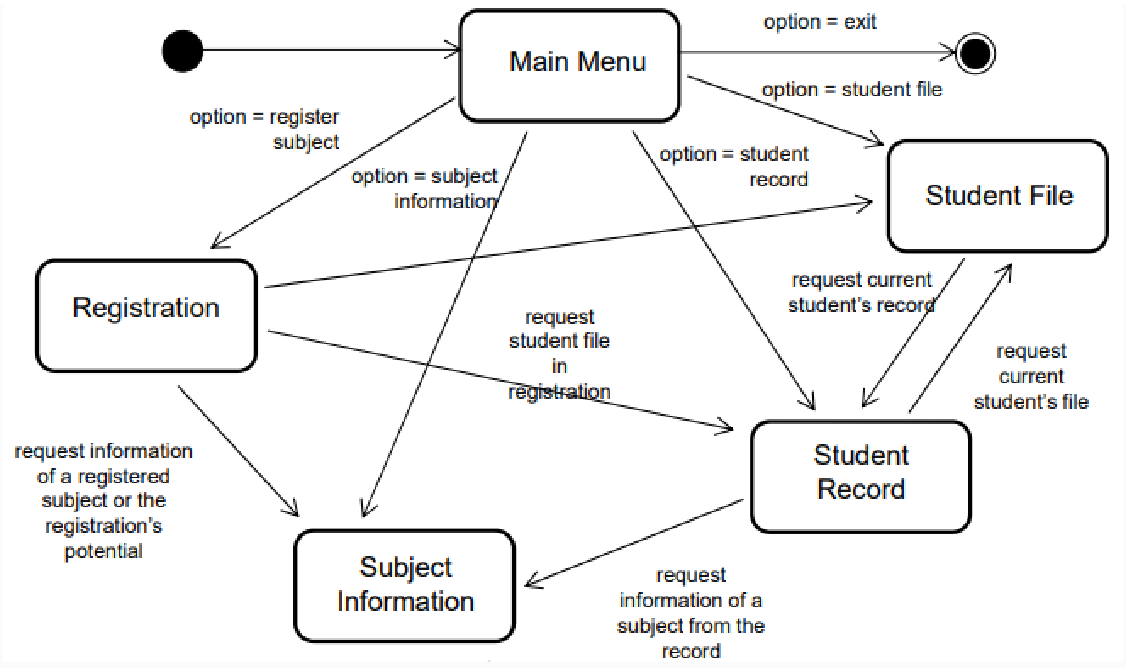
Working with events complicates the comprehension of the user interface, because we ignore which are the relevant ones, or under which conditions something is going to be triggered. Navigational maps represent the paths between screens, and offer a general perspective of the system.

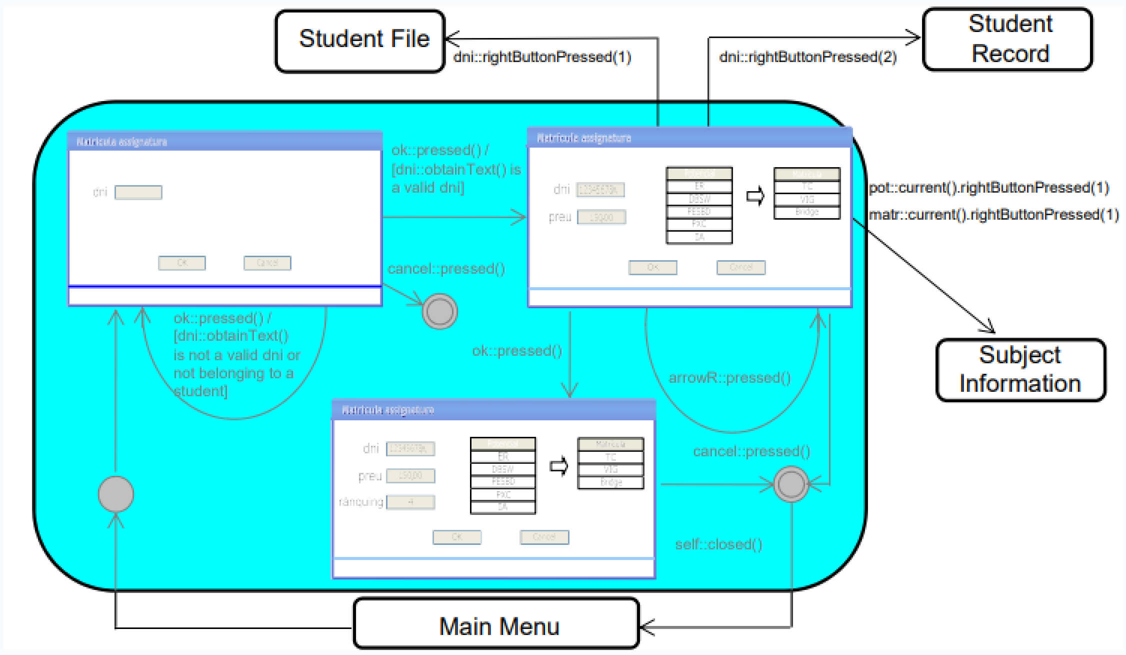
Use Case Diagram of the FIB's system:



System Navigational Map:

- High level description:


Main Menu
option = exit
option = student file
option = register subject
option = subject information
option = student record
Student File
Registration
request current student's record
request student file in registration
request current student's file
Student Record
request information of a registered subject or the registration's potential
Subject Information
request information of a subject from the record


Student File
dni::rightButtonPressed(1)
dni::rightButtonPressed(2)
Student Record
Matrícula assignatura
ok::pressed() / [dni::obtainText() is a valid dni]
cancel::pressed()
pot::current().rightButtonPressed(1)
matr::current().rightButtonPressed(1)
ok::pressed() / [dni::obtainText() is not a valid dni or not belonging to a student]
ok::pressed()
arrowR::pressed()
Subject Information
cancel::pressed()
self::closed()
dni
preu
rànquing
OK
Cancel
Main Menu

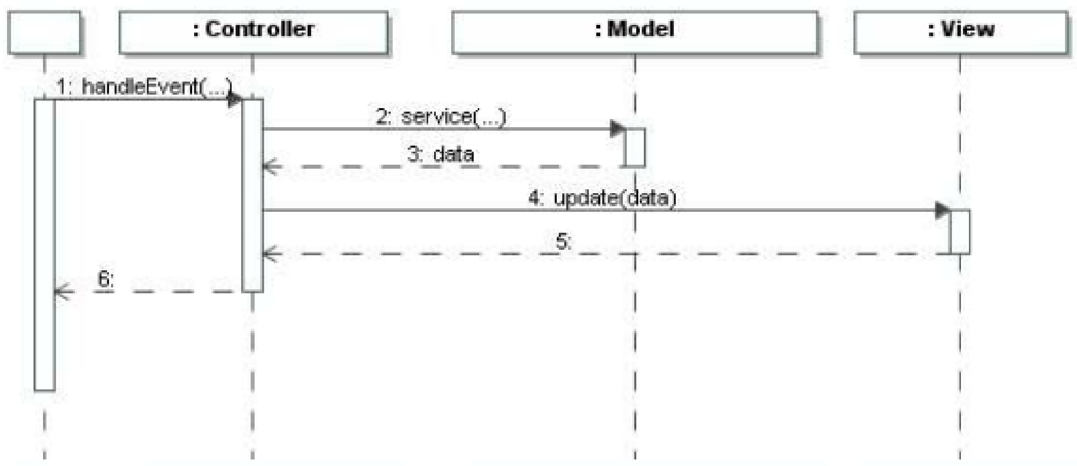
Internal Design of the Presentation Layer

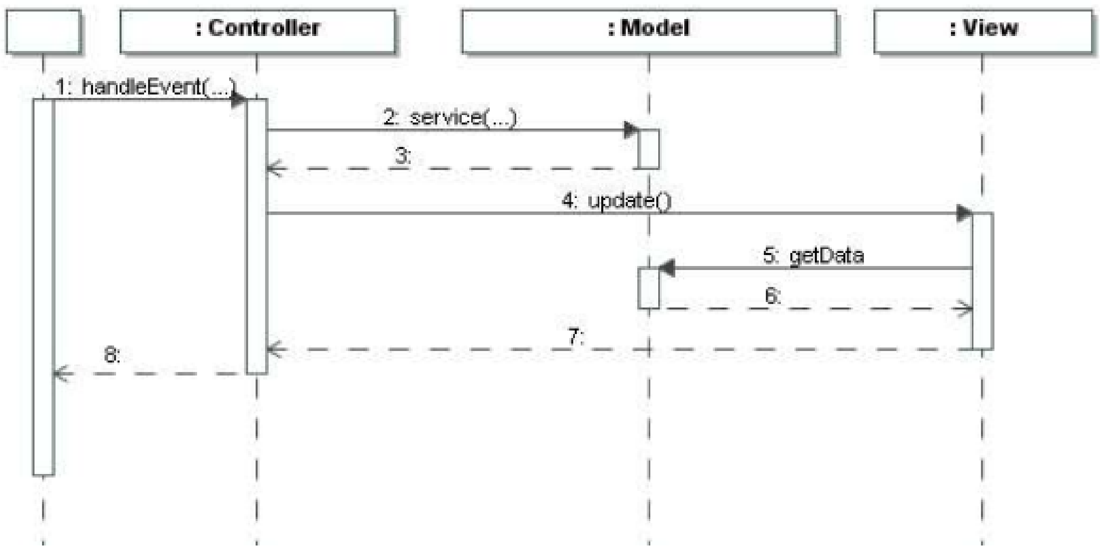
Is the receiver of the presentation events. Controls the communication of presentation events of the receiver and processes these events as well as identifying external ones. Allows the dialog with the Domain Layer (sends the external events that have to be processed and receives the corresponding answers). Presents the data (own or received from the Domain Layer) in the formats determined by the external design.

- Model-View-Controller Pattern:
    - Context: interactive software system with flexible user interfaces.
    - Problem: how to modularize user interface functionality of a software system so that it can be easily modified.
    - Forces to balance: user interface logic tends to change frequently. It is necessary to display the same data in different ways (if the user changes data in one view, the system must update data in the other views). User interface logic tends to be more device-dependent than business logic.

- Solution: separate the modeling of the domain, the presentation and the actions based on user input into three separate classes:
  - Model: manages the behaviour and data of the application domain, responds the requests for information about its state (view) and responds to instructions to change state (controller).
  - View: manages the display of information.
  - Controller: interprets the mouse and keyboard inputs from the user, informing the model and/or the view to change as appropriate.

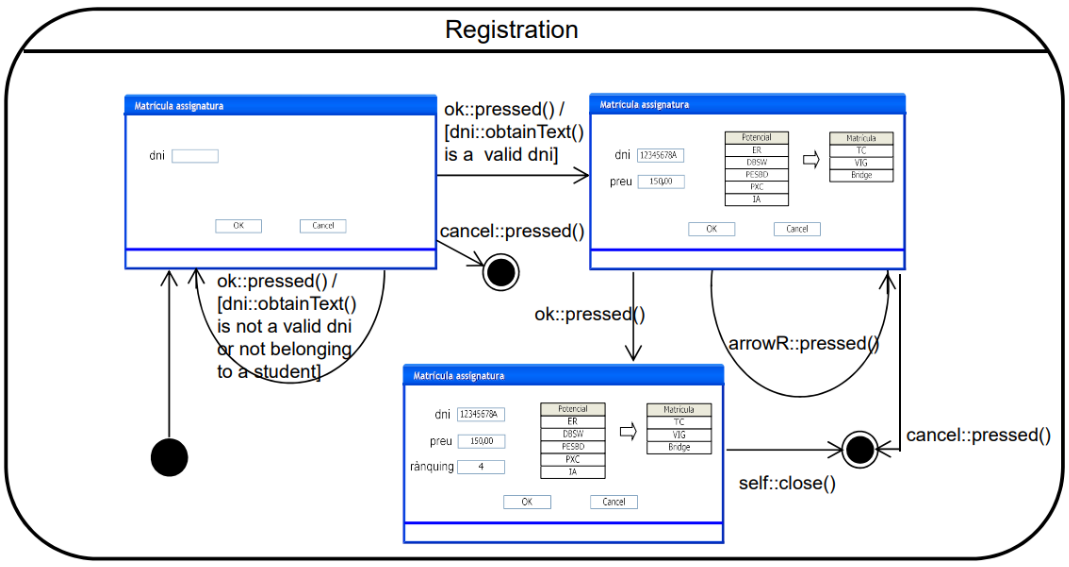
The Presentation Layer contains the views and the controllers. The model represents the Domain and the Data Layer.



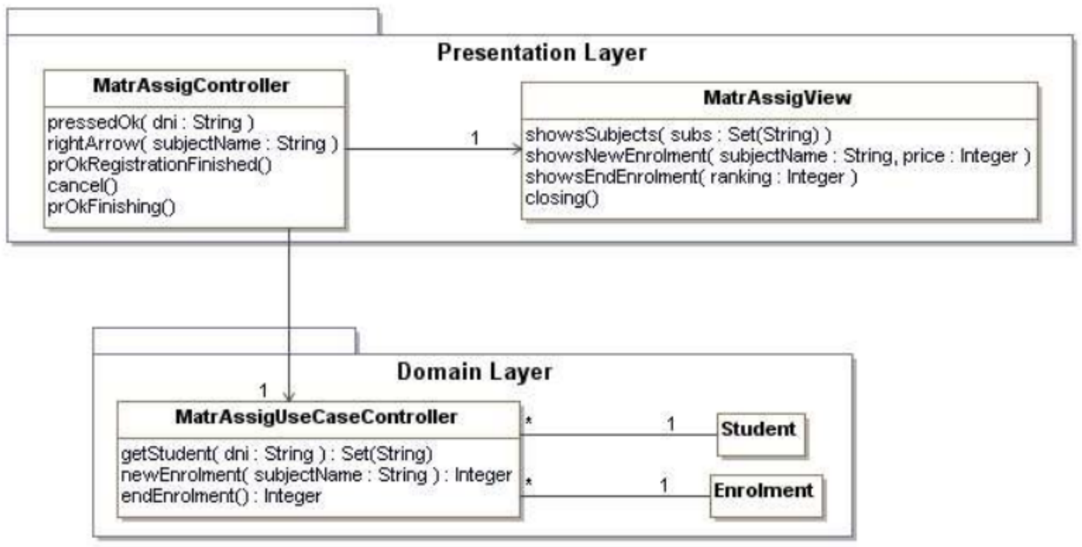


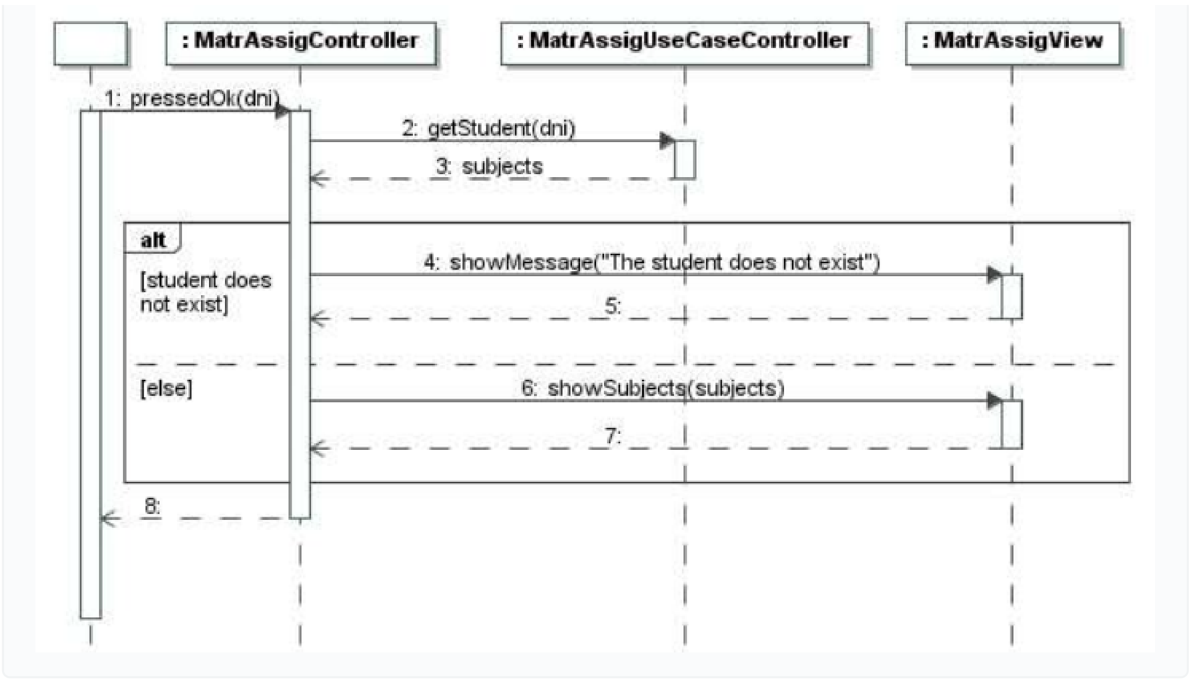
- Advantages: It may exists several view related to a model. At the execution time, it may be several open views. Views and controllers are easily reusable. High portability of the Presentation Layer. Low coupling between Domain and Presentation layers.
- Drawbacks: High coupling between views and controllers.

Example:



Presentation Layer	Domain Layer	Presentation Layer
Controller Operations	*Model Operations*	*View Operations*
pressedOk(dni)	getStudent(dni):Set(string)	showSubjects(subs:Set(String))
rightArrow(subjectName)	newEnrolment(subjectName):Integer	showNewEnrolment(subjectName, price)
prOkRegistrationFinished()	endEnrolment():Integer	showEndEnrolment(ranking)
cancel()	-	closing()
prOkFinishing()	-	closing()

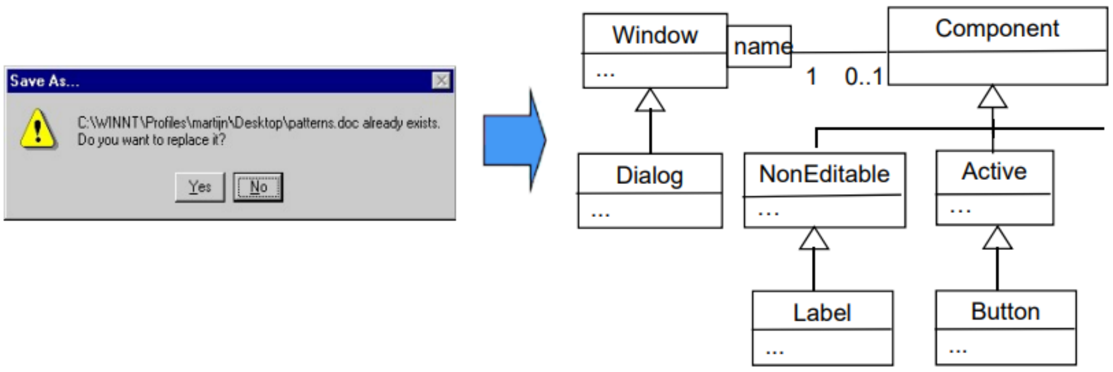

Presentation Layer
MatrAssigController
pressedOk(dni : String)
rightArrow(subjectName : String)
prOkRegistrationFinished()
cancel()
prOkFinishing()
1
MatrAssigView
showsSubjects(subs : Set(String))
showsNewEnrolment(subjectName : String, price : Integer)
showsEndEnrolment(ranking : Integer)
closing()
Domain Layer
1
MatrAssigUseCaseController
getStudent(dni : String) : Set(String)
newEnrolment(subjectName : String) : Integer
endEnrolment() : Integer
*
1
Student
*
1
Enrolment



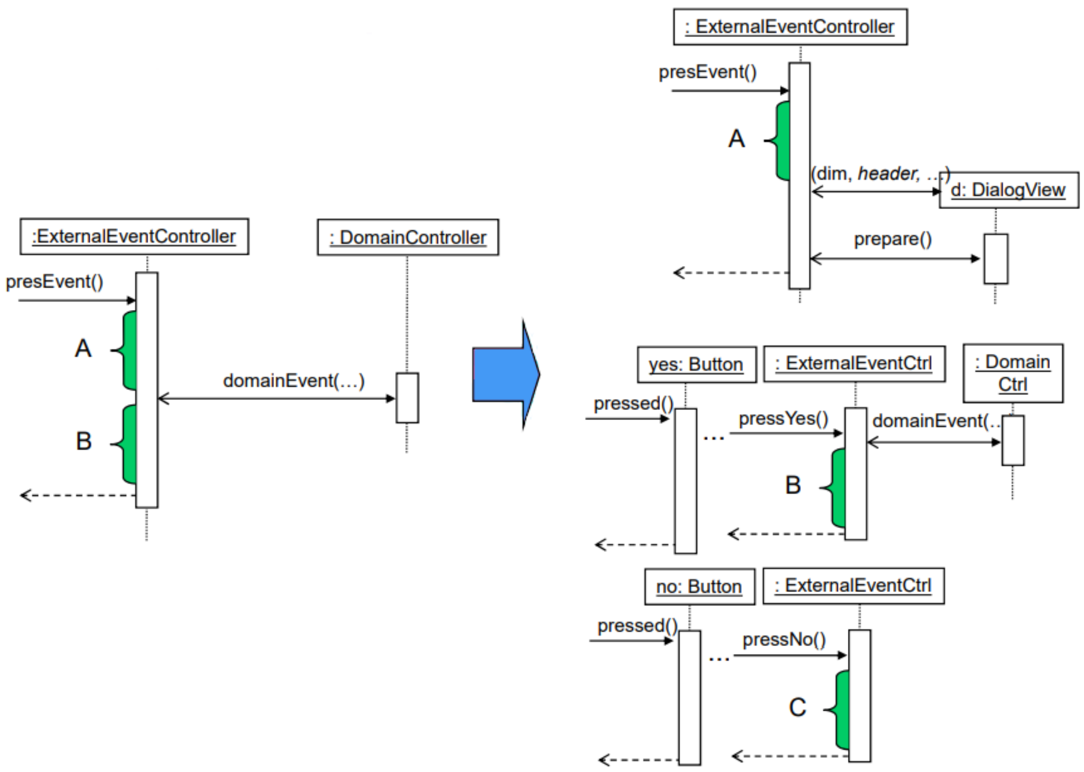
Representation Layer Patterns

They allow tackling the design of the Presentation Layer with the classic context-problem-solution approach. From the point of view of the external design, the patterns are related with the principles of the interface design and from the point of view of the internal design, the patterns follow the MVC pattern and the principles of the OO architectural pattern.

- Protection Pattern:
  - Context: there are several actions that have important (and irreversible) effects on the system, and some, once completed, are very costly to undo.
  - Problem: the user can accidentally select an option that has irreversible effects or that is very costly to undo.
  - Solution: add an extra level of protection in the function (double confirmation). The user has to confirm explicitly the chosen option (default option is to not execute the function). Consider the possibility of allowing to configure (deactivate) the behaviour.


Save As...

C:\WINNT\Profiles\martijn\Desktop\patterns.doc already exists.
Do you want to replace it?



3.4 Data Layer Domain