Object-Oriented Analysis and Design with UML2 and Rational Software Modeler

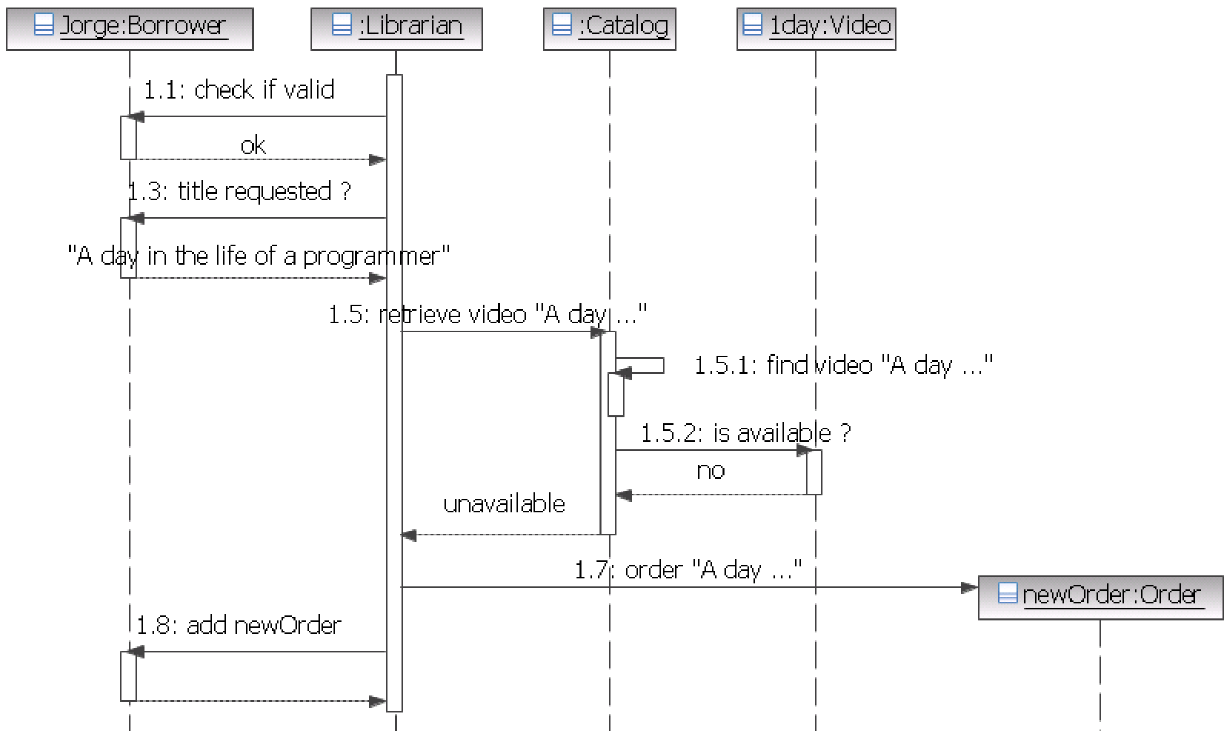
Student Workbook

V1.1 (2007-11-09)

Table of contents

Lab 1 – Sequence Diagram

Interpret the following sequence diagram:



Lab 2 – Class Diagrams

Task 2.1: Modeling graphs, points and connectors

Create a class diagram to model the following concepts:

1. A drawing is composed of several graphs.
2. A graph is composed of one or more points, that may be connected or not.
3. One connector connects two points. There can be only one connector between two points.
4. Every point has a color.
5. Every connector has a color.
6. All the connectors of a given graph have the same color.
7. All the points of a given graph have the same color.
8. Destroying a point also destroys the associated connectors.

You can complete these statements with your own assumptions.

Task 2.2: Modeling a family tree

Model in a class diagram a family tree: the fact that one has parents and possibly children. Add relevant attributes.

Focus on the “biological” family ties – one has always two parents, even if they are deceased.

Consider creating a first model without using generalizations, and a second one with generalizations.

Hint: It might be necessary to add constraints to accurately model the family tree.

Task 2.3: Modeling a file system

Model a simple file system: a directory may contain other directories and/or files.

Implement the *destroy()* operation for both directories and files. Assume that a file or directory can destroy itself by calling a hypothetical system function *System.destroy(self)*. However, in the case of a directory, this function can be called only if the directory is empty. For a file, the *destroy()* operation looks like (in Java-like pseudo-code):

```
public class File {
      public void destroy() {
            System.destroy(self);
      }
}
```

Solutions to these exercises can be found in appendix 1.

Lab 3 – Requirements Management

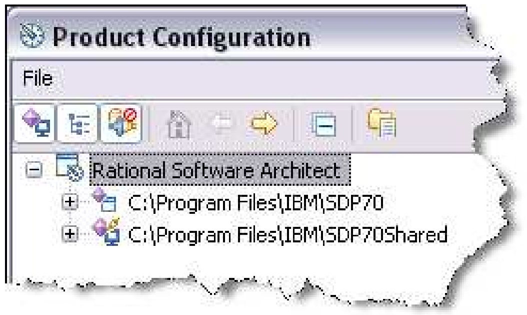
Task 3.1: Preliminary setup

For the remaining labs, the following files are provided:

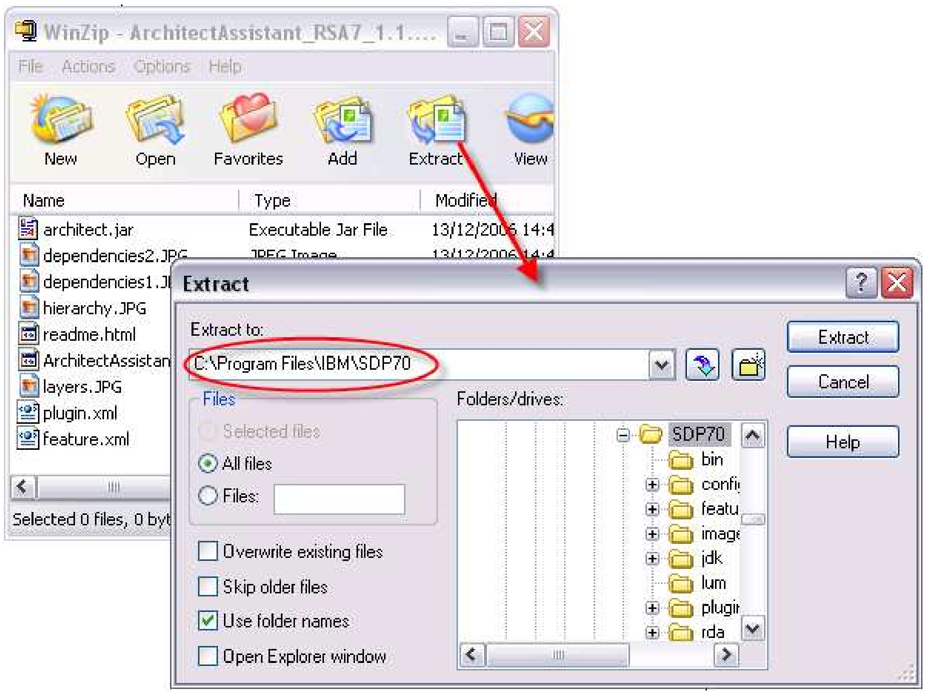
- *ArchitectAssistant_xxx.zip* contains a plug-in that you will be using in some of the labs to automate some operations.
- *JDBCPattern_xxx.zip* is another plug-in that provides a pattern that you will use during the design phase.
- *PayrollSystem.zip* contains the solutions to the different labs (starting from lab 3). <u>Please do NOT open the RSM/RSA models before being told to</u>.

Installing the plug-ins (*ArchitectAssistant* and *JDBCPattern*):

1. Unzip *ArchitectAssistant_xxx.zip* and *JDBCPattern_xxx.zip* into a valid RSM extension location:
    a. An extension location is where plug-ins are installed. You can see all extension locations by using the command *Help > Software Updates > Manage Configuration* (this may take a long time). The snapshot below shows the default configuration settings: in this example, you should use *C:\Program Files\IBM\SDP70* as the *C:\Program Files\IBM\SDP70Shared* location is not updatable.


Product Configuration
File
Rational Software Architect
C:\Program Files\IBM\SDP70
C:\Program Files\IBM\SDP70Shared

    b. In the example below, the contents of the zip files will be extracted into the *C:\Program Files\IBM\SDP70* directory:



Name	Type	Modified
architect.jar	Executable Jar File	13/12/2006 14:4
dependencies2.JPG	JPEG Image	13/12/2006 14:4
dependencies1.JI		
hierarchy.JPG		
readme.html		
ArchitectAssistan		
layers.JPG		
plugin.xml		
feature.xml		

Selected 0 files, 0 byt

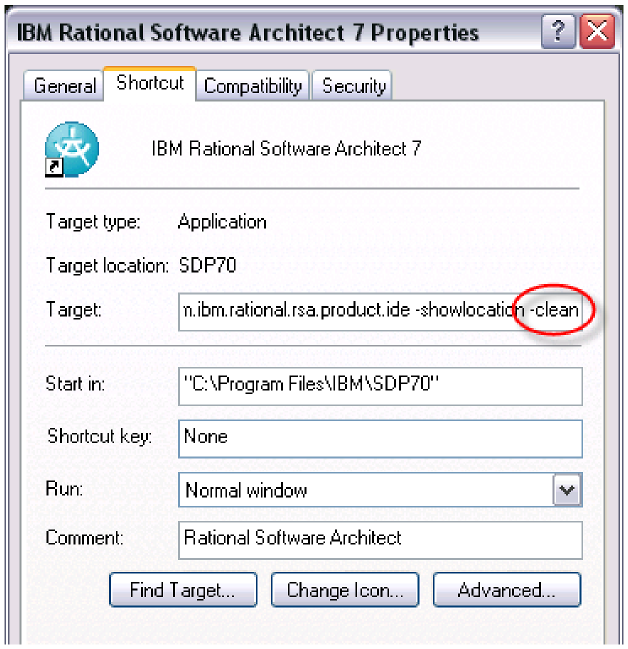
Extract to: C:\Program Files\IBM\SDP70

Files: Selected files [✓ All files] Files:

☐ Overwrite existing files
☐ Skip older files
☑ Use folder names
☐ Open Explorer window

Folders/drives: SDP70, bin, confi, featu, image, jdk, lum, plugi, rda

2. To ensure that RSA/RSM picks up the newly added plug-ins, you must restart RSA/RSM with the –*clean* option
   a. To add the –*clean* option, create (or edit) the RSM/RSA shortcut and add "–clean" as shown below:



IBM Rational Software Architect 7

Target type: Application

Target location: SDP70

Target: n.ibm.rational.rsa.product.ide -showlocation -clean

Start in: "C:\Program Files\IBM\SDP70"

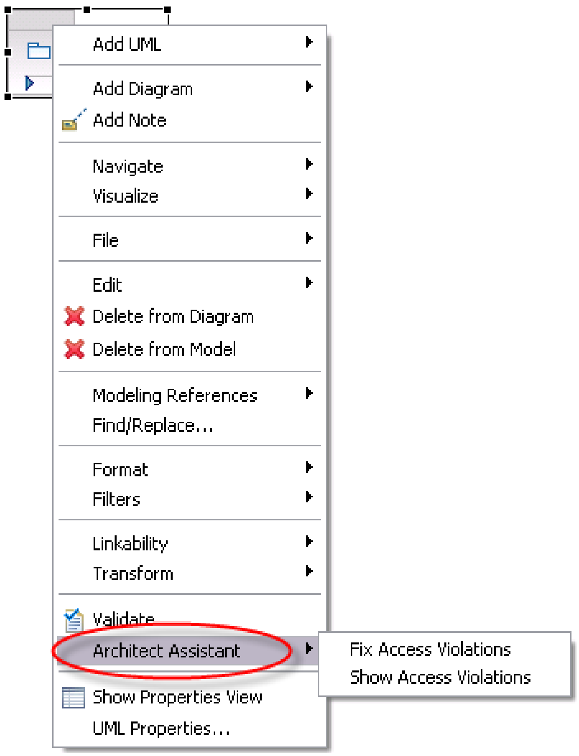
Shortcut key: None

Run: Normal window

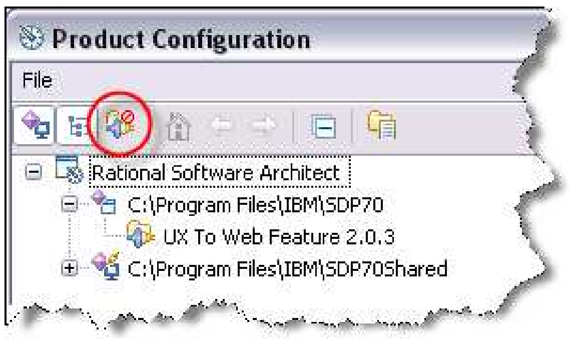
Comment: Rational Software Architect

   b. Restart RSM/RSA.
   c. Verify that the *ArchitectAssistant* plug-in was correctly added. (We will check the *JDBCPattern* plug-in at a later time.) Right-click on any UML package or

model. You should have an *Architect Assistant* submenu as shown below: (please note that you might see more entries in the *Architect Assistant* menu than shown below – this is normal)

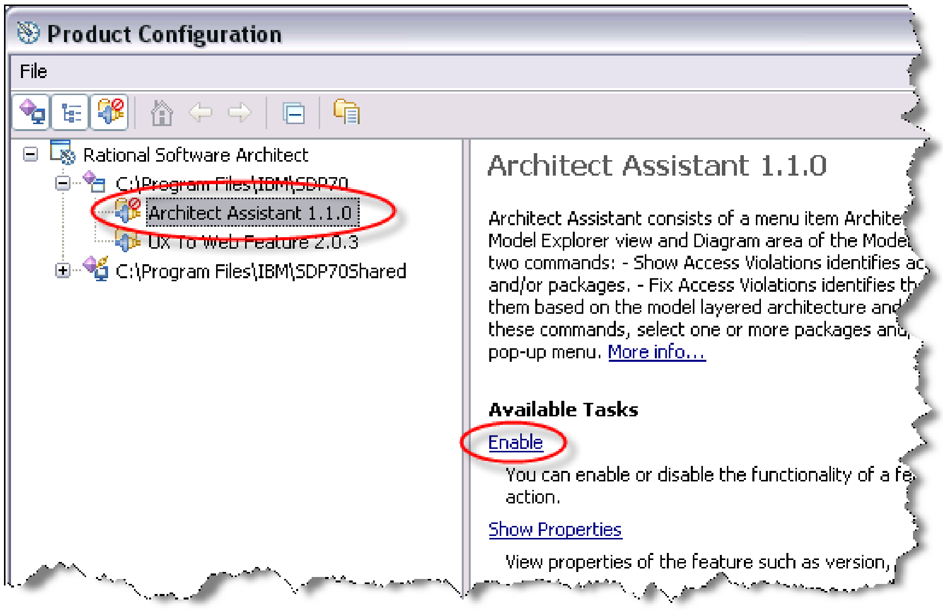


3. <u>If the *Architect Assistant* submenu is not available, the plug-in may be installed but not enabled</u>:
   a. Select *Help* > *Software Updates* > *Manage Configuration* (this may take a long time).
   b. Open the Eclipse location where you installed your plug-in (*C:\Program Files\IBM\SDP70* in our example).
   c. If you do not see an entry *Architect Assistant x.y.z*, press the *Show Disabled Features* icon as shown below:


Product Configuration
File
Rational Software Architect
C:\Program Files\IBM\SDP70
UX To Web Feature 2.0.3
C:\Program Files\IBM\SDP70Shared

   d. If you now see the *Architect Assistant x.y.z* entry with the symbol , the feature is disabled. At this point, you simply need to click on *Enable* as shown below and restart the workspace as requested. If you don't see the *Architect*

Assistant x.y.z entry, it is likely you have not extracted the plug-in to the correct location. Go back to step 1!


Product Configuration
File
Rational Software Architect
C:\Program Files\IBM\SDP70
Architect Assistant 1.1.0
Ux To Web Feature 2.0.3
C:\Program Files\IBM\SDP70Shared

Architect Assistant 1.1.0

Architect Assistant consists of a menu item Archite Model Explorer view and Diagram area of the Model two commands: - Show Access Violations identifies ac and/or packages. - Fix Access Violations identifies th them based on the model layered architecture and these commands, select one or more packages and pop-up menu. More info...

Available Tasks

Enable

You can enable or disable the functionality of a fe action.

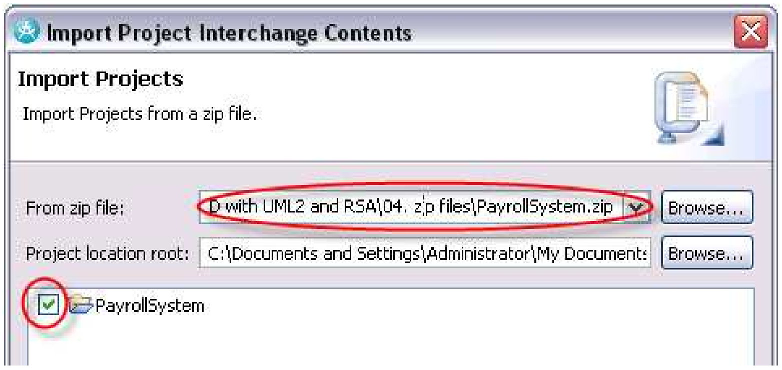
Show Properties

View properties of the feature such as version,

4. Remove the –*clean* option from the shortcut (or create a separate shortcut).

Importing the Payroll System solutions (RSA only – for RSM, see next section):

1. *File > Import … > Project Interchange > Next*
2. In the *Import Project Interchange Contents* dialog:
   a. Select *PayrollSystem.zip* using the *Browse* button to the right of the field *From zip file:*
   b. Check the *PayrollSystem* project and click *Finish*



Import Projects

Import Projects from a zip file.

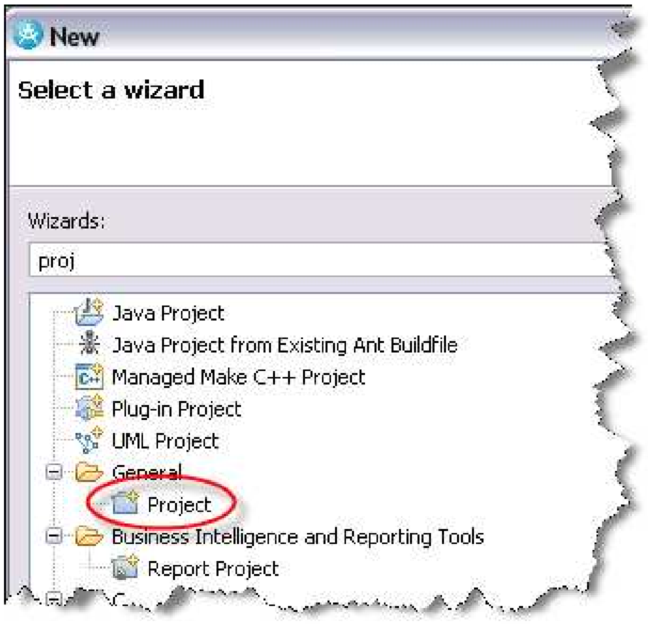
From zip file: D with UML2 and RSA\04. zip files\PayrollSystem.zip

Project location root: C:\Documents and Settings\Administrator\My Document:

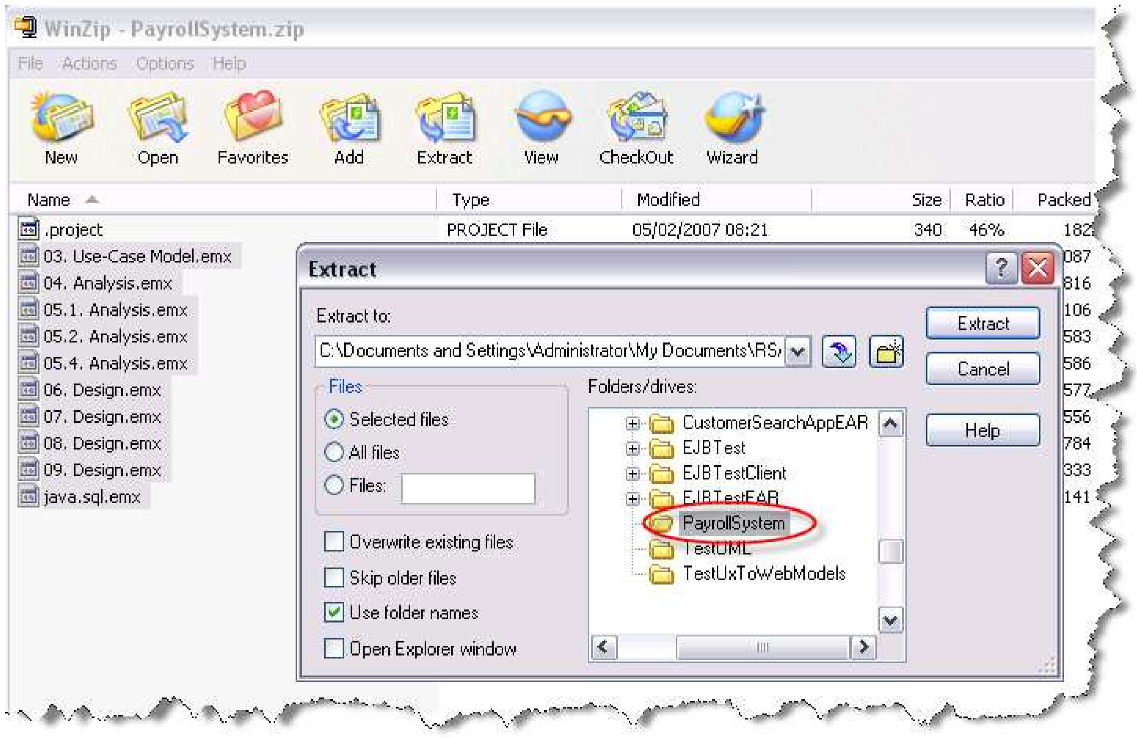
☑ PayrollSystem

Importing the Payroll System solutions (RSM – also works with RSA):

1. If it is not already the case, start RSM (or RSA) and go to the *Modeling* perspective.
2. Create a new project called *PayrollSystem* using the *Project* wizard:



3. Open the zip file, select all the .emx file (see below) and extract them into the directory where your project was created (check out the full path of the project in the *Properties* window):



Name	Type	Modified	Size	Ratio	Packed
.project	PROJECT File	05/02/2007 08:21	340	46%	182
03. Use-Case Model.emx					087
04. Analysis.emx					816
05.1. Analysis.emx					106
05.2. Analysis.emx					583
05.4. Analysis.emx					586
06. Design.emx					577
07. Design.emx					556
08. Design.emx					784
09. Design.emx					333
java.sql.emx					141

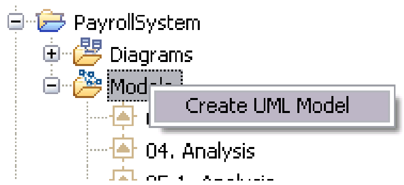
Extract to: C:\Documents and Settings\Administrator\My Documents\RS

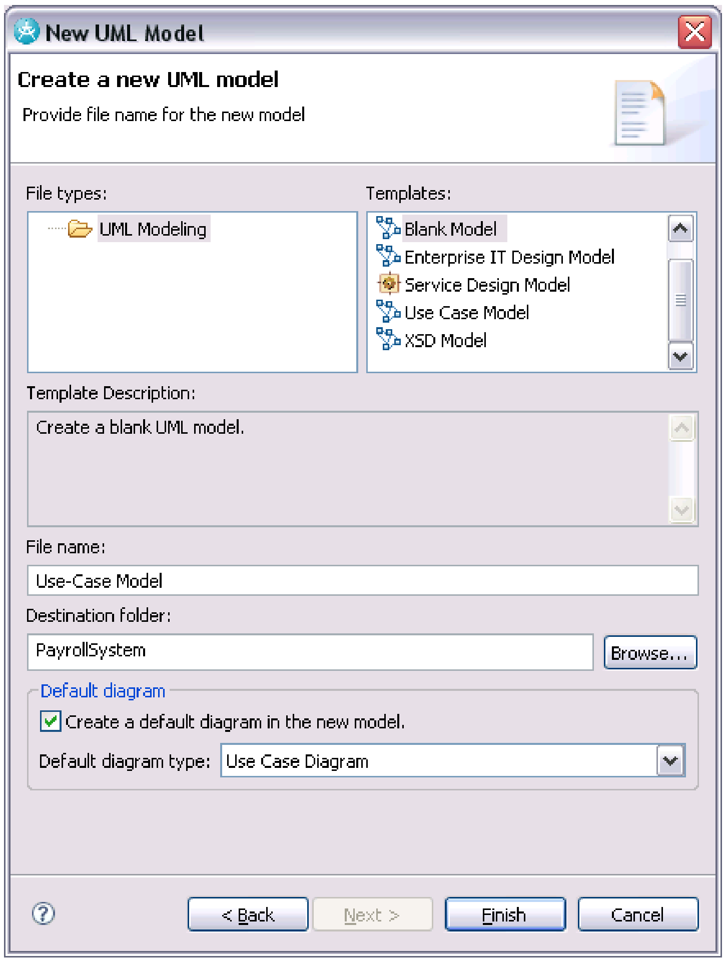
4. In RSM, right-click the *PayrollSystem* project and select the *Refresh* command to see the files in the *Project Explorer*.

Task 3.2: Identify the actors and use cases of the payroll system

1. If it is not already the case, start RSM (or RSA) and go to the *Modeling* perspective.

2. Create in the *PayrollSystem* project a new blank model named *Use-Case Model* with *Use Case Diagram* as the default diagram type.





Create a blank UML model.

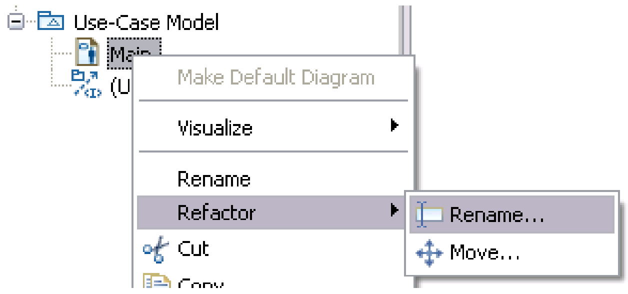
File name: Use-Case Model

Destination folder: PayrollSystem

✓ Create a default diagram in the new model.

Default diagram type: Use Case Diagram

3. Rename the diagram *Main* as *Global View*:



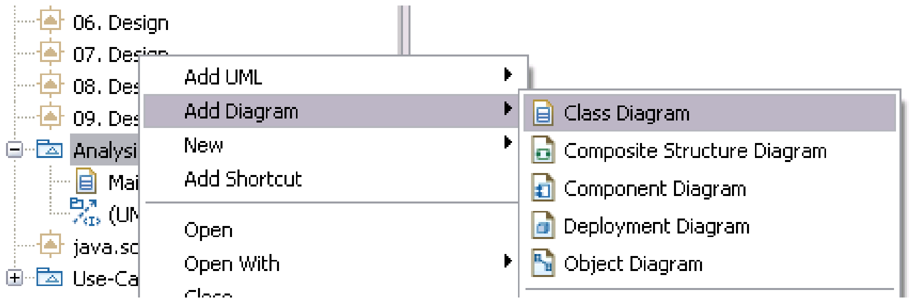
4. Create and briefly describe the actors and use cases for the Payroll System in this diagram, based on the problem statement, glossary and supplementary specifications provided in the Appendix 3 of this document.

Lab 4 – Architectural Analysis

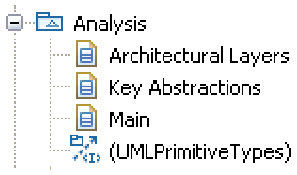
Task 4.1: Creating the Analysis Model

1. In the *PayrollSystem* project, create a new blank model, named *Analysis*, and with class diagram as the default diagram type.

2. Create in the *Analysis* model two class diagrams respectively names *Architectural Layers* and *Key Abstractions*.

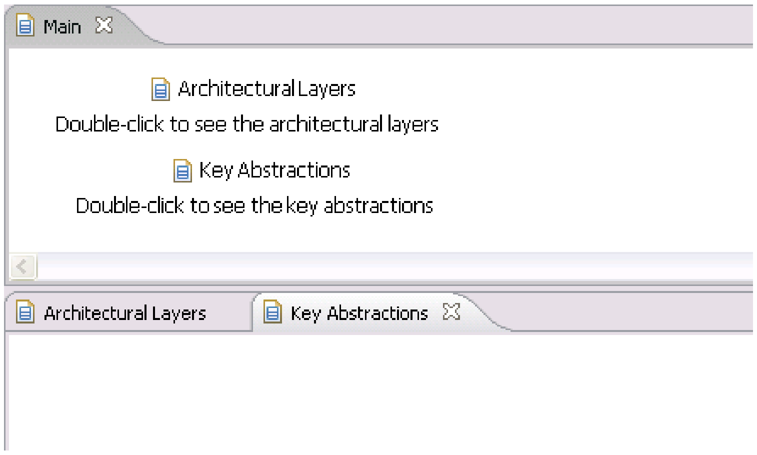


The analysis model should look like this in the project explorer:



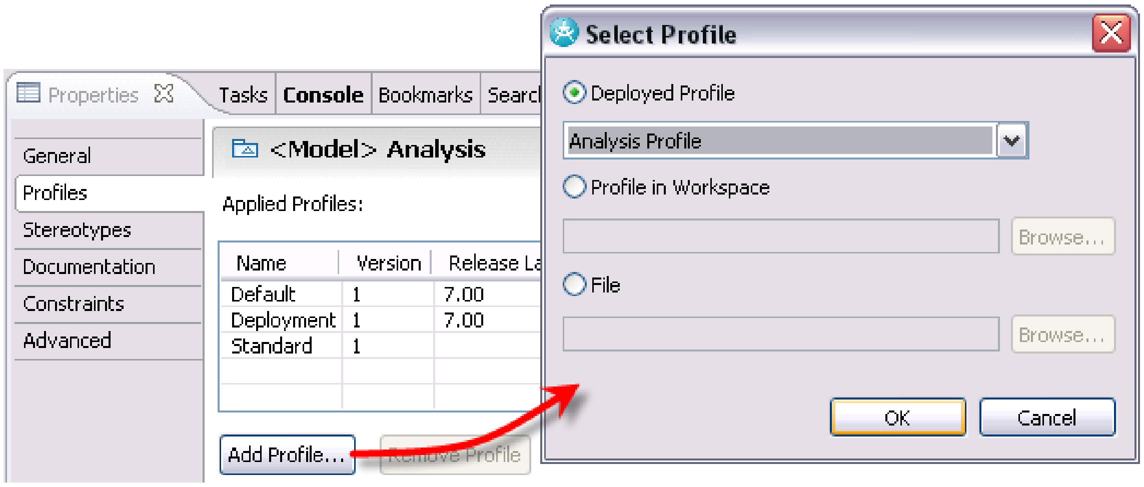
3. In the *Main* diagram, create shortcuts to the newly created diagrams. To do this, you can simply drag and drop each diagram (make sure to read the note below before you do) from the *Project Explorer* onto the *Main* diagram.

   Note: In some configurations, clicking on a diagram in the project explorer or attempting to drag-and-drop it automatically brings the diagram to the front if it is already open, thus hiding the intended target (the *Main* diagram). To avoid this situation, close the *Architectural Layers* and *Key Abstractions* diagrams or use "split screens" as in the snapshot below.


Main
Architectural Layers
Double-click to see the architectural layers
Key Abstractions
Double-click to see the key abstractions
Architectural Layers
Key Abstractions

Important: Every package in a model (including the model itself) should have a "default diagram". This default diagram is the entry point into the package, i.e. the diagram that is opened when double-clicking on the package. The *Main* diagram should contain all relevant information for the user to easily find his/her way in the package (shortcuts to other diagrams, main nested packages, textual information and notes, etc.).

4. Add the profiles *Analysis Profile* and *ArchitectAssistant* to the *Analysis* model. First select the model in the project explorer, then the *Profiles* tab in the *Properties* view. You will need to repeat the *Add Profile* for each profile.

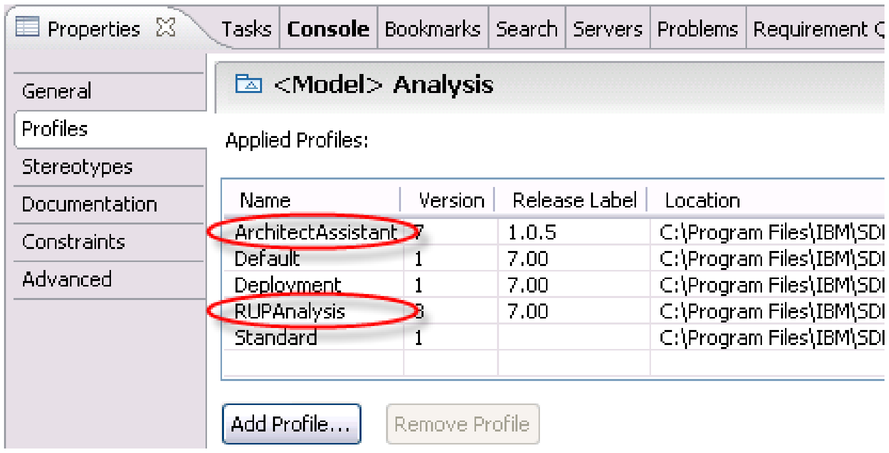


Applied Profiles:

Name	Version	Release La
Default	1	7.00
Deployment	1	7.00
Standard	1	

[✓ Deployed Profile] Analysis Profile

The profiles should appear in the *Applied Profiles* list. We will use them later on.



Applied Profiles:

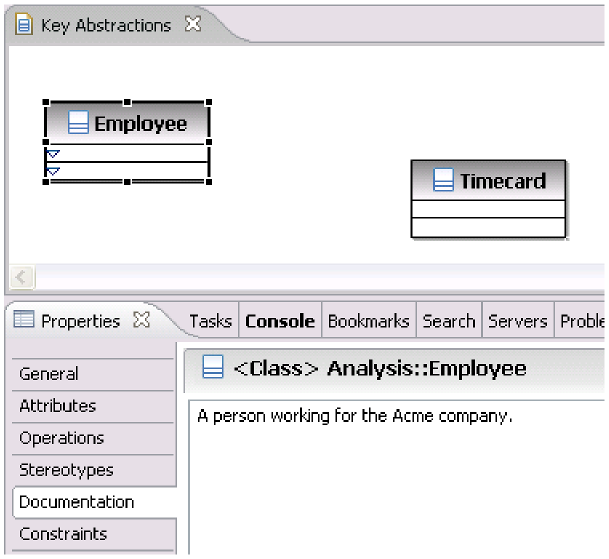
Name	Version	Release Label	Location
ArchitectAssistant	7	1.0.5	C:\Program Files\IBM\SDI
Default	1	7.00	C:\Program Files\IBM\SDI
Deployment	1	7.00	C:\Program Files\IBM\SDI
RUPAnalysis	3	7.00	C:\Program Files\IBM\SDI
Standard	1		C:\Program Files\IBM\SDI

Task 4.2: Identify the Key Abstractions of the Payroll System

1. For this task, you need:
   a. The problem statement and glossary for the Payroll System in Appendix 3 You should already be familiar with these documents as you used them in lab 3.
   b. The Use-Case Model created in lab 3: you can either use the model you created or the solution provided in model *03. Use Case Model*, which you can now open.

2. Open the *Key Abstractions* diagram in the *Analysis* model.

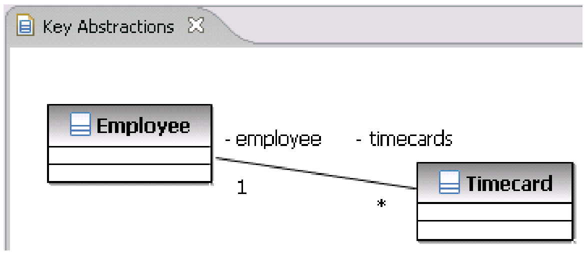
3. Identify the key abstractions of the system and represent them as classes in the *Key Abstractions* diagram:
   a. Remember: A key abstraction is a concept, an entity that the system must be able to handle. The key abstractions form an initial set of classes that is useful to “jump-start” the analysis work.
   b. As an example, consider the following extracted from the problem statement: “*Some employees (…) submit timecards that record the date and number of hours worked for a particular charge number.*”
      i. Employees and timecards are major entities that the system will have to handle. They are key abstractions. We therefore want to create two classes to represent them: *Employee* and *Timecard*. This is also the right time to provide a brief description of each class.



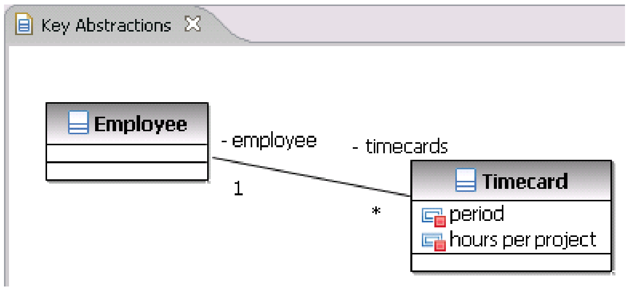
<Class> Analysis::Employee

A person working for the Acme company.

ii. Because each employee "owns" the timecards that he/she submits every week, we will add an association between the 2 classes. An employee may have 0 to n timecards. A timecard only makes sense if it can be associated with exactly 1 employee.



iii. Because it is also said that the timecards record the date and number of hours worked for a particular charge number, we can add *period* (*date* is not enough) and *hours per project* as attributes of *Timecard*.



Should *hours per project* be more detailed? First, keep in mind that the purpose of key abstractions is <u>not</u> to create classes that will survive throughout design (although most will). It may make sense to provide a more detailed representation (introducing additional classes), but it is only just that: another representation of the same information… And there are other factors: knowledge of the business domain, whether this is a new application or an overhaul of an existing one, etc.

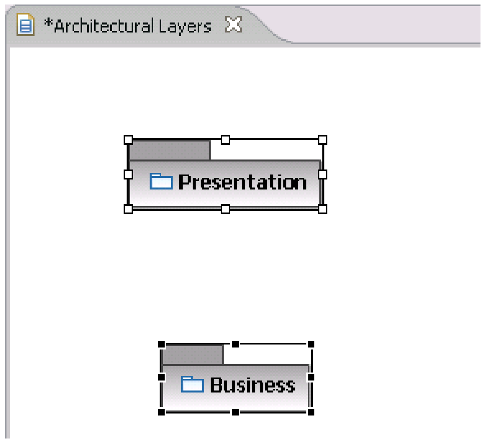
Note: When a class is created in a diagram, its parent is the package containing the diagram. For now, we will not worry about the exact location of the classes. If it helps you organize the information, feel free to allocate those classes to specific packages, but be aware we may have to change this allocation.

Task 4.3: Represent the Higher Layers of the Architecture

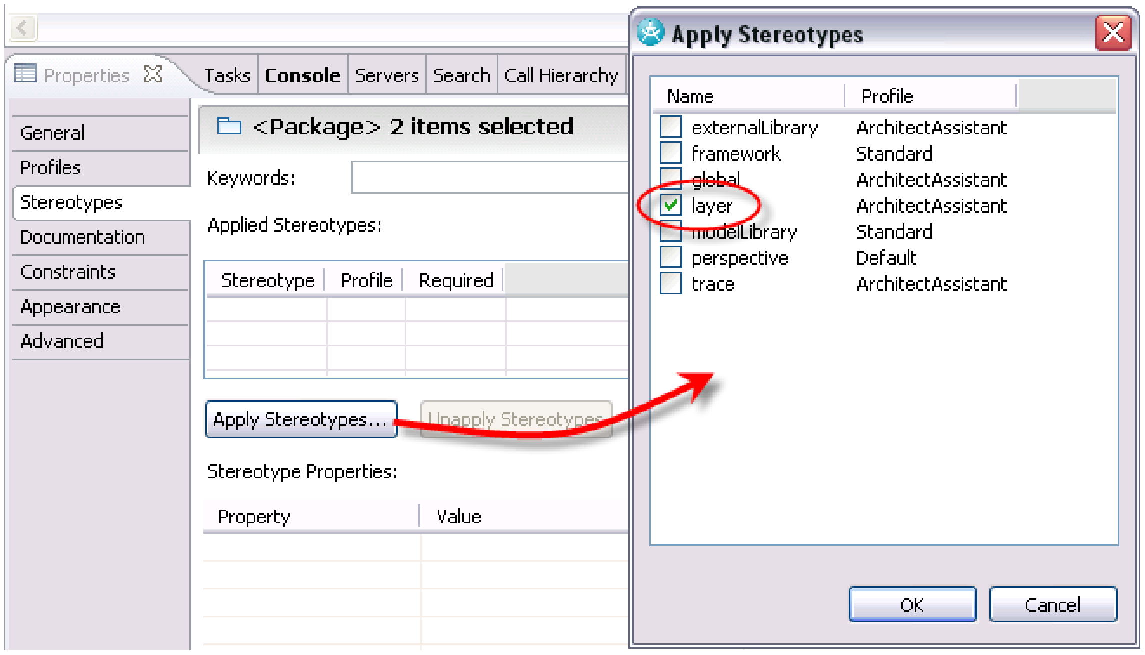
1. The architect has indicated that, at this stage of the analysis, two architectural layers must be created: the *Presentation* layer and the *Business* layer. The *Presentation* layer depends on the *Business* layer.

2. Open the *Architectural Layers* diagram in the *Analysis* model.

3. While in the *Architectural Layers* diagram, create two packages, *Presentation* and *Business.*

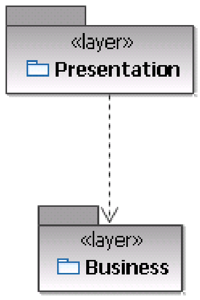


4. Assign the stereotype <<*layer*>> to these packages: you can select both packages (as shown in the picture above), then apply the stereotype from the *Properties* window as shown below:



Name	Profile
☐ externalLibrary	ArchitectAssistant
☐ framework	Standard
☐ global	ArchitectAssistant
✓ layer	ArchitectAssistant
☐ modelLibrary	Standard
☐ perspective	Default
☐ trace	ArchitectAssistant

5. Draw the necessary relationship to support the statement "the *Presentation* layer depends on the *Business* layer". Your diagram should now look something like:



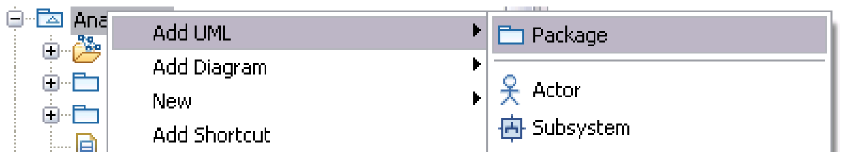
Note: Our two layers are empty for now. In later labs, we will assign our classes and other packages to these layers. But for now, we are only structuring our model for future use. As previously mentioned, if it helps you organize the information, feel free to allocate your classes and packages to the architectural layers.

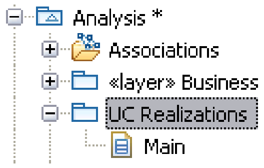
Lab 5 – Use Case Analysis

Task 5.1: Create Use-Case Realizations

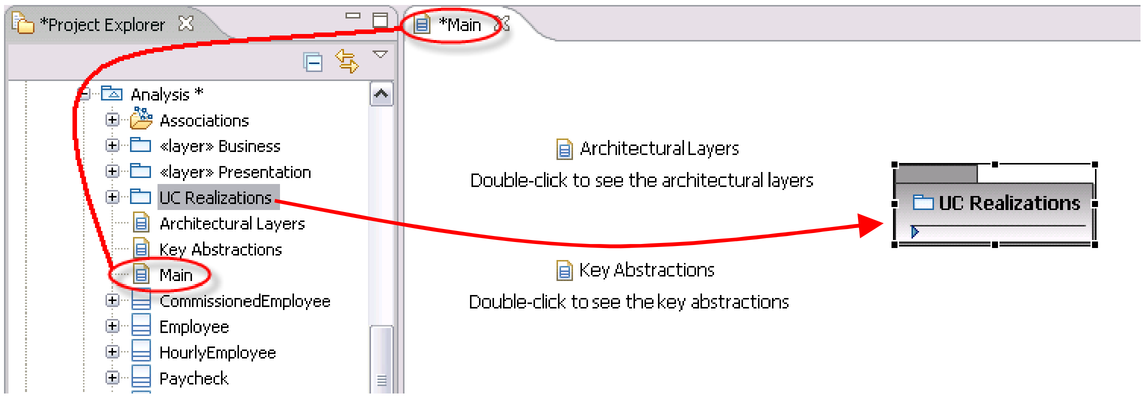
1. You have been assigned the use case *Maintain Timecard* to analyze. Your first task is to prepare the model for the analysis work. For this task, you may use the Analysis model you created in the previous lab or the model *04. Analysis Model*, which you can now open.

2. Working in the *Project Explorer* (see snapshot below), create the package *UC Realizations*.

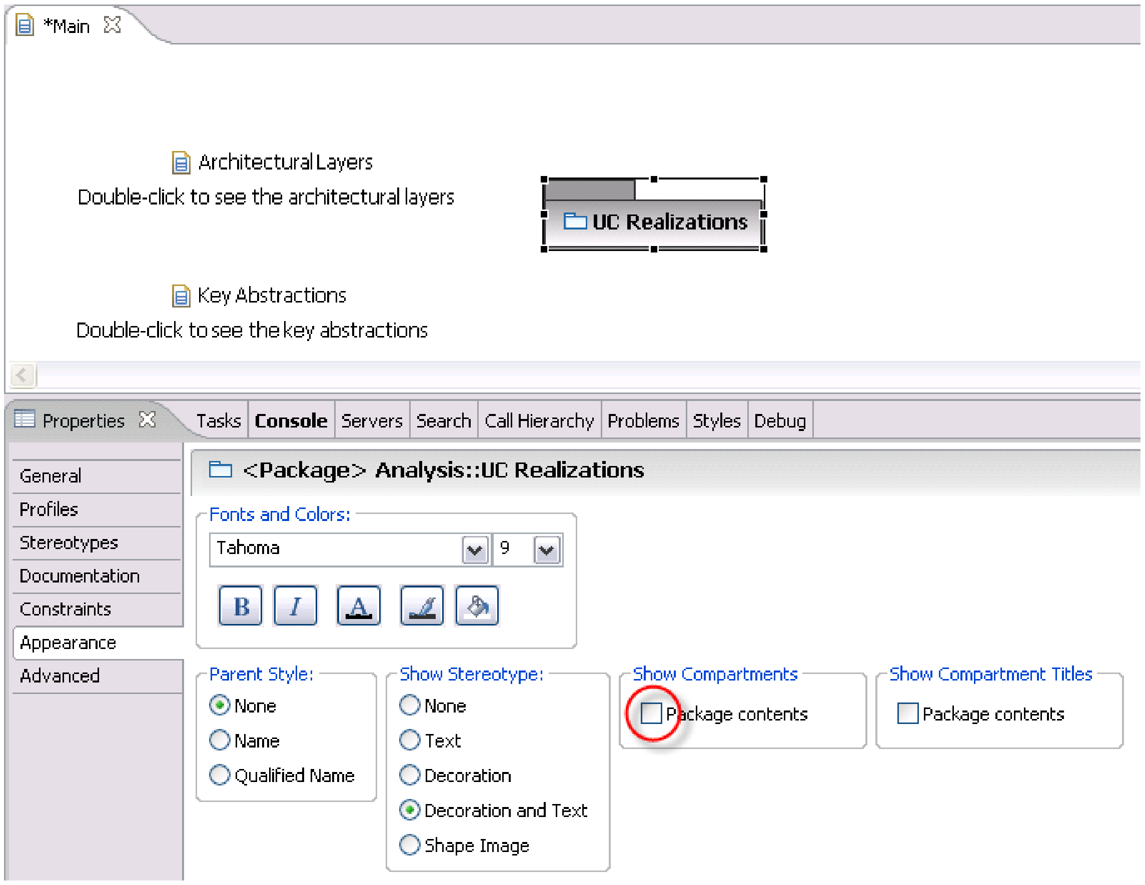




3. Drag the newly created package to the *Main* diagram of the analysis model:



4. You can use the *Appearance* tab in the *Properties* window to modify the appearance of elements like packages and classes in diagrams. For instance, you can hide the *Package Contents* compartment of the *UC Realizations* (the bottom part of the package in the picture above) by un-checking the *Package contents* checkbox as shown below.


*Main

Architectural Layers
Double-click to see the architectural layers

UC Realizations

Key Abstractions
Double-click to see the key abstractions

Properties | Tasks | Console | Servers | Search | Call Hierarchy | Problems | Styles | Debug

<Package> Analysis::UC Realizations

Fonts and Colors: Tahoma 9

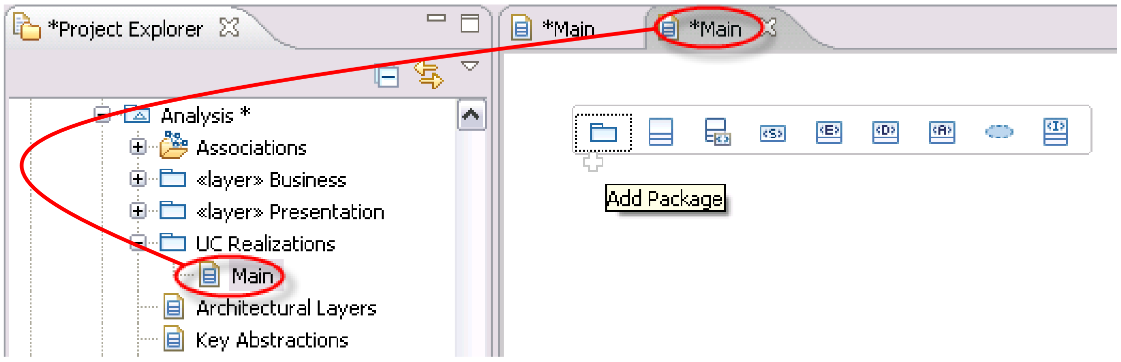
Parent Style: [✓ None] Name Qualified Name

Show Stereotype: None Text Decoration [✓ Decoration and Text] Shape Image

Show Compartments: ☐ Package contents

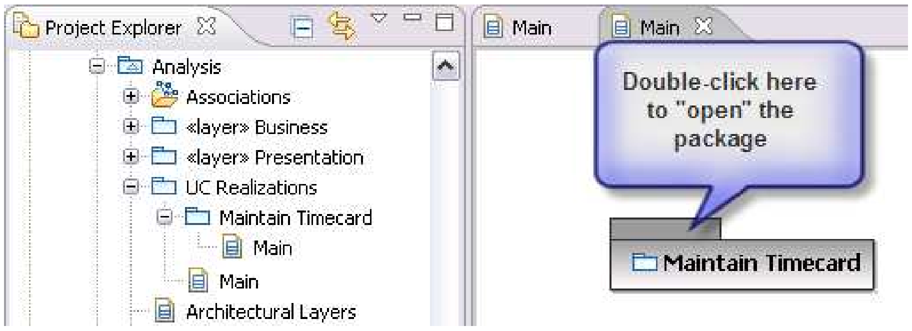
Show Compartment Titles: ☐ Package contents

5. Open the *Main* diagram in the package *UC Realizations* and use the Action bar to create a package called *Maintain Timecard*.

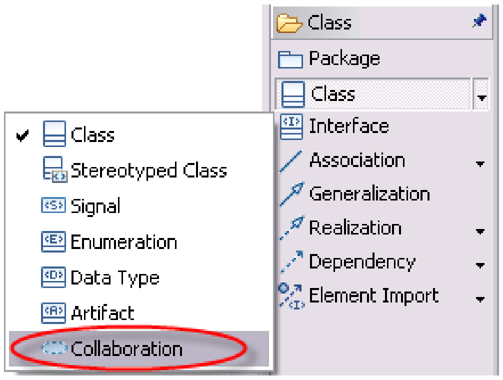


Note: If you don't see the action bar, click again anywhere in the diagram.

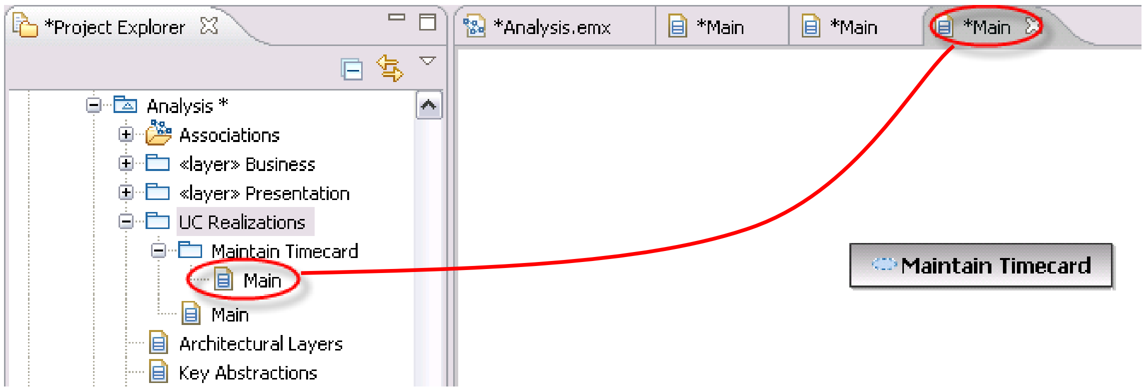
6. Double-click on the *Maintain Timecard* package to open it (make sure to double-click outside the text area as this would turn on text editing instead of opening the package – double-clicking on the upper left tab of the package will ensure you get the desired effect).



7. You are now in the *Main* diagram of the package *Maintain Timecard.* You are going to create a use-case realization named *Maintain Timecard*. Keep in mind: we are still in the process of structuring the model.
   a. Select the *Collaboration* tool in the *Class* drawer under the *Class entry*.

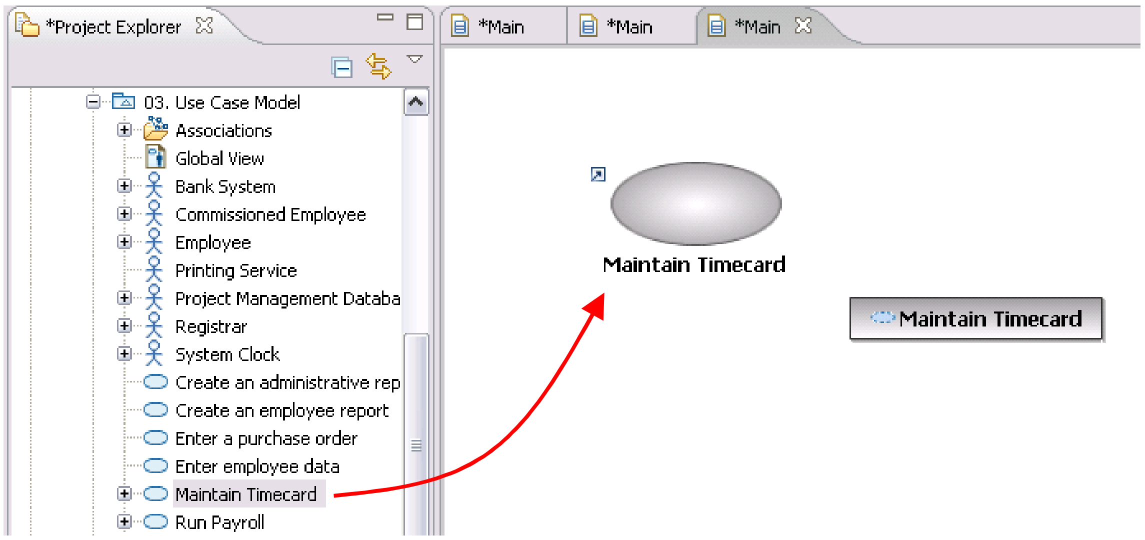


   b. With the *Collaboration* tool selected, click once in the *Main* diagram and name the collaboration *Maintain Timecard*. Optionally, hide the structure compartment as explained in step 4 above.



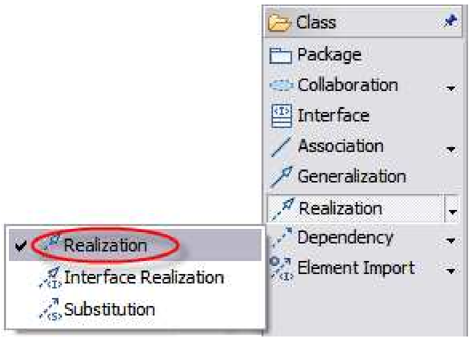
Note: The collaboration is displayed as a rectangular shape. In the version of RSM used for setting up this lab, it is not possible to change it to its icon representation.

c. Retrieve in the *Project Explorer* the use case *Maintain Timecard* created in lab 3 (or use the use case from model *03. Use-Case Model*). Then drag and drop the use case in the diagram next to the collaboration as shown below:

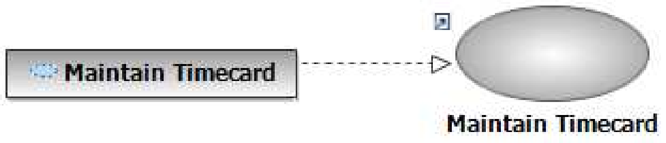


Note: The use case has a little arrow in the upper left corner. This indicates that it is located in a different model as the diagram in which it appears.

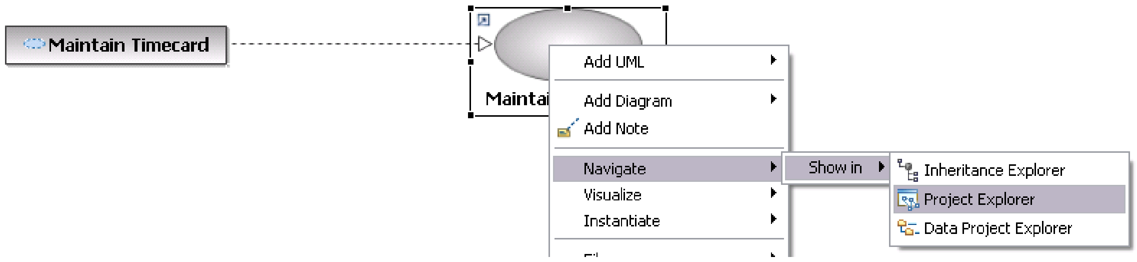
d. Draw a *Realization* relationship from the use-case realization to the use case.



The diagram should look something like:



Note: The use-case description and the use-case realization are best kept separated as they will often be handled by different people and/or at different times. Also having the use case “shortcut” in the diagram makes it very easy to locate the use case in the model where it is defined, and therefore to get access to any associated information, including its textual description (use-case specification), documenting diagrams, etc.

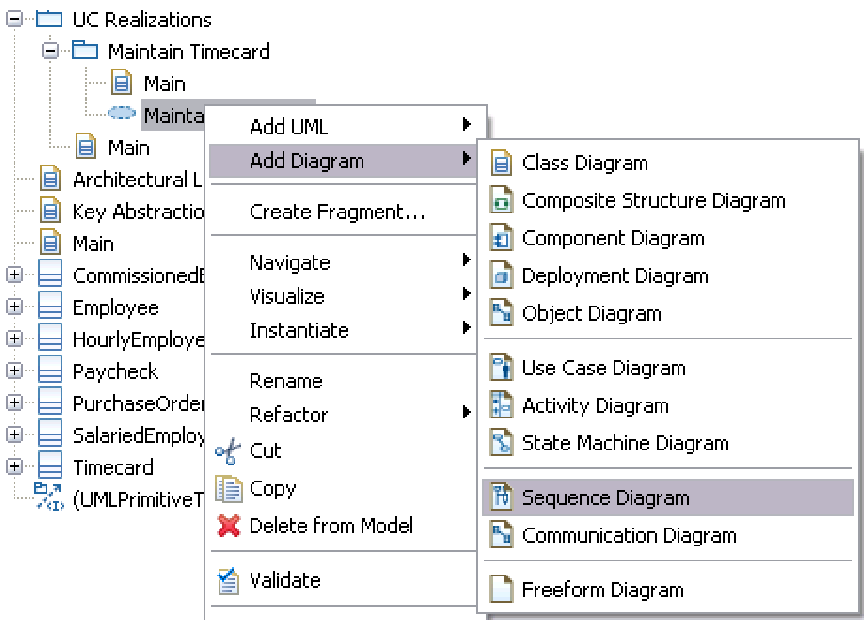


Task 5.2: Assign the Use-Case Behavior to Classes

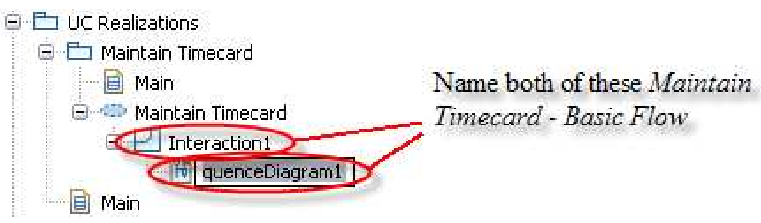
1. For this exercise, you will need the specification for the *Maintain Timecard* UC in Appendix 3 of this document. You can either work from the *Analysis* model created during the previous exercise, or from model *05.1. Analysis*.

2. If this is not already the case, make sure the *Main* diagram for the *Maintain Timecard* UC Realization is displayed on your screen.

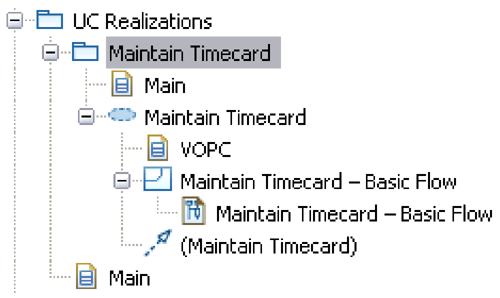
3. Create a new sequence diagram for the basic flow of the use case:
   a. In the *Project Explorer*, right-click the *Maintain Timecard* collaboration () and select *Add Diagram > Sequence Diagram.*



   b. Name the interaction () <u>and</u> the diagram () *Maintain Timecard – Basic Flow*:

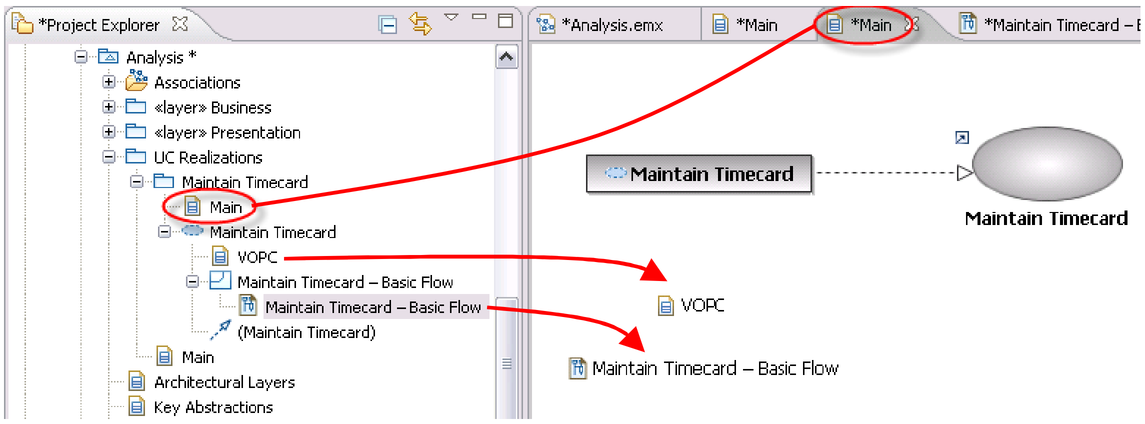


c. Create a class diagram called *VOPC* under the *Maintain Timecard* collaboration ( ).



UC Realizations
Maintain Timecard
Main
Maintain Timecard
VOPC
Maintain Timecard – Basic Flow
Maintain Timecard – Basic Flow
(Maintain Timecard)
Main

d. Drag and drop the *VOPC* and Maintain Timecard – Basic Flow diagrams onto the *Main* diagram of the *Maintain Timecard* UC, which should look like this:



e. Now here is the challenging bit: From the textual description of the basic flow of the use case, identify the classes that participate in the use case and assign use case behavior to these classes as this was previously shown by the instructor: (an example follows)

i. Each object must be assigned to a class (existing or new).

ii. Each message between objects must correspond to a class operation.

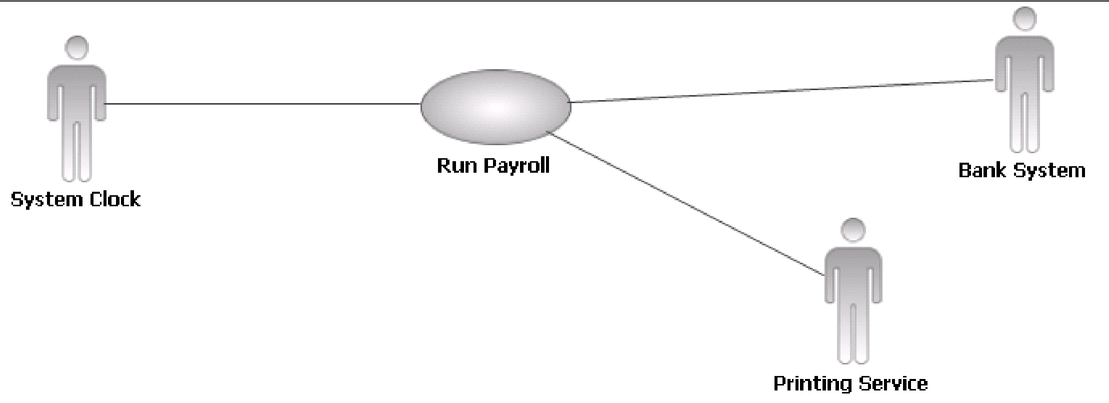
iii. Each participating class will be added to the *VOPC* diagram in the second part of the exercise (keep in mind: for the time being, we are not concerned with the actual location of the classes – this will be dealt with at a later time).

iv. For each class, define its documentation, analysis stereotype (*boundary*, *control* or *entity*), and main attributes (*entity* classes).

Example: Let's take the example of the Run Payroll UC.

What we have:

1. A use-case diagram:



2. The description for the Run Payroll UC:

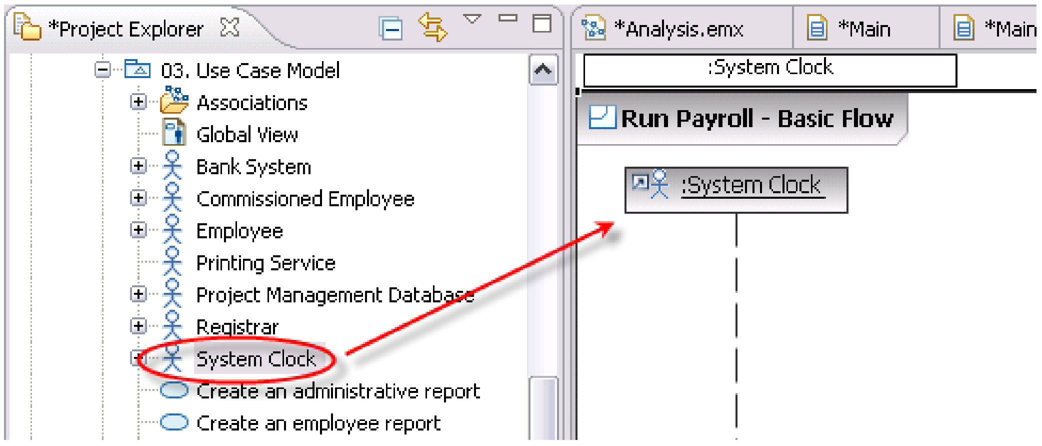
Basic Flow

1. The use case begins when it's time to run the payroll. The payroll is run automatically every Friday and the last working day of the month.
2. The system retrieves all employees who should be paid on the current date.
3. …

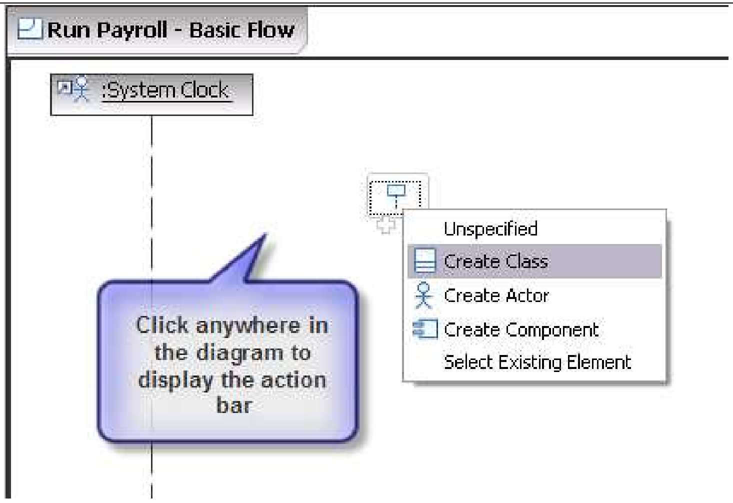
3. A set of key abstractions: *Employee* and its subclasses, *Paycheck*, *Timecard*, *PurchaseOrder*.

Get started with the sequence diagram:

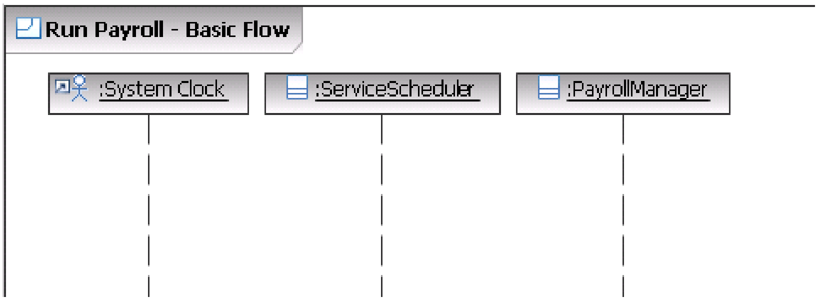
1. The actor initiating the use case is the *System Clock*. We'll instantiate this actor by dragging it from the *Project Explorer* into our sequence diagram:



2. As discussed in the course, we will define one controller object (*:PayrollManager*) for handling the sequencing of actions in the scenarios for this use case. Note that in the process we automatically create the corresponding class (as shown in the snapshot below). We also create boundary objects: one per actor-use case pair, although for our example we'll create only one (*:ServiceScheduler*) and skip the other two (*:BankSystem* and *:PrintingService*).

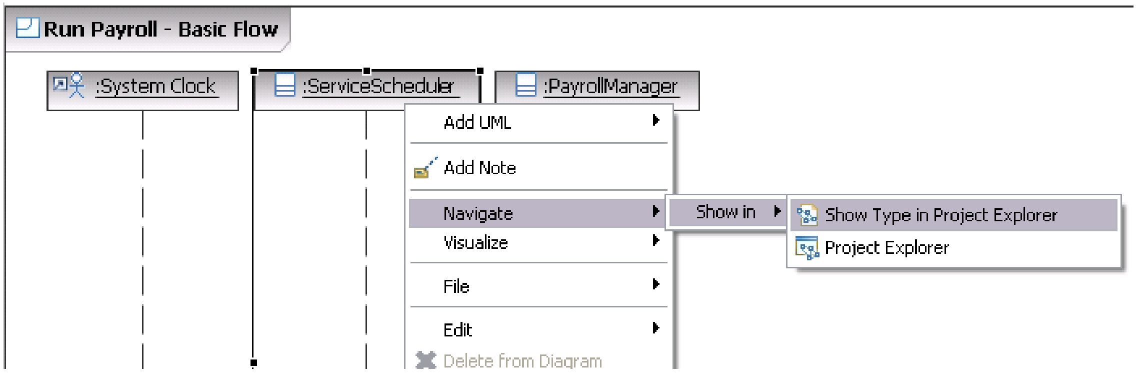


We now have the following diagram:

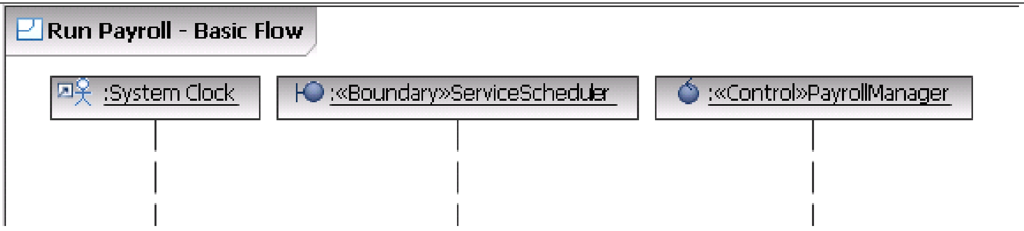


Note: All three objects are unnamed. This is because they are unique, at least within the context of this scenario.

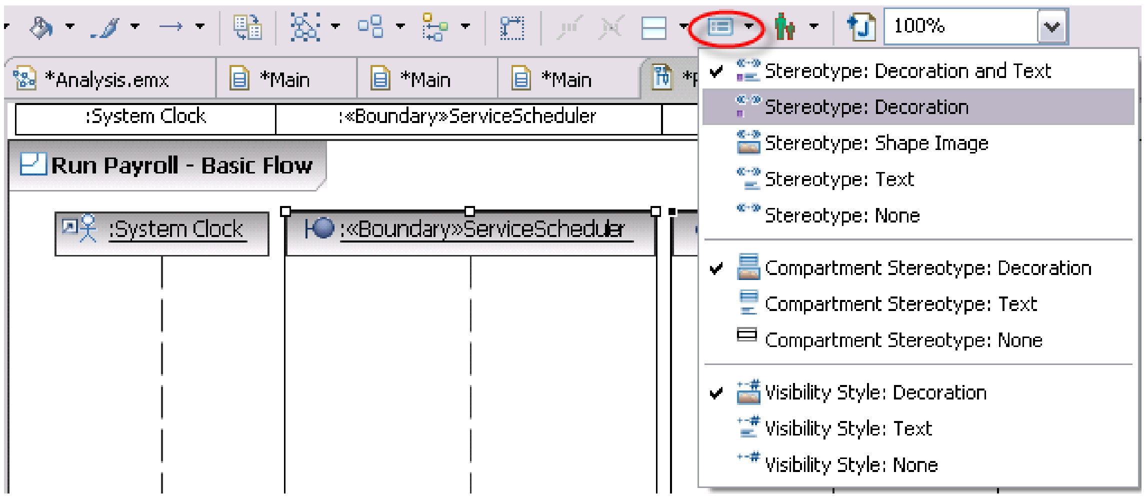
Let’s not forget to stereotype the classes appropriately. Here is a useful tip: from an object in the sequence diagram, you can select in the *Project Explorer* its “type” (the parent class in this case) directly from the right click:

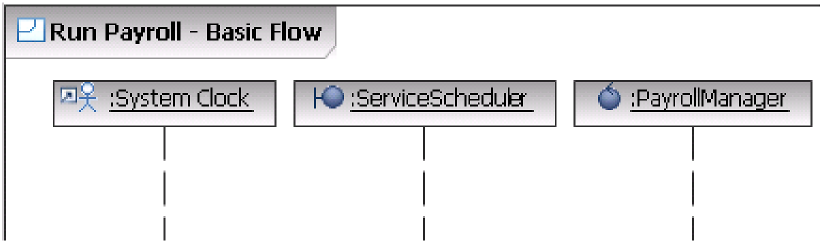


The diagram now looks something like:

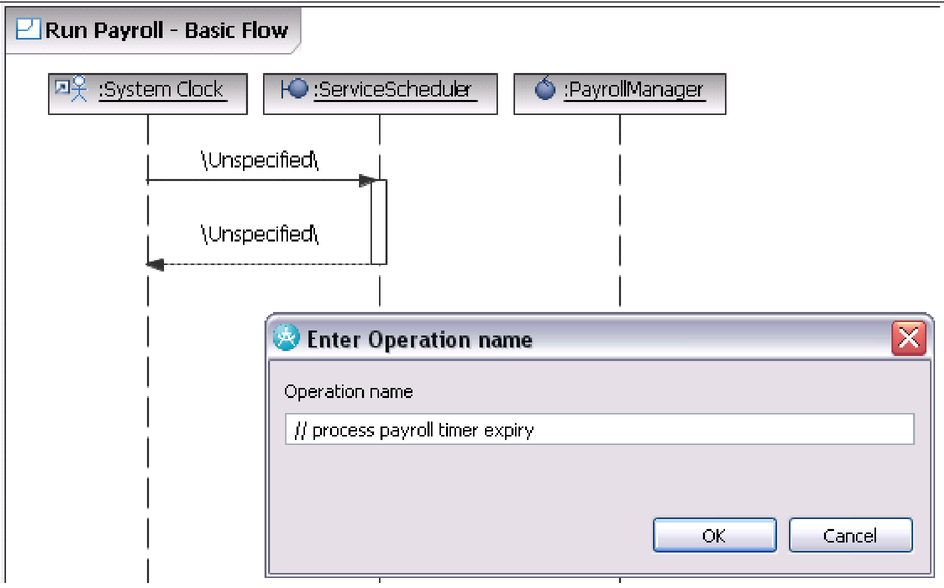


To reduce the width of the objects on the screen, you might also want to display the icon without its text:

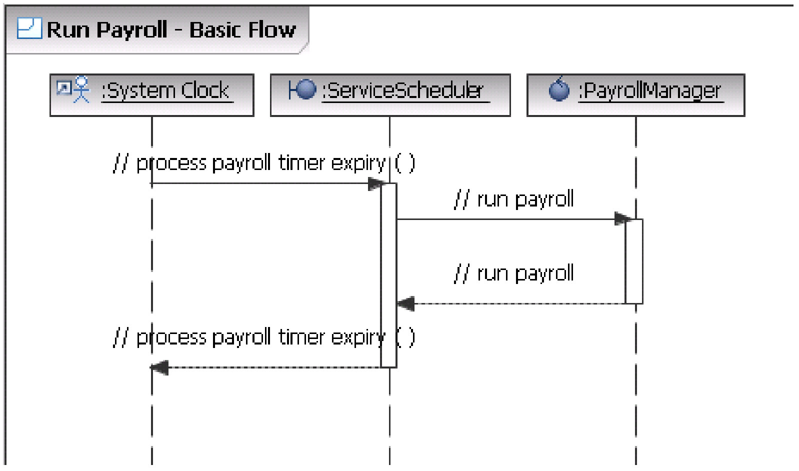

100%
*Analysis.emx
*Main
*Main
*Main
:System Clock
:«Boundary»ServiceScheduler
Run Payroll - Basic Flow
:System Clock
:«Boundary»ServiceScheduler
Stereotype: Decoration and Text
Stereotype: Decoration
Stereotype: Shape Image
Stereotype: Text
Stereotype: None
Compartment Stereotype: Decoration
Compartment Stereotype: Text
Compartment Stereotype: None
Visibility Style: Decoration
Visibility Style: Text
Visibility Style: None



3. Step 1 of our basic flow reads: *The use case begins when it's time to run the payroll. The payroll is run automatically every Friday and the last working day of the month*. This translates in our actor-object *:System Clock* sending a notification to *:ServiceScheduler*. What really happens is that some timer previously set expires, and that we had registered our service scheduler to be notified upon expiry. It really doesn't matter at this stage how this is done exactly. We are only interested here in the result.



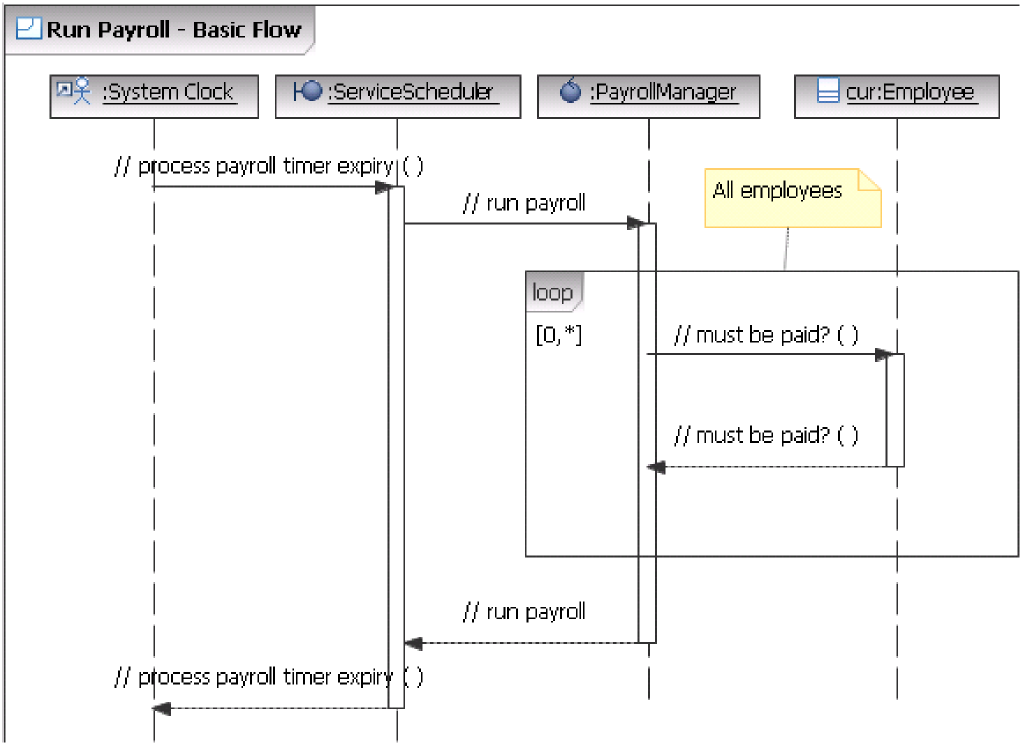
4. On receipt, our scheduler object knows this is the payroll timer (again we are interested in the result, not in how this is determined) and that we need to perform the run payroll processing. The scheduler object being a boundary object, it will simply pass the information to the *PayrollManager* by invoking, say, its *run payroll* operation. Our sequence diagram now looks like this:



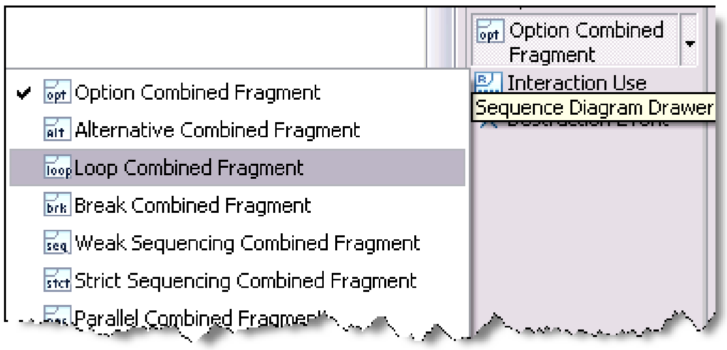
5. Step 2 says that *The system retrieves all employees who should be paid on the current date*. At this stage, we'll make the assumption that *PayrollManager* knows about all the employees. This seems reasonable enough. After all, this is the payroll manager! In practice this will translate at the class level into an association *0..n* from *PayrollManager* to *Employee* with a role name like *allEmployees* or *staff*.

   However we are interested only in *employees who should be paid on the current date*. Because an employee knows its classification (hourly, salaried, commissioned) and the associated data (e.g. hourly rate and number of hours worked for an hourly employee), and when the last payment was made (the *Employee* object has a list of *Paycheck* objects with the date and amount), it seems natural to say that the employee should be able to answer the question *Should you be paid at this date?*

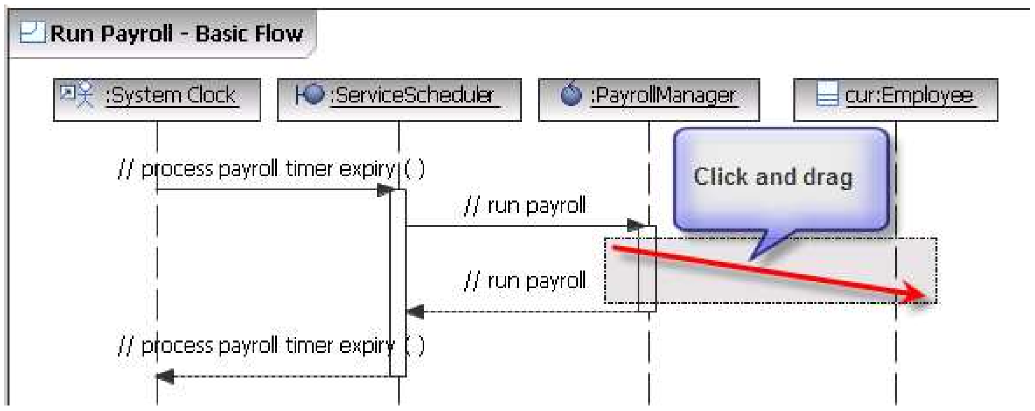
In our sequence diagram, we will materialize our thoughts by adding a loop on all employees and asking each one if he/she has to be paid.



Notes:
1) What is this *cur:Employee* object? As we iterate through the list of all existing employees, we use *cur:Employee* to reference the employee object being currently examined. If they are 10 employees, *cur* will successively point to each one of the 10 employees.
2) To draw the loop fragment, first select it in the palette:



Then drag across the lifelines of the objects to cover. (Objects can also be added or removed after the fact.)

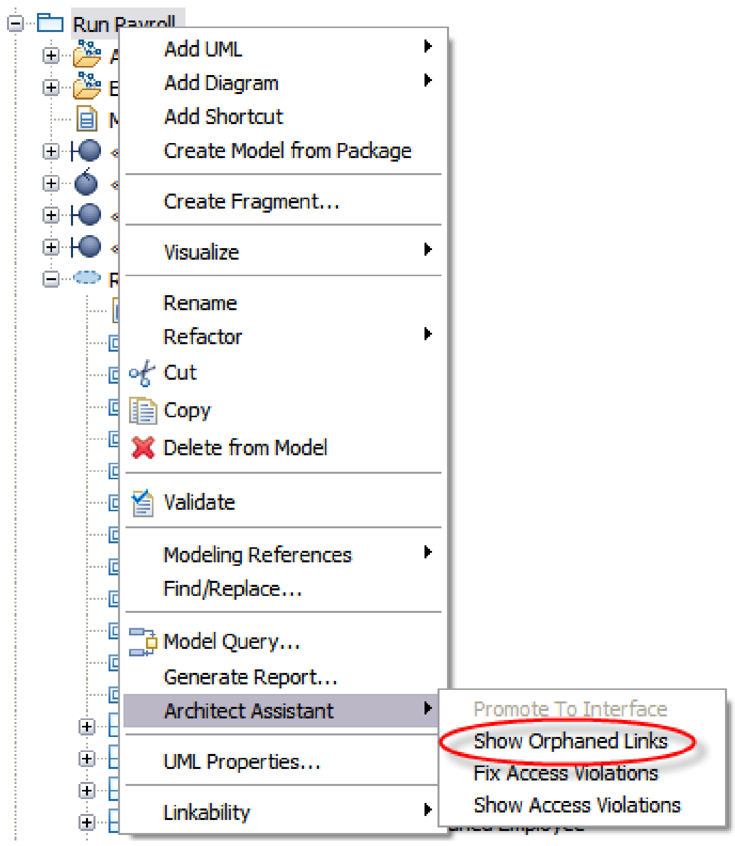


Task 5.3: Complete the VOPC

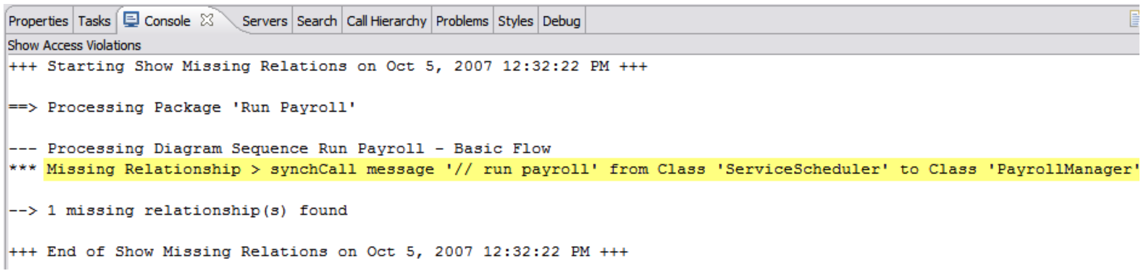
1. For this exercise, you will need the *Maintain Timecard – Basic Flow* sequence diagram created in task 5.2. You can either work from the *Analysis* model created during the previous exercise, or from model *05.2. Analysis*.

2. Open the *VOPC* diagram for the *Maintain Timecard* UC Realization.

3. Complete the diagram as follows: (an example follows)
   a. Add stereotypes to all classes (if not already done).
   b. Create relationships between classes: remember, every link between object is an instance of a relationship between the corresponding classes. To that effect, you can use the *ArchitectAssistant* plug-in you installed earlier: right-click on the package containing your interaction diagrams and select the command *Architect Assistant > Show Orphaned Links*:



The results are logged in the console. In the sample below, a missing relationship has been detected between the *ServiceScheduler* class and the *PayrollManager* class. This is because an object of type *ServiceScheduler* is invoking the operation *// run payroll* on an object of type *PayrollManager*. (You can actually see these objects on the sequence diagram in the previous page.)



```
+++ Starting Show Missing Relations on Oct 5, 2007 12:32:22 PM +++

==> Processing Package 'Run Payroll'

--- Processing Diagram Sequence Run Payroll - Basic Flow
*** Missing Relationship > synchCall message '// run payroll' from Class 'ServiceScheduler' to Class 'PayrollManager'

--> 1 missing relationship(s) found

+++ End of Show Missing Relations on Oct 5, 2007 12:32:22 PM +++
```

We fix this case by adding a bi-directional association between the two classes. This is discussed in more detail in the example below.

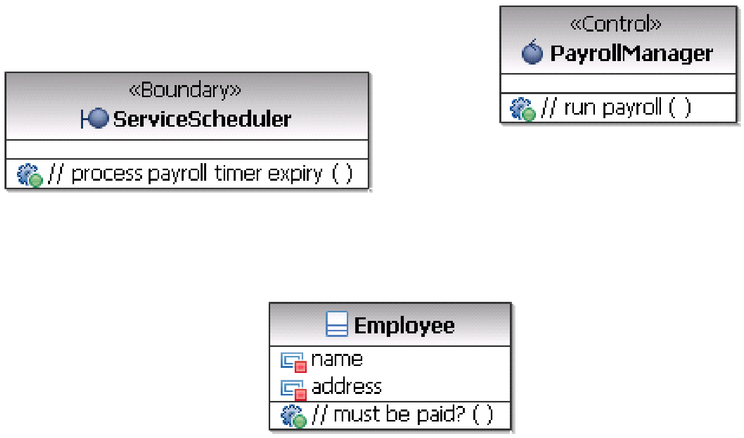
Run the *Show Orphaned Links* command until you have no missing relations left. To ensure you have not missed any, you can run the *Show Orphaned Links* command against the whole model.

c. Specify multiplicity and role names as appropriate.
d. Add class responsibilities (if not already done).
e. Add relevant attributes to entity classes (if not already done).

<u>Example</u>:

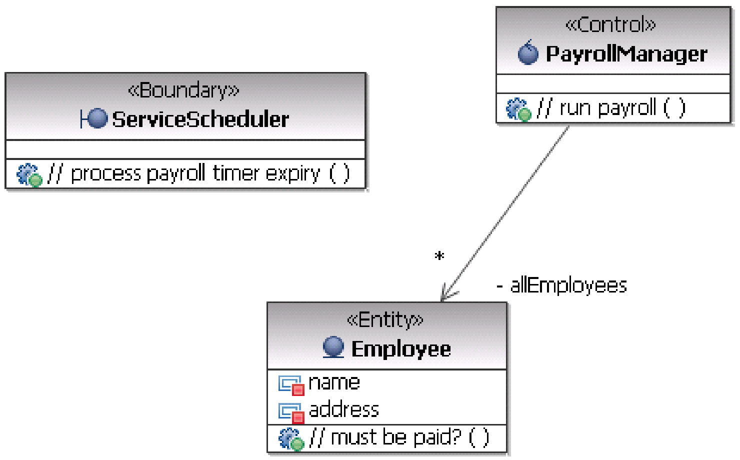
1. In the previous example, we had defined 4 objects: *:SystemClock*, *:ServiceScheduler*, *:PayrollManager* and *cur:Employee*. The object *:SystemClock* corresponds to an actor and is outside our system. The last three are classes in our model. Two of the classes were created as a result of our work, class *Employee* had been identified in task 4.2 (Identify Key Abstractions).

2. Let's drag the three classes to our VOPC:



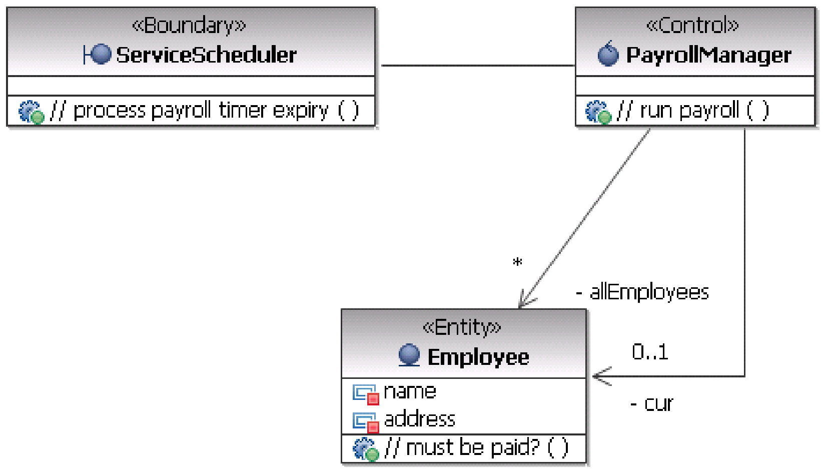
3. We note that we forgot to make *Employee* an <<Entity>> class.

4. If you remember the example, we had made the assumption that *PayrollManager* knows about all the employees and we had stated that this would translate into an association *0..n* from *PayrollManager* to *Employee*. Our VOPC now looks like:



5. Because *:ServiceScheduler* has a link to *:PayrollManager*, we need to add a relationship between *ServiceScheduler* and *PayrollManager*. For now, we will not specify the exact nature and details of this relationship.

   Because *:PayrollManager* talks to *cur:Employee*, we will represent this relationship as an association from *PayrollManager* to *Employee* with a role name of *cur* and a multiplicity of *0..1*. Since we don't want entities to depend on control (or boundary) objects, we will use a one-way association – as we did for the *allEmployees* association.

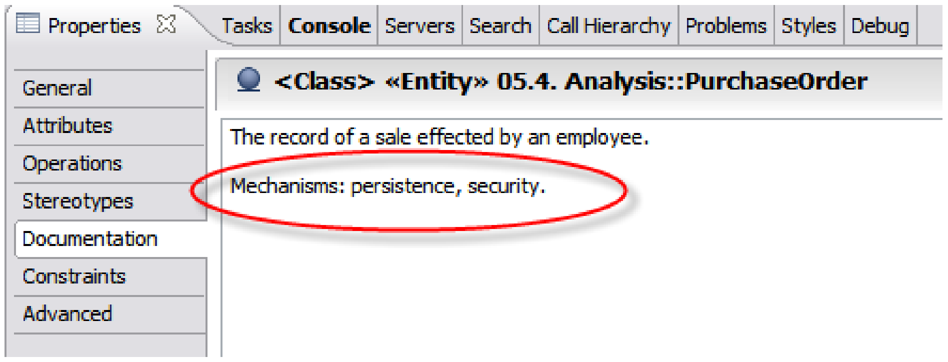


Task 5.4: Map Analysis Mechanisms to Classes

The architect has indicated that the following analysis mechanisms have been identified:

1. Persistence
2. Distribution
3. Security
4. Legacy interface

Inspect all the classes in your model and determine the mechanisms to apply to each one. You will do this by completing the documentation field of the corresponding classes as shown below.



<Class> «Entity» 05.4. Analysis::PurchaseOrder

The record of a sale effected by an employee.

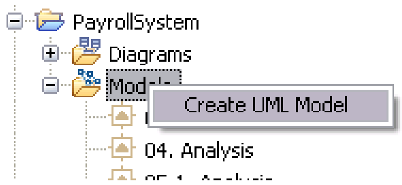
Mechanisms: persistence, security.

Lab 6 – Identify Design Elements

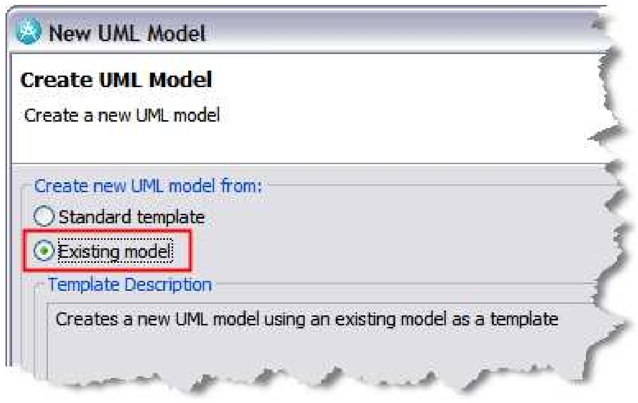
Task 6.1: Transform an Analysis Class into a Subsystem

1. For this exercise, you will work from the model *05.4. Analysis*. Your task is to transform the *ProjectManagementDB* boundary class into a subsystem and its corresponding interface.

2. Because we are now moving into design, let's create a new design model based on the analysis model:
    a. Make sure the model *05.4. Analysis* is closed.
    b. Create a new UML model in your project:



    c. In the *New UML Model* dialog, select *Existing Model*, then click *Next*.



Create UML Model

Create a new UML model

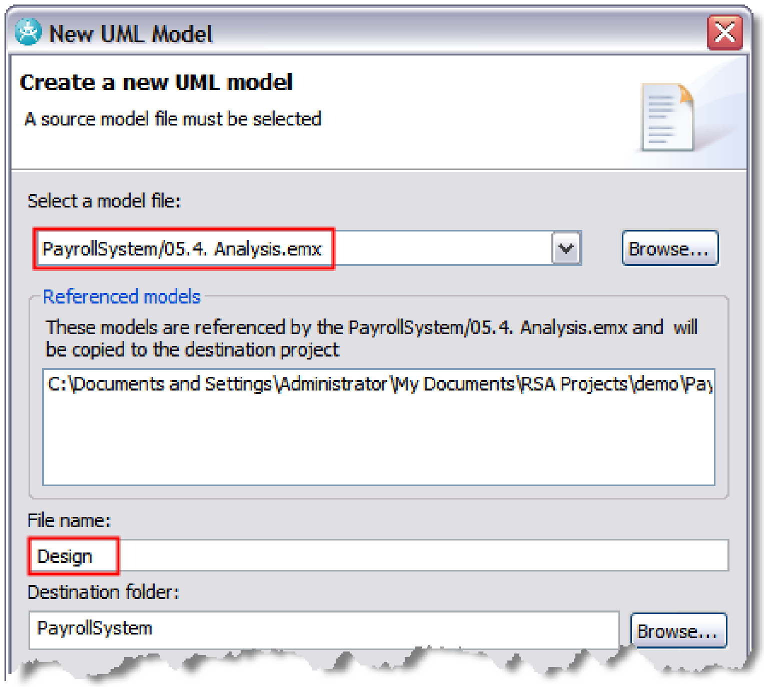
Create new UML model from:

Standard template [✓ Existing model]

Template Description

Creates a new UML model using an existing model as a template

    d. Select *05.4. Analysis* as the model file and enter *Design* as the *File name*, then click *Finish*.


New UML Model

Create a new UML model

A source model file must be selected

Select a model file: PayrollSystem/05.4. Analysis.emx

Referenced models

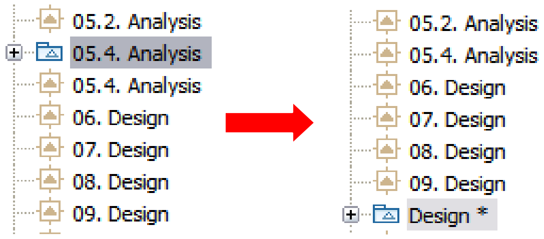
These models are referenced by the PayrollSystem/05.4. Analysis.emx and will be copied to the destination project

C:\Documents and Settings\Administrator\My Documents\RSA Projects\demo\Pay

File name: Design

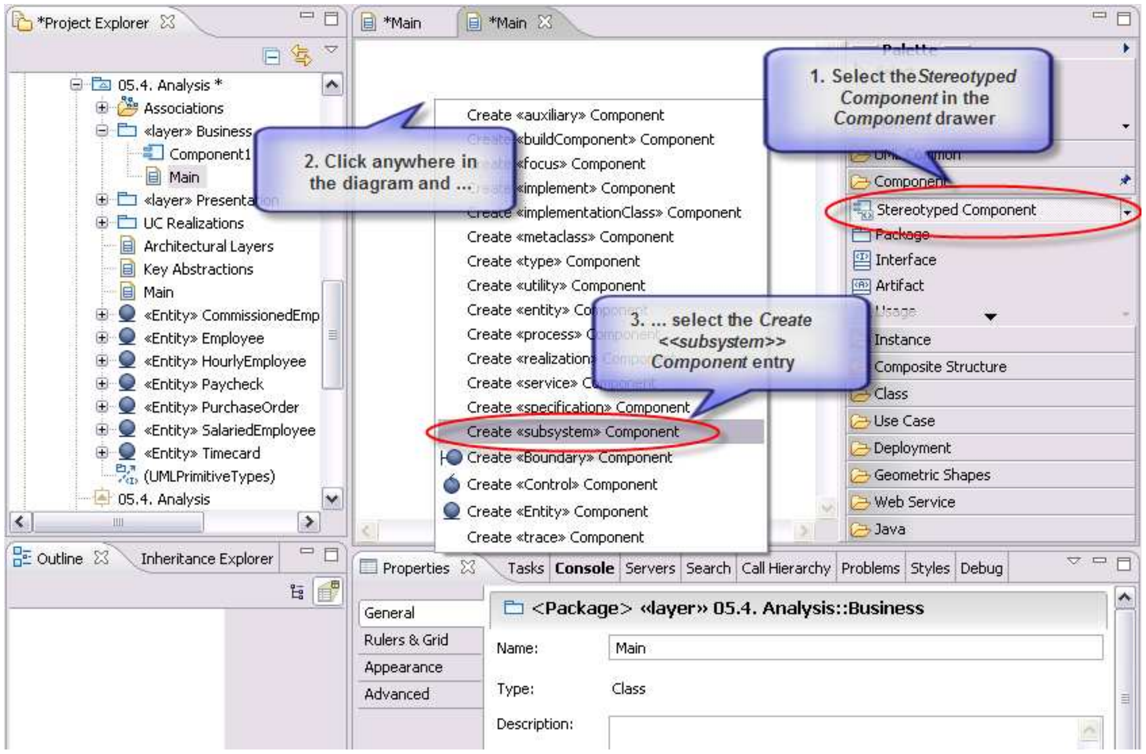
Destination folder: PayrollSystem

e. Although you named the <u>file</u> *Design*, the <u>model</u> is still named *05.4. Analysis* (as shown below on the left). Select the model and, using the *Properties* view, rename it *Design*. Save.



<u>Note</u>: Some of the snapshots below were created in earlier versions of this document and still reference *05.4. Analysis* instead of *Design*.

3. Create the subsystem *ProjectManagementDB* in the *Business* package:


1. Select the *Stereotyped Component* in the *Component* drawer

2. Click anywhere in the diagram and ...

3. ... select the *Create <<subsystem>> Component* entry

<Package> «layer» 05.4. Analysis::Business

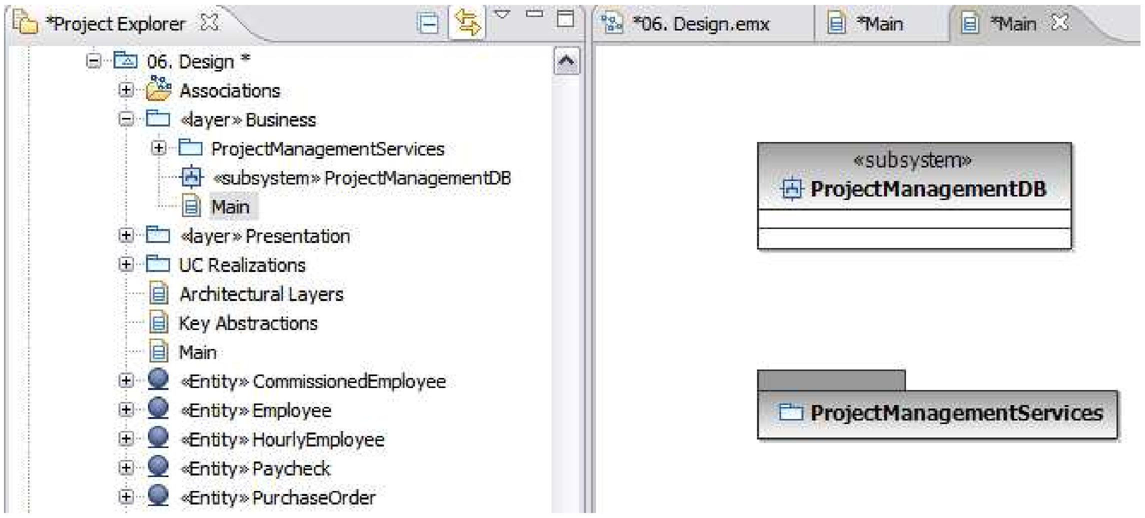
Name: Main

Type: Class

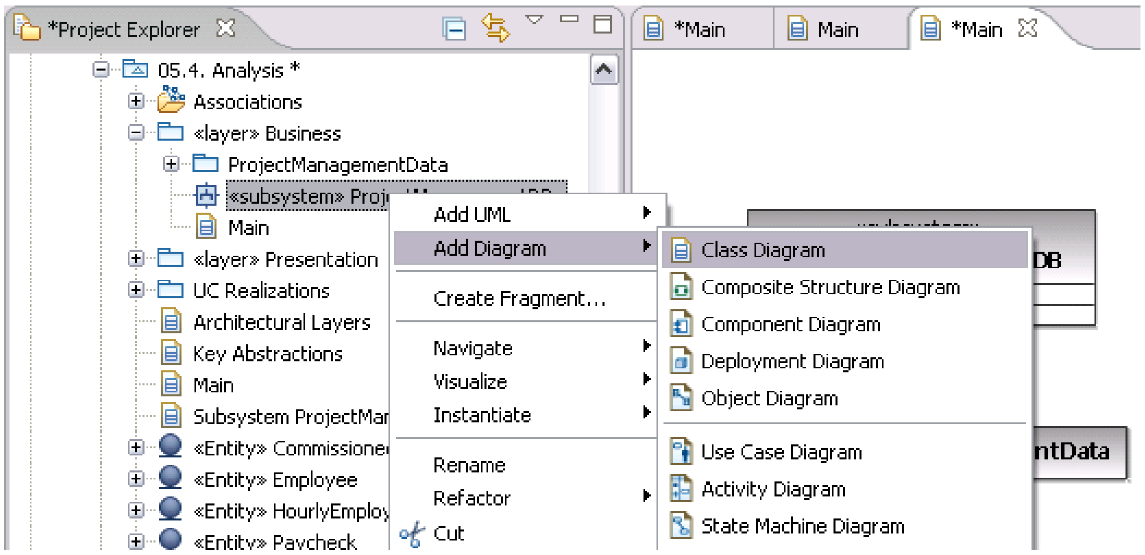
Description:

Name the component *ProjectManagementDB* (be careful not to delete the stereotype while renaming it if you are working from the *Project Explorer*).

4. Create a package *ProjectManagementServices* in the *Business* package. This new package will contain the interface corresponding to the subsystem as well as associated data. Your model should now look like this:

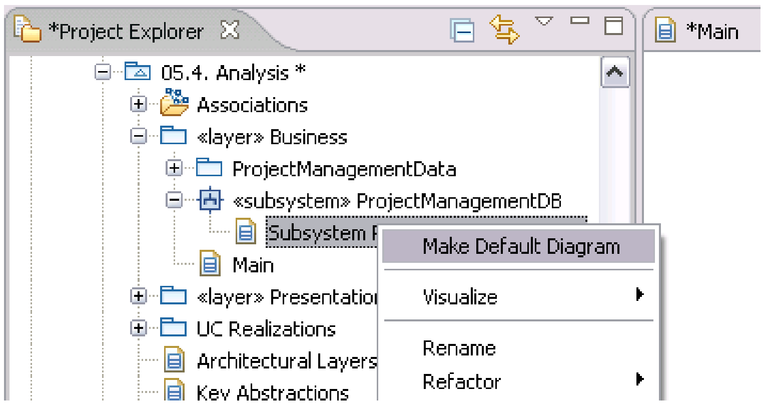


5. In the *ProjectManagementDB* subsystem, create a class diagram called *Subsystem ProjectManagementDB*:
   a. Right-click the subsystem in the *Project Explorer* and select *Add Diagram > Class Diagram*.



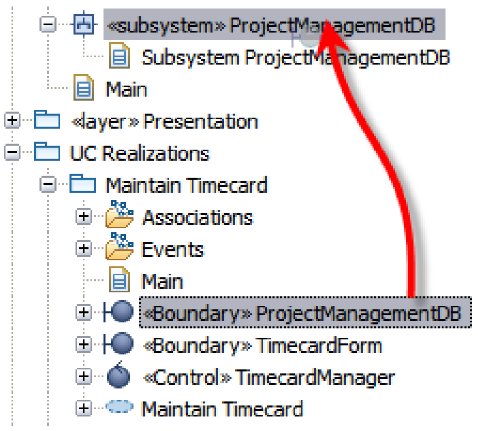
b. Name the diagram *Subsystem ProjectManagementDB.*

c. Right-click the newly created diagram in the *Project Explorer* and select *Make Default Diagram.*

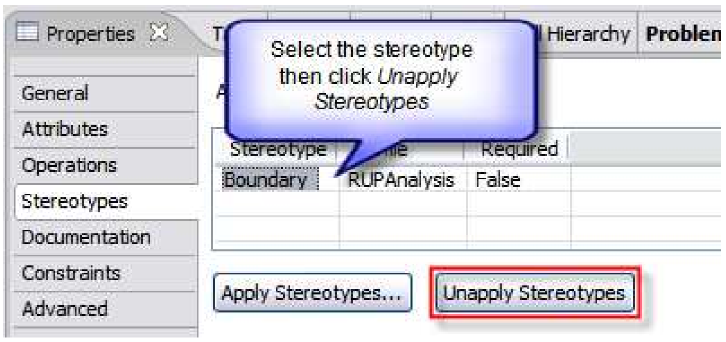


6. Move the *ProjectManagementDB* class into the *ProjectManagementDB* subsystem:

   a. Locate the <<*boundary*>> class *ProjectManagementDB.*
   b. Move the class to the *ProjectManagementDB* subsystem: From the *Project Explorer*, drag the class into the subsystem as shown below.



c. Remove the stereotype <<*boundary*>>:



Stereotype	Profile	Required
Boundary	RUPAnalysis	False

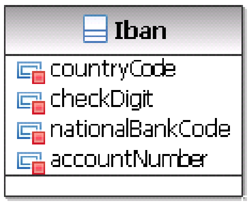
d. Add the class to the *Subsystem ProjectManagementDB* diagram.
e. Transform the responsibility *// retrieve charge numbers* into a full operation. Name the operation *retrieveChargeNumbers()*. Add the appropriate parameters and return value. An example of transforming a responsibility into an operation is provided below.

Example:

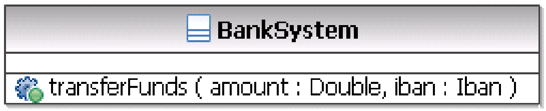
During the use-case analysis of the Run Payroll, we identified the responsibility *// transfer funds with paycheck + bank infos* on class *BankSystem*.

We first decide to name the operation *transferFunds()*.

In terms of parameters, we might be tempted to use *paycheck* and *bank infos* as parameters. We do have a class <<*Entity*>> *PayCheck*. However, because we also want to reuse *BankSystem* outside the scope of our payroll system, we decide to define the first parameter as the amount to transfer and to assign it the type *Double*, a primitive type defined for the project. For the second parameter, we decide to use the *International Bank Account Number (IBAN)*, an internationally standardized and uniform representation for the bank and account number, elaborated by the European Committee for Banking Standards (ECBS). We therefore create an *Iban* class:



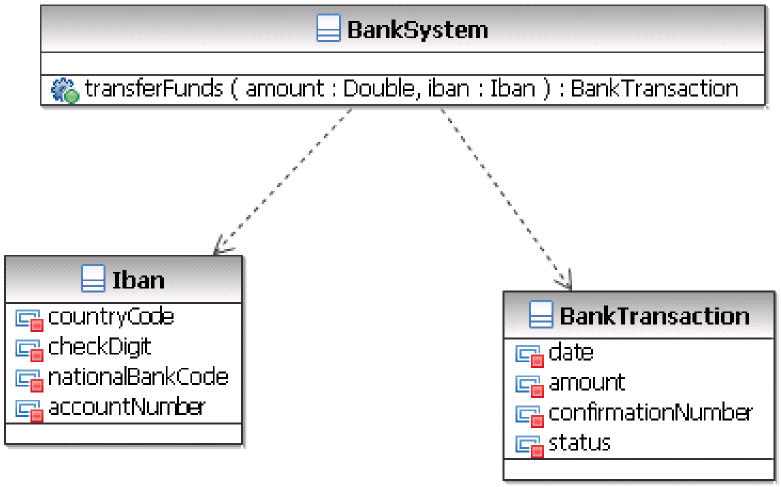
Our *BankSystem* class now looks like this:



For the return value, we could use a simple status, but we did note that the basic flow for the Run Payroll UC includes the following step:

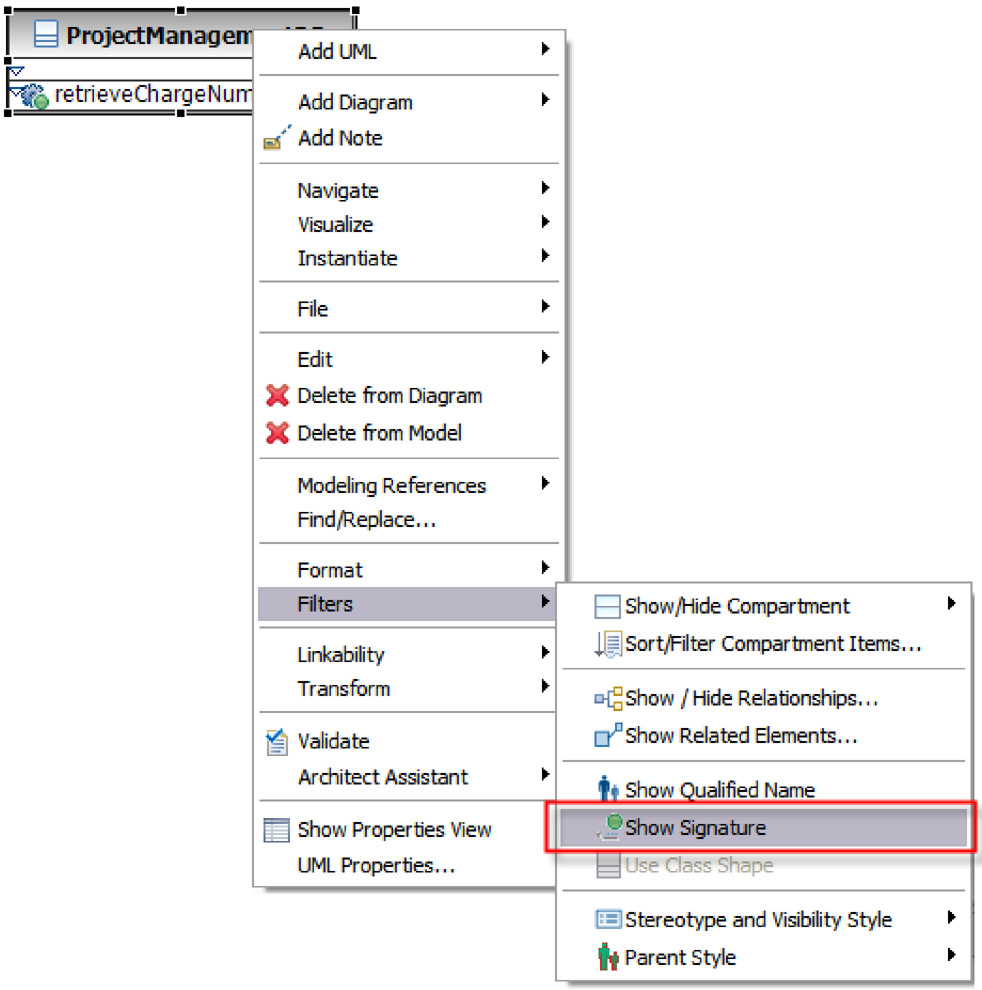
If the payment delivery method is direct deposit, the system creates a bank transaction and sends it to the Bank System for processing.

As a result we will create a *BankTransaction* class with the appropriate attributes. Here is the resulting model:



If, in the process of transforming the responsibility, you have created new classes, move these classes to the *ProjectManagementServices* package. (As stated previously, this package will contain the interface corresponding to the subsystem as well as associated data.) Add the classes to the *Main* diagram in the *ProjectManagementServices* package.

To see the “full signature” of the operation in the diagram:



7. Keep in mind what we are up to! The analysis class *ProjectManagementDB* (with the responsibility *// retrieve charge numbers* in a *full operation*) is becoming in design:

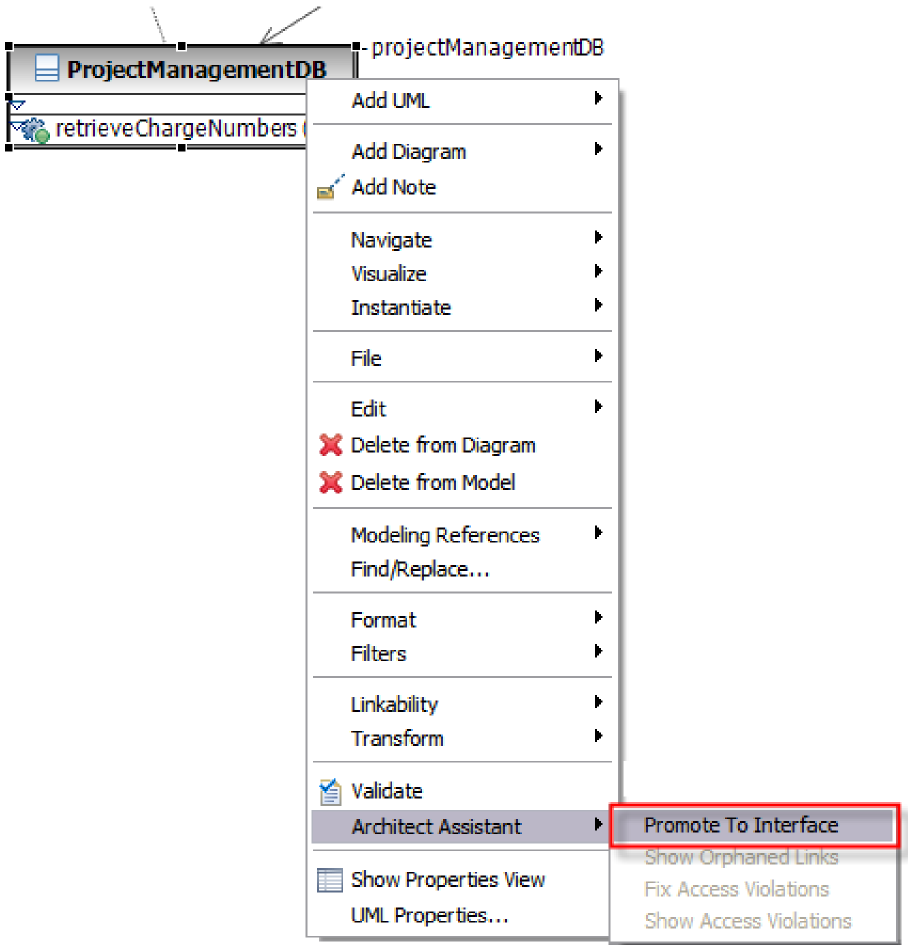
    (1) An interface *IProjectManagementDB* (with the operation *retrieveChargeNumbers()*).
    (2) A subsystem *ProjectManagementDB* that will realize the interface *IProjectManagementDB*.
    (3) A class *ProjectManagementDB* within our subsystem that will realize the interface *IProjectManagementDB* (and therefore implement the operation *retrieveChargeNumbers()*).

    For information about subsystems and subsystem interfaces, refer to the slides on p. 14 to 21 in *Part III – Object-Oriented Design* of the course (module 10, section *Identify Subsystems and Subsystem Interfaces*).

    In step 6, we moved – and modified – the analysis class *ProjectManagementDB* to the subsystem. The result of this operation is the class mentioned in (3) above. We will now create the interface (1). The *ArchitectAssistant* plug-in you installed earlier on provides the functionality to “promote” a class to an interface. This functionality applied to *ProjectManagementDB* will perform the following:

- Create the interface *IProjectManagementDB*
- Create the interface realization from *ProjectManagementDB* to *IProjectManagementDB*
- Duplicate the *retrieveChargeNumbers()* operation
- “Migrate” relationships from *ProjectManagementDB* to *IProjectManagementDB*
- Change the type of every lifeline from *ProjectManagementDB* to *IProjectManagementDB*

To create the interface *IProjectManagementDB*, right-click *ProjectManagementDB* and select *Architect Assistant > Promote To Interface*:



Carefully review the comments in the console to verify that the changes are what you expected.

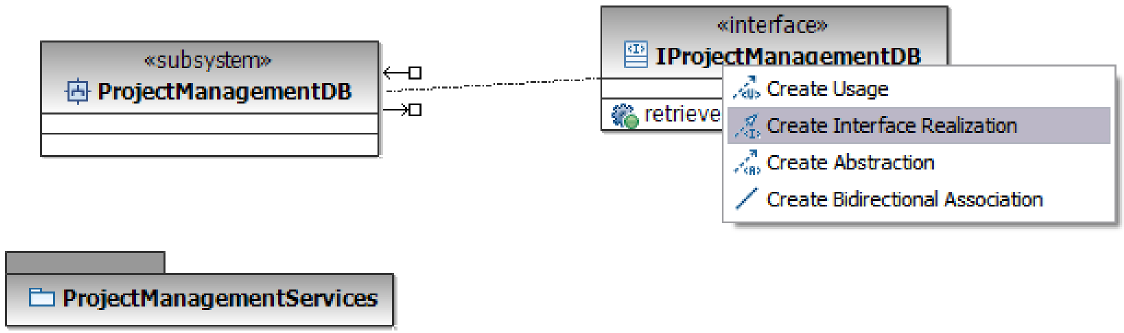
You now need to complete the work performed by *ArchitectAssistant* as follows:

a. Locate the interface *IProjectManagementDB* in the Project Explorer and move it to the *ProjectManagementServices* package.

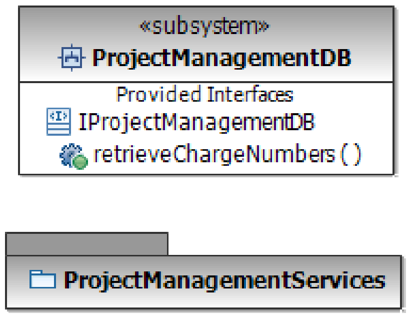
b. Add *IProjectManagementDB* to the *Main* diagram of the *ProjectManagementServices.*

c. In the *Subsystem ProjectManagementDB* diagram of the subsystem *ProjectManagementDB*, add the interface *IProjectManagementDB.*

d. In the *Main* diagram of the *<<layer>> Business* package, add the interface *IProjectManagementDB.* Draw an interface realization relationship from the subsystem *ProjectManagementDB* to the interface.

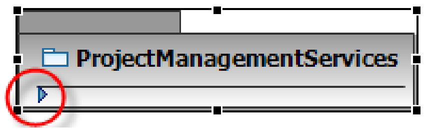


You can remove the interface from the diagram, which should now look something like: (hint: to obtain the same look and feel, use the *Appearance* tab in the *Properties* view for the subsystem)

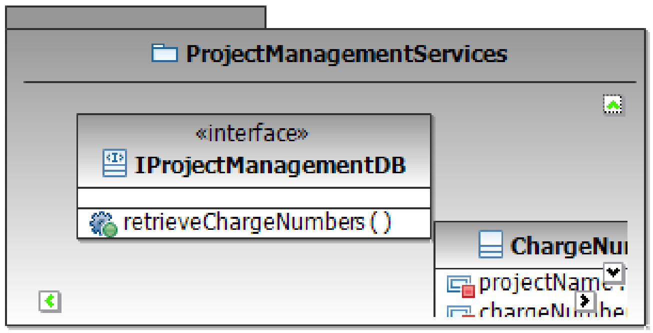


You can also show the contents of the *ProjectManagementServices* package as follows:

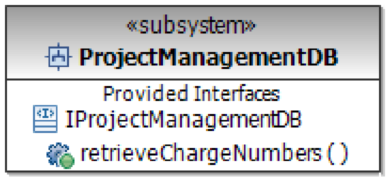
a. First in the *Appearance* tab of the *Properties* view of the package, check *Package Contents*. Make sure to click the triangle as shown below.

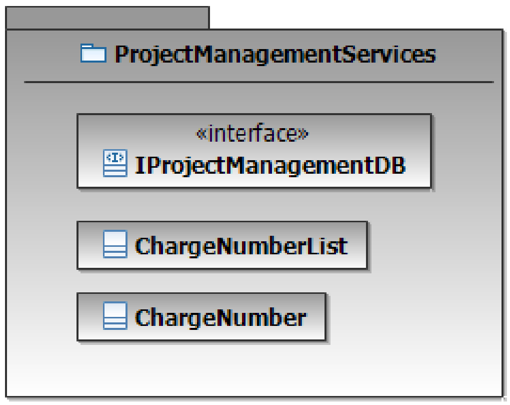


b. Then enlarge the rectangle and drag the classes contained in *ProjectManagementServices* to the *Package Contents* compartment.



c. The end result might be something like:



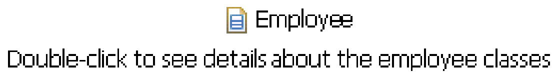


Lab 7 – Class Design

1. For this exercise, you will work from the model *06. Design*.

2. Inspect the model:

   a. From the model's *Main* diagram, double-click on the *Business* package, then on the *Employee* diagram shortcut:



Employee

Double-click to see details about the employee classes

   b. The diagram contains the class *Employee*, its subclasses and several associated classes.
      i. The class *Paycheck* is not displayed not to overload the diagram.
      ii. Note the attributes in classes like *Employee* and *Timecard*: Most of them are now typed except the *hours per project* attribute in *Timecard*, for which we didn't have enough information.
      iii. In the *Employee* class, note that most operations are shown with their I/O parameters and return values. For the operation *getMethodPayment()*, we have introduced a new class of type *Enumeration*.
      iv. In the class *PurchaseOrder*, getters have been added.
      v. Finally, note that the compositions *Employee > Timecard*, and *CommissionedEmployee > PurchaseOrder* are unidirectional.
      vi. In order to type the miscellaneous attributes and parameters, the classes *Date* and *Double* have been added to the *Primitives* package.

3. You have been asked to complete the following points in the diagram:

   a. For the remainder of the exercise, use ONLY the types already defined (including *Date* and *Double*):
      i. A time interval or period (of time) will be expressed with two attributes (or parameters) of type *Date*, for instance *from : Date* and *to : Date*.

   b. Change the responsibility *// retrieve amount to pay* in the *Employee* class into a *calculatePay()* operation:
      i. *calculatePay()* returns the amount to pay an employee for a given period of time.
      ii. It must be designed with polymorphism in mind: it must be possible to calculate the amount to pay for an *Employee* object WITHOUT knowing its type (hourly, salaried, commissioned).

c. Update the analysis class *Timecard*:
    i. Make sure in particular that the analysis attribute *hours per project* is converted to a proper design type.

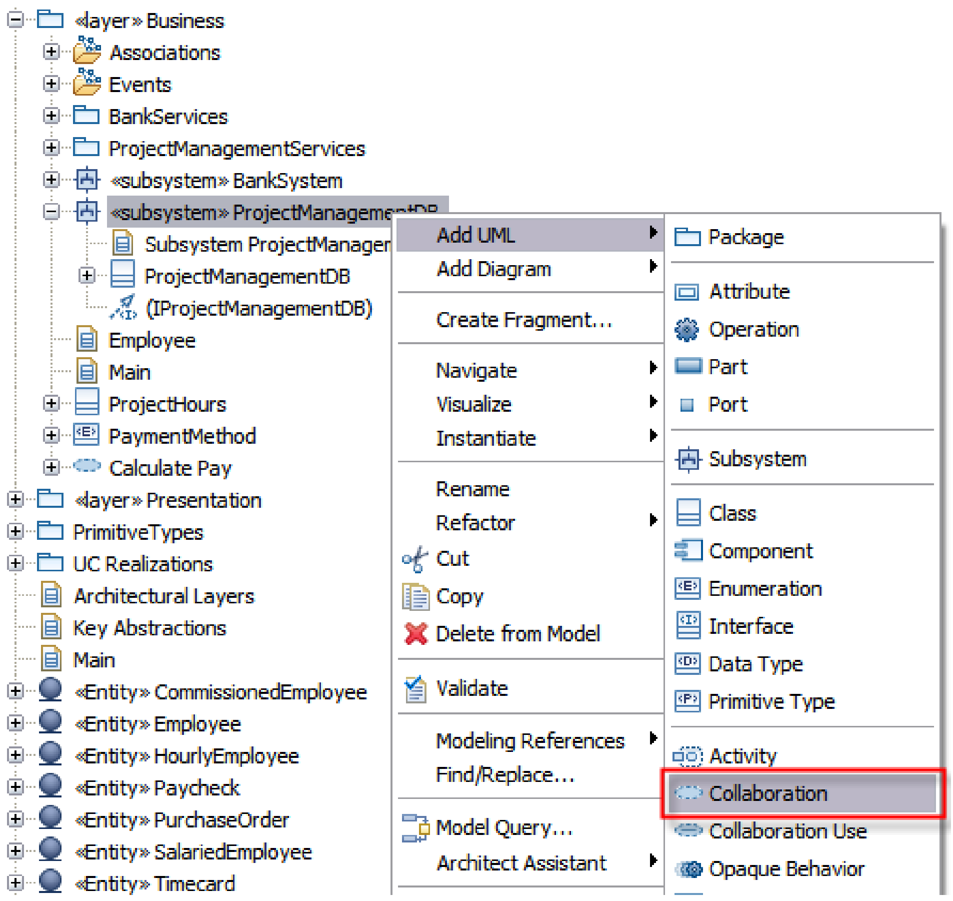
d. A new requirement has been added: it must be possible to change an hourly employee into a salaried employee. What do you suggest to handle this situation? (Discussion with the instructor.)

Lab 8 – Subsystem Design

1. For this exercise, you will work from the model *07. Design*.

2. Your task is to apply the JDBC mechanism described in module 13 to the subsystem *ProjectManagementDB*:

   a. In the *ProjectManagementDB* subsystem, create a collaboration named *IProjectManagementDB Implementation*.

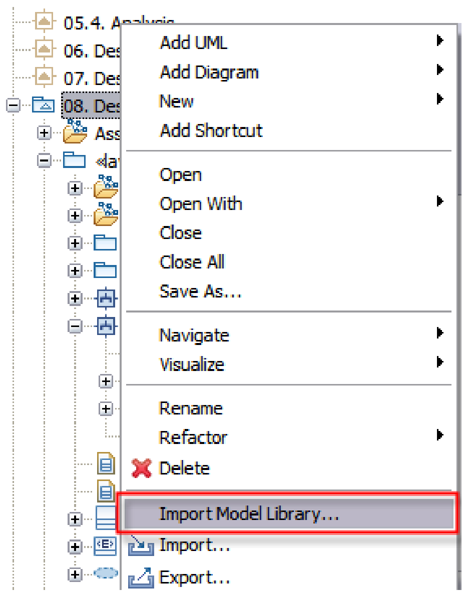


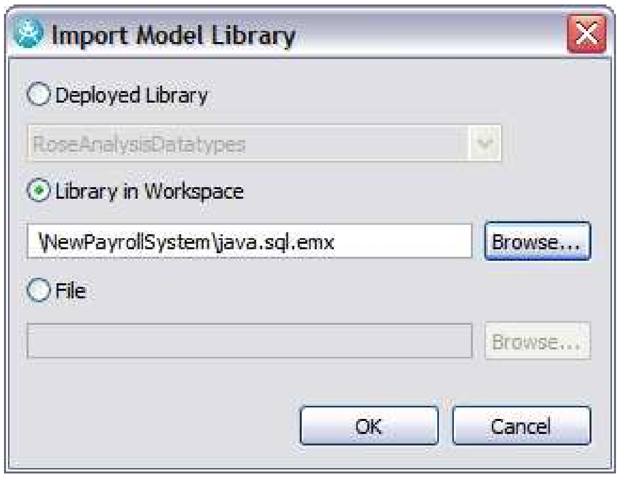
   b. In the collaboration, create a *retrieveChargeNumbers VOPC* class diagram and a sequence diagram titled *retrieveChargeNumbers Implementation*. Add these diagrams to the subsystem main diagram.

   c. To apply the JDBC mechanism, we will use a pattern:

i. Open the *Subsystem ProjectManagementDB* diagram (main diagram of the subsystem *ProjectManagementDB*).

ii. Import the *java.sql* library:

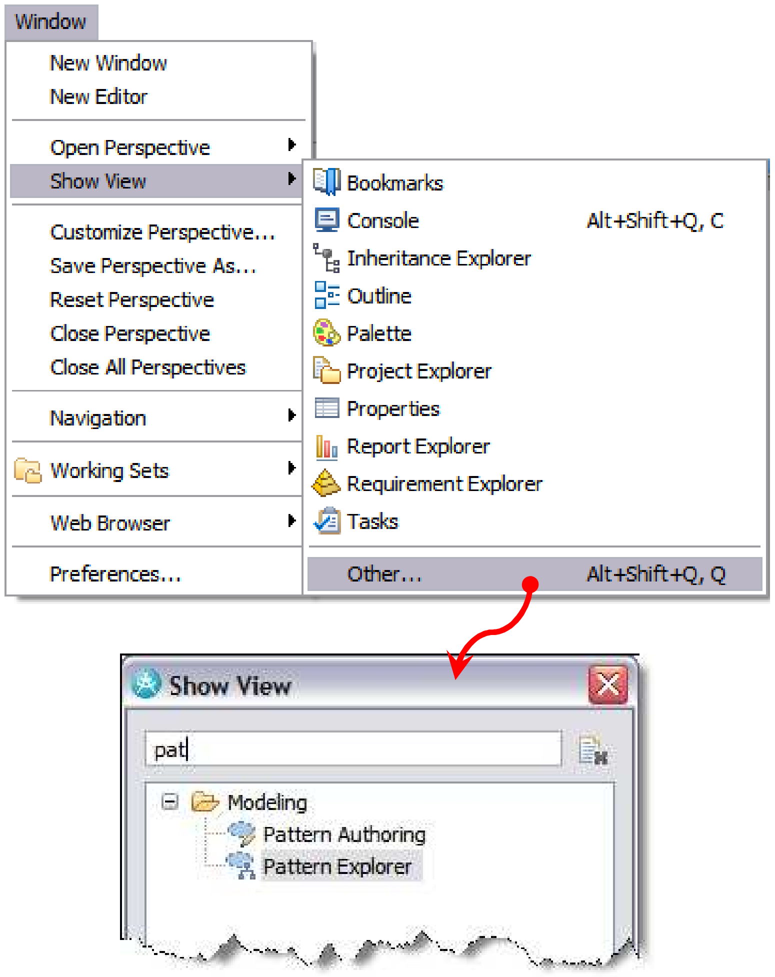




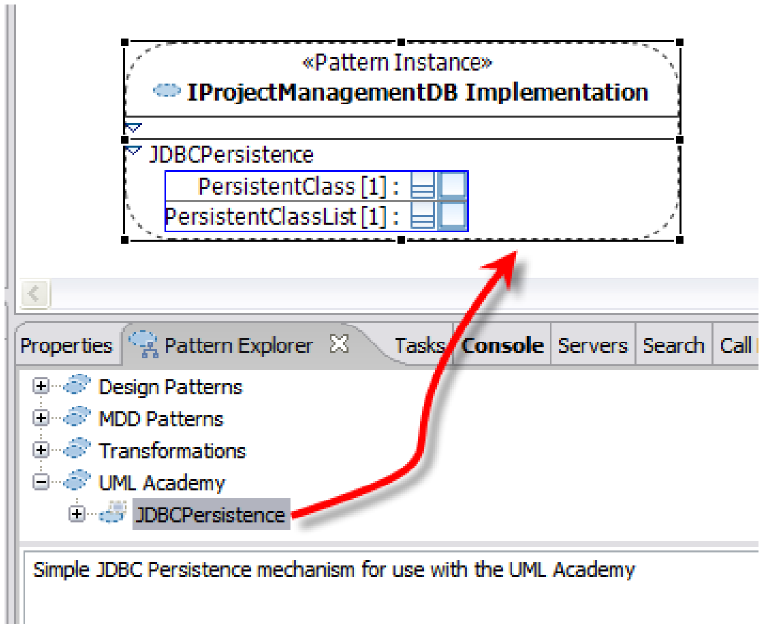
[✓ Library in Workspace]

\NewPayrollSystem\java.sql.emx

iii. Open the Pattern Explorer view:

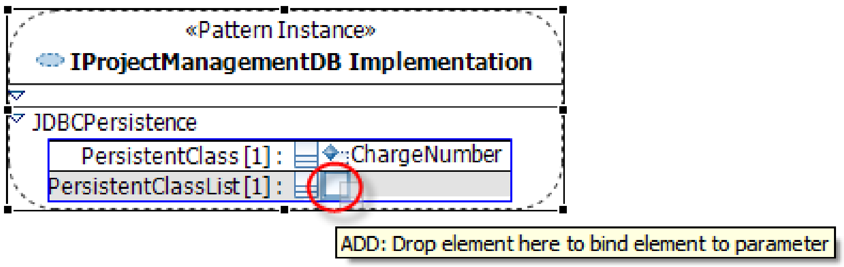


iv. In the *Pattern Explorer*, you should see the *UML Academy* group. Expand it, select the pattern *JDBCPersistence* and drag it onto the *Subsystem ProjectManagementDB* diagram. (If you don't see *UML Academy*, please check the installation steps in task 3.1.)



Simple JDBC Persistence mechanism for use with the UML Academy

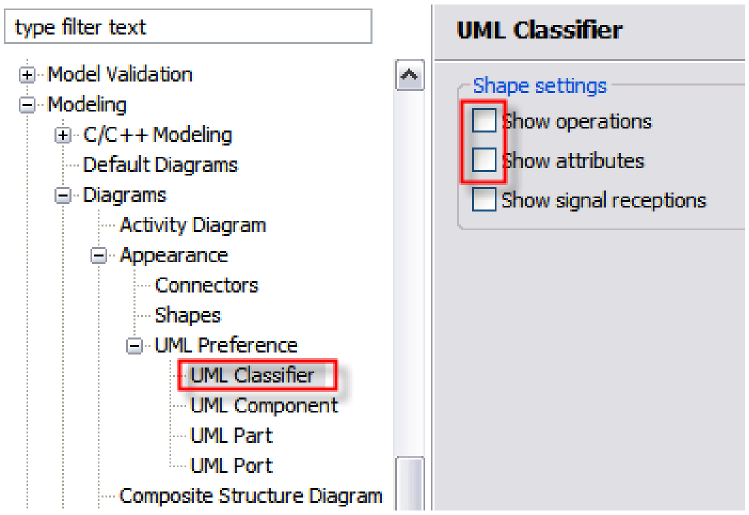
v. Drag the classes *ChargeNumber* and *ChargeNumberList* onto the pattern parameters *PersistentClass* and *PersistentClassList* respectively.


«Pattern Instance»
IProjectManagementDB Implementation
JDBCPersistence
PersistentClass [1] :
ChargeNumber
PersistentClassList [1] :
ADD: Drop element here to bind element to parameter

vi. Open the *retrieveChargeNumbers VOPC* diagram and add the *DBChargeNumber*, *ChargeNumber* and *ChargeNumberList* classes to the diagram. (Note: *DBChargeNumber* was created by the pattern.)

Note: The current version of the *JDBCPersistence* pattern will create an aggregation relationship from the *PersistentClassList* role to the *PersistentClass* role, even if it already exists. You should remove the duplicate relation if it exists.

vii. Modify the display preferences for classes as shown below (*Window > Preferences …*): (the purpose is to allow us to display the *java.sql* classes without showing the many attributes and operations of these classes)


type filter text

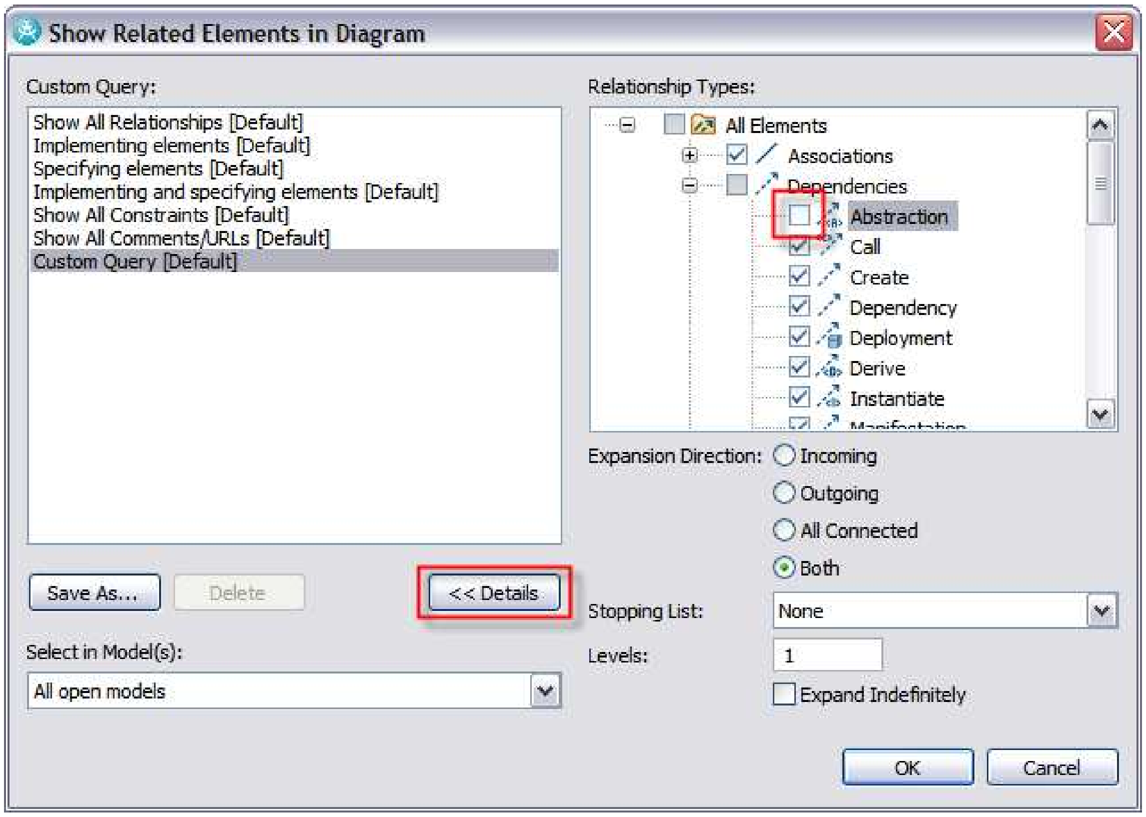
- Model Validation
- Modeling
  - C/C++ Modeling
  - Default Diagrams
  - Diagrams
    - Activity Diagram
    - Appearance
      - Connectors
      - Shapes
      - UML Preference
        - UML Classifier
        - UML Component
        - UML Part
        - UML Port
    - Composite Structure Diagram

UML Classifier

Shape settings

- [ ] Show operations
- [ ] Show attributes
- [ ] Show signal receptions

viii. Right-click *DBChargeNumber* and select *Filters > Show Related Elements…*. In the *Show Related Elements in Diagram* dialog, click *Details* and clear the *Abstraction* entry as shown below. (These relationships are used to show the relationship of the classes involved in the pattern to the pattern instance.)


Show Related Elements in Diagram

Custom Query:

Show All Relationships [Default]
Implementing elements [Default]
Specifying elements [Default]
Implementing and specifying elements [Default]
Show All Constraints [Default]
Show All Comments/URLs [Default]
Custom Query [Default]

Relationship Types:

- [ ] All Elements
  - [x] Associations
  - [ ] Dependencies
    - [ ] Abstraction
    - [x] Call
    - [x] Create
    - [x] Dependency
    - [x] Deployment
    - [x] Derive
    - [x] Instantiate
    - [x] Manifestation

Expansion Direction: Incoming / Outgoing / All Connected / [✓ Both]

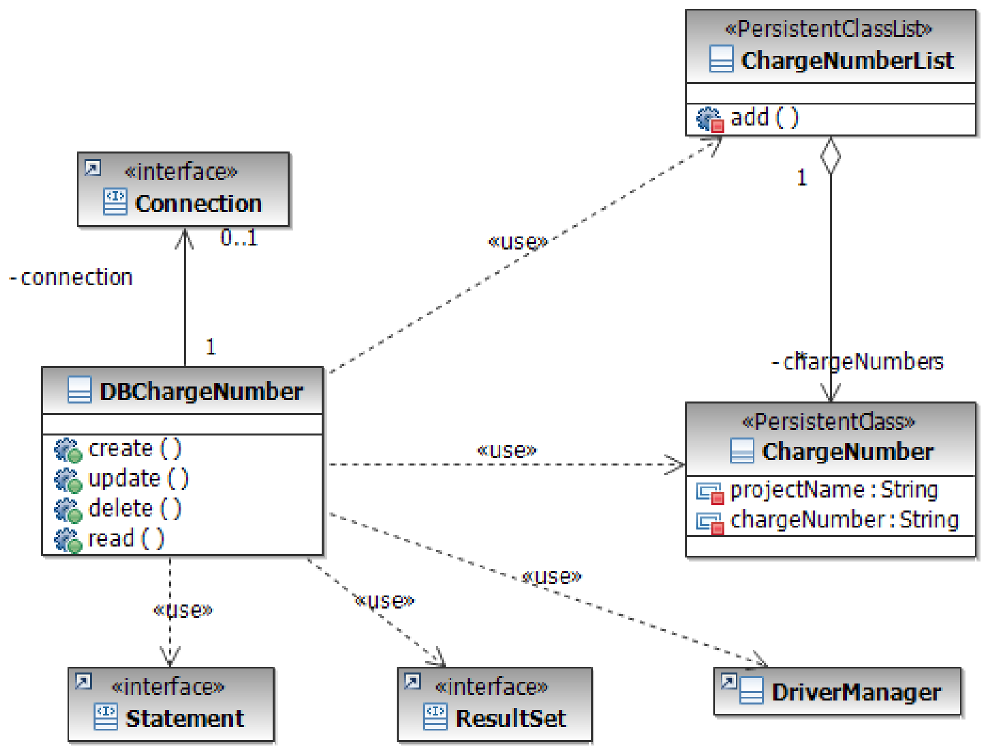
Stopping List: None

Levels: 1

- [ ] Expand Indefinitely

Select in Model(s): All open models

ix. Your diagram should now look something like this:



x. Restore the display preferences for classes if you wish (see step vii).

xi. Optionally, build the sequence diagram for *retrieveChargeNumbers Implementation* based on the diagram on page 48 of the *Part III – Object-Oriented Design* fascicule.

Lab 9 – Finalize the Design Model

For this exercise, you will work from the model *08. Design*.

All the classes you have created must be assigned to packages. The packages in turn must be assigned to the layers of your architecture. Relations between classes determine the dependencies between packages. The potential for reuse, the scalability and the flexibility of your system depend on the resulting hierarchy.

Allocate all classes to packages contained within the *Presentation* and *Business* layers. Please refer to the following slides for information:

- In PART II – Object-Oriented Analysis, slides 58 to 60 (about Layered Architectures)
- In PART II – Object-Oriented Analysis, slides 85, 86, 89, 90, 94, 95 (about Analysis Classes and their positioning in the architecture)
- In PART III – Object-Oriented Design, slides 23 to 28 (about the Organization of the Models)

Here are a few high-level suggestions (note: some of these suggestions are minimalist – in real life, the package structure would be significantly different):

1. Stereotype the package *UC Realizations* as <<perspective>> (this means that *UC Realizations* only contains diagrams and no classes or other UML elements).
2. Add the stereotypes *global* and *layer* to the package *PrimitivesTypes* (again reality would be quite different but it does not modify our message).
3. Consider creating two packages in the *Presentation* layer (for instance *AdminActivities* and *EmployeeActivities*). Hint: Application-dependent control classes and some of the boundary classes should be allocated to these packages.
4. All other classes are allocated to packages in the *Business* layer. Consider the potential reuse you can expect for those classes (in particular for service-oriented classes).

Solve the access violations:

1. Right-click on the *Design* model in the *Project Explorer*, then select *ArchitectAssistant > Show Access Violations*. You should get a number of access violations reported in the console.
2. To solve the access violations, use the *ArchitectAssistant > Fix Access Violations*. To keep this processing manageable, it is recommended to work on one package at a time. *Fix Access Violations* attempts to automatically resolve the access violations based on the layered architecture. When a potential access violation is identified, *Fix Access Violations* can:
    a. Create a dependency if the violation can be solved automatically.
    b. Report an error in the case of a “true” violation (for instance if the *Business* layer attempts to access the *Presentation* layer).

    c. Prompt the user in all other cases.

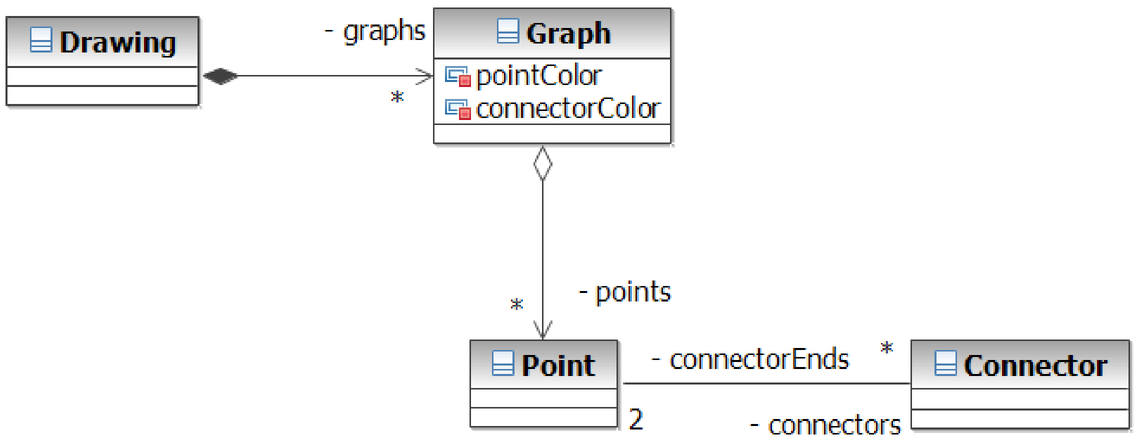
3. Inspect the *Console* and/or *Problems* view to identify the remaining access violations and modify the model to eliminate them.
4. When all packages have been processed, run the *ArchitectAssistant > Show Access Violations* on the model itself until you get no access violations left.

Create a diagram titled *Package Dependencies* in the *Design* model (at the same level as *Main*, *Key Abstractions*, and *Architectural Layers* diagrams). Add to this diagram all the packages and subsystems from the layers. Inspect the result.

Appendix 1 – Lab 2 Solutions

This appendix 1 offers possible solutions for the Lab 2 exercises. There are many possible variations.

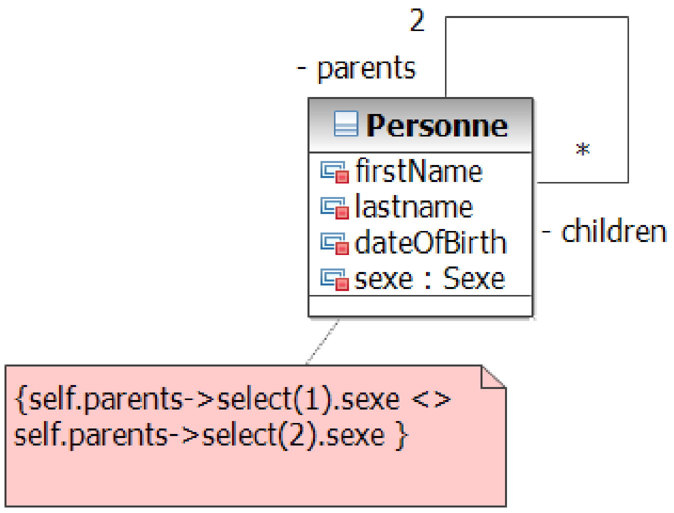
Modeling graphs, points and connectors



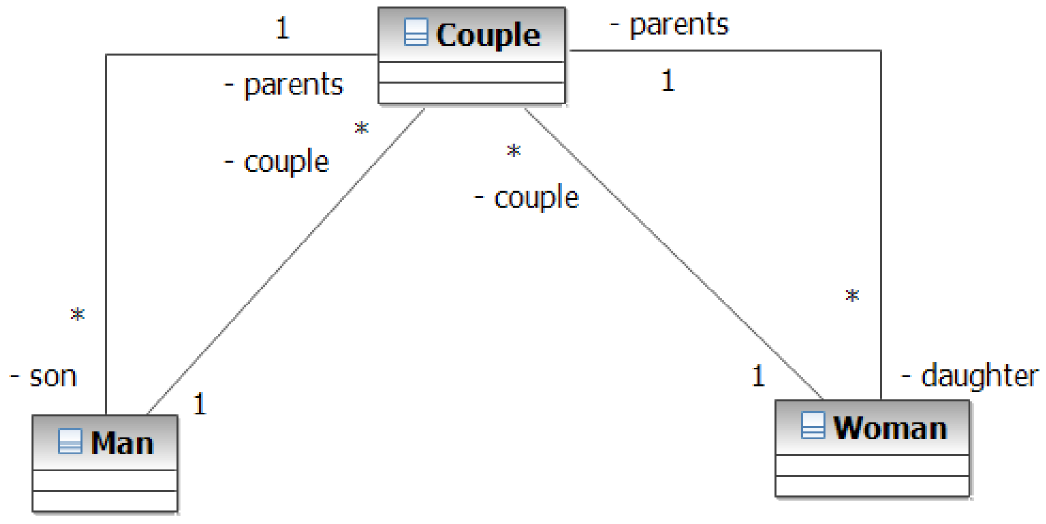
An additional constraint (text or OCL) is needed to express the fact that there can be only one connector between 2 points.

Modeling a family tree (without generalization)

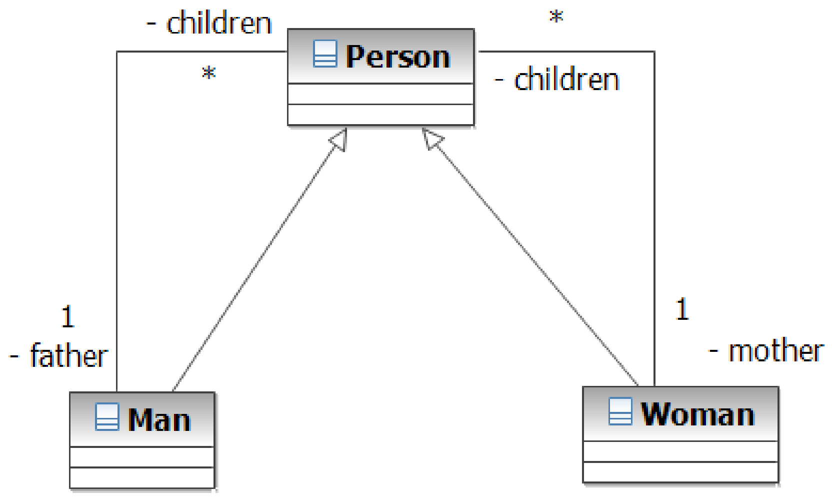
Here is a very simple solution:



And here is a variant:

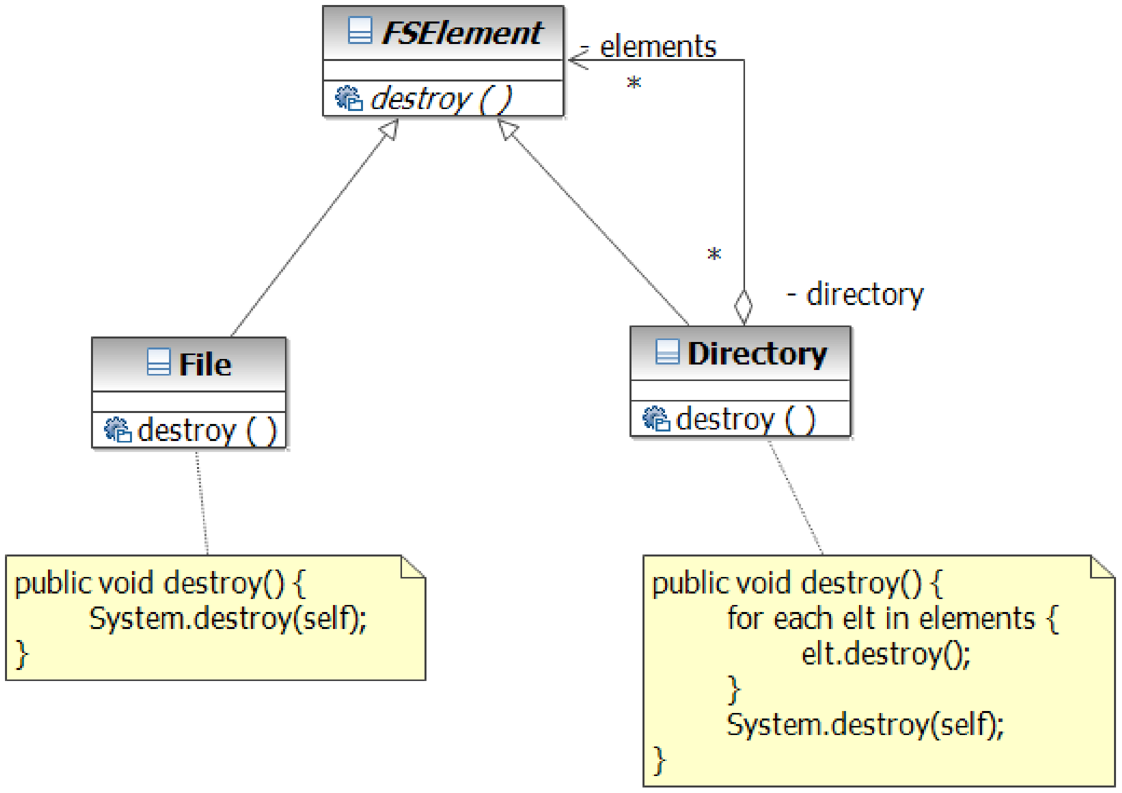


Modeling a family tree (with generalization)



Modeling a file system

The solution below is an application of the *Composite* design pattern.



```
public void destroy() {
        System.destroy(self);
}
```

```
public void destroy() {
        for each elt in elements {
                elt.destroy();
        }
        System.destroy(self);
}
```

Appendix 2 – Course Registration Requirements

Problem Statement

As the head of information systems for Wylie College you are tasked with developing a new student registration system. The college would like a new client-server system to replace its much older system developed around mainframe technology. The new system will allow students to register for courses and view report cards from personal computers attached to the campus LAN. Professors will be able to access the system to sign up to teach courses as well as record grades.

Due to a decrease in federal funding, the college cannot afford to replace the entire system at once. The college will keep the existing course catalog database where all course information is maintained. This database is an Ingres relational database running on a DEC VAX. Fortunately the college has invested in an open SQL interface that allows access to this database from the college's Unix servers. The legacy system performance is rather poor, so the new system must ensure that access to the data on the legacy system occurs in a timely manner. The new system will access course information from the legacy database but will not update it. The registrar's office will continue to maintain course information through another system.

At the beginning of each semester, students may request a course catalogue containing a list of course offerings for the semester. Information about each course, such as professor, department, and prerequisites, will be included to help students make informed decisions. The new system will allow students to select four course offerings for the coming semester. In addition, each student will indicate two alternative choices in case the student cannot be assigned to a primary selection. Course offerings will have a maximum of ten students and a minimum of three students. A course offering with fewer than three students will be canceled. For each semester, there is a period of time that students can change their schedule. Students must be able to access the system during this time to add or drop courses. Once the registration process is completed for a student, the registration system sends information to the billing system so the student can be billed for the semester. If a course fills up during the actual registration process, the student must be notified of the change before submitting the schedule for processing.

At the end of the semester, the student will be able to access the system to view an electronic report card. Since student grades are sensitive information, the system must employ extra security measures to prevent unauthorized access.

Professors must be able to access the on-line system to indicate which courses they will be teaching. They will also need to see which students signed up for their course offerings. In addition, the professors will be able to record the grades for the students in each class.

Glossary

Introduction

This document is used to define terminology specific to the problem domain, explaining terms, which may be unfamiliar to the reader of the use-case descriptions or other project documents. Often, this document can be used as an informal data dictionary, capturing data definitions so that use-case descriptions and other project documents can focus on what the system must do with the information.

Definitions

The glossary contains the working definitions for the key concepts in the Course Registration System.

Course
A class offered by the university.

Course Offering
A specific delivery of the course for a specific semester – you could run the same course in parallel sessions in the semester. Includes the days of the week and times it is offered.

Course Catalog
The unabridged catalog of all courses offered by the university.

Faculty
All the professors teaching at the university.

Finance System
The system used for processing billing information.

Grade
The evaluation of a particular student for a particular course offering.

Professor
A person teaching classes at the university.

Report Card
All the grades for all courses taken by a student in a given semester.

Roster
All the students enrolled in a particular course offering.

Student
A person enrolled in classes at the university.

Schedule
The courses a student has selected for the current semester.

Transcript
The history of the grades for all courses, for a particular student sent to the finance system, which in turn bills the students.

Supplementary Specifications

Objectives

The purpose of this document is to define requirements of the Course Registration System. This Supplementary Specification lists the requirements that are not readily captured in the use cases of the use-case model. The Supplementary Specifications and the use-case model together capture a complete set of requirements on the system.

Scope

This specification defines the non-functional requirements of the system; such as reliability, usability, performance, and supportability, as well as functional requirements that are common across a number of use cases. (The functional requirements are defined in the Use Case Specifications.)

References

None.

Functionality

Multiple users must be able to perform their work concurrently.
If a course offering becomes full while a student is building a schedule including that offering, the student must be notified.

Usability

The desktop user-interface shall be Windows 95/98 compliant.

Reliability

The system shall be available 24 hours a day 7 days a week, with no more than 10% down time.

Performance

The system shall support up to 2000 simultaneous users against the central database at any given time, and up to 500 simultaneous users against the local servers at any one time.
The system shall provide access to the legacy course catalog database with no more than a 10 second latency.
Note: Risk-based prototypes have found that the legacy course catalog database cannot meet our performance needs without some creative use of mid-tier processing power
The system must be able to complete 80% of all transactions within 2 minutes.

Supportability

None.

Security

The system must prevent students from changing any schedules other than their own, and professors from modifying assigned course offerings for other professors.
Only Professors can enter grades for students.
Only the Registrar is allowed to change any student information.

Design Constraints

The system shall integrate with an existing legacy system, the Course Catalog System, which is an RDBMS database.
The system shall provide a Windows-based desktop interface.

Register for Courses UC

Brief Description

This use case allows a Student to register for course offerings in the current semester. The Student can also update or delete course selections if changes are made within the add/drop period at the beginning of the semester. The Course Catalog System provides a list of all the course offerings for the current semester.

Flow of Events

Basic Flow

This use case starts when a Student wishes to register for course offerings, or to change his/her existing course schedule.

1. The Student provides the function to perform (one of the sub flows is executed):
   If the Student selected “Create a Schedule”, the Create a Schedule subflow is executed.
   If the Student selected “Update a Schedule”, the Update a Schedule subflow is executed.
   If the Student selected “Delete a Schedule”, the Delete a Schedule subflow is executed.

Create a Schedule

1. The system retrieves a list of available course offerings from the Course Catalog System and displays the list to the Student.
2. The Select Offerings subflow is executed.
3. The Submit Schedule subflow is executed.

Update a Schedule

1. The system retrieves and displays the Student’s current schedule (e.g., the schedule for the current semester).
2. The system retrieves a list of available course offerings from the Course Catalog System and displays the list to the Student.
3. The Student may update the course selections on the current selection by deleting and adding new course offerings. The Student selects the course offerings to add from the list of available course offerings. The Student also selects any course offerings to delete from the existing schedule.
4. Once the student has made his/her selections, the system updates the schedule for the Student using the selected course offerings.
5. The Submit Schedule subflow is executed.

Delete a Schedule

1. The system retrieves and displays the Student’s current schedule (e.g., the schedule for the current semester).
2. The system prompts the Student to confirm the deletion of the schedule.
3. The Student verifies the deletion.
4. The system deletes the Schedule. If the schedule contains “enrolled in” course offerings, the Student must be removed from the course offering.

Select Offerings

The Student selects 4 primary course offerings and 2 alternate course offerings from the list of available offerings.

Once the student has made his/her selections, the system creates a schedule for the Student containing the selected course offerings.

Submit Schedule
For each selected course offering on the schedule not already marked as "enrolled in", the system verifies that the Student has the necessary prerequisites, that the course offering is open, and that there are no schedule conflicts.

The system then adds the Student to the selected course offering. The course offering is marked as "enrolled in" in the schedule.

The schedule is saved in the system.

Alternative Flows

Save a Schedule
At any point, the Student may choose to save a schedule rather than submitting it. If this occurs, the Submit Schedule step is replaced with the following:

The course offerings not marked as "enrolled in" are marked as "selected" in the schedule.

The schedule is saved in the system.

Unfulfilled Prerequisites, Course Full, or Schedule Conflicts
If, in the Submit Schedule sub-flow, the system determines that the Student has not satisfied the necessary prerequisites, or that the selected course offering is full, or that there are schedule conflicts, an error message is displayed. The Student can either select a different course offering and the use case continues, save the schedule, as is (see Save a Schedule subflow), or cancel the operation, at which point the Basic Flow is re-started at the beginning.

No Schedule Found
If, in the Update a Schedule or Delete a Schedule sub-flows, the system is unable to retrieve the Student's schedule, an error message is displayed. The Student acknowledges the error, and the Basic Flow is re-started at the beginning.

Course Catalog System Unavailable
If the system is unable to communicate with the Course Catalog System, the system will display an error message to the Student. The Student acknowledges the error message, and the use case terminates.

Course Registration Closed
When the use case starts, if it is determined that registration for the current semester has been closed, a message is displayed to the Student, and the use case terminates. Students cannot register for course offerings after registration for the current semester has been closed.

Delete Cancelled
If, in the Delete A Schedule sub-flow, the Student decides not to delete the schedule, the delete is cancelled, and the Basic Flow is re-started at the beginning.

Special Requirements

None.

Pre-Conditions

The Student must be logged onto the system before this use case begins.

Post-Conditions

If the use case was successful, the student schedule is created, updated, or deleted. Otherwise, the system state is unchanged.

Appendix 3 – Payroll System

Problem Statement

As the head of Information Technology at Acme, Inc., you are tasked with building a new payroll system to replace the existing system, which is hopelessly out of date. Acme needs a new system to allow employees to record timecard information electronically and automatically generate paychecks based on the number of hours worked and total amount of sales (for commissioned employees).

The new system will be state of the art and will have a Windows-based desktop interface to allow employees to enter timecard information, enter purchase orders, change employee preferences (such as payment method), and create various reports. The system will run on individual employee desktops throughout the entire company. For reasons of security and auditing, employees can only access and edit their own timecards and purchase orders.

The system will retain information on all employees in the company (Acme currently has around 5,000 employees world-wide.) The system must pay each employee the correct amount, on time, by the method that they specify (see possible payment methods described later). Acme, for cost reasons, does not want to replace one of their legacy databases, the Project Management Database, which contains all information regarding projects and charge numbers. The new system must work with the existing Project Management Database, which is a DB2 database running on an IBM mainframe. The Payroll System will access, but not update, information stored in the Project Management Database.

Some employees work by the hour, and they are paid an hourly rate. They submit timecards that record the date and number of hours worked for a particular charge number. If someone works for more than 8 hours, Acme pays them 1.5 times their normal rate for those extra hours. Hourly workers are paid every Friday.

Some employees are paid a flat salary. Even though they are paid a flat salary, they submit timecards that record the date and hours worked. This is so the system can keep track of the hours worked against particular charge numbers. They are paid on the last working day of the month.

Some of the salaried employees also receive a commission based on their sales. They submit purchase orders that reflect the date and amount of the sale. The commission rate is determined for each employee, and is one of 10%, 15%, 25%, or 35%.

One of the most requested features of the new system is employee reporting. Employees will be able to query the system for number of hours worked, totals of all hours billed to a project (i.e., charge number), total pay received year-to-date, remaining vacation time, etc.

Employees can choose their method of payment. They can have their paychecks mailed to the postal address of their choice, or they can request direct deposit and have their paycheck deposited into a bank account of their choosing. The employee may also choose to pick their paychecks up at the office.

The Payroll Administrator maintains employee information. The Payroll Administrator is responsible for adding new employees, deleting employees and changing all employee information such as name, address, and payment classification (hourly, salaried, commissioned), as well as running administrative reports.

The payroll application will run automatically every Friday and on the last working day of the month. It will pay the appropriate employees on those days. The system will be told what date the employees are to be paid, so it will generate payments for records from the last time the employee was paid to the specified date. The new system is being designed so that the payroll will always be generated automatically, and there will be no need for any manual intervention.

Glossary

Introduction

This document is used to define terminology specific to the problem domain, explaining terms, which may be unfamiliar to the reader of the use-case descriptions or other project documents. Often, this document can be used as an informal data dictionary, capturing data definitions so that use-case descriptions and other project documents can focus on what the system must do with the information.

Definitions

The glossary contains the working definitions for the key concepts in the Payroll System.

Bank System
Any bank(s) to which direct deposit transactions are sent.

Employee
A person that works for the company that owns and operates the payroll system (Acme, Inc.)

Payroll Administrator
The person responsible for maintaining employees and employee information in the system.

Project Management Database
The legacy database that contains all information regarding projects and charge numbers.

System Clock
The internal system clock that keeps track of time. The internal clock will automatically run the payroll at the appropriate times.

Pay Period
The amount of time over which an employee is paid.

Paycheck
A record of how much an employee was paid during a specified Pay Period.

Payment Method
How the employee is paid, either pick-up, mail, or direct deposit.

Timecard
A record of hours worked by the employee during a specified pay period.

Purchase Order
A record of a sale made by an employee.

Salaried Employee
An employee that receives a salary.

Commissioned Employee
An employee that receives a salary plus commissions.

Hourly Employee
An employee that is paid by the hour.

Supplementary Specifications

Objectives

The purpose of this document is to define requirements of the Payroll System. This Supplementary Specification lists the requirements that are not readily captured in the use cases of the use-case model. The Supplementary Specifications and the use-case model together capture a complete set of requirements on the system.

Scope

This Supplementary Specification applies to the Payroll System, which will be developed by the OOAD students.
This specification defines the non-functional requirements of the system; such as reliability, usability, performance, and supportability as well as functional requirements that are common across a number of use cases. (The functional requirements are defined in the Use Case Specifications.).

Functionality

None.

Usability

None.

Reliability

The main system must be running 98% of the time. It is imperative that the system be up and running during the times the payroll is run (every Friday and the last working day of the month).

Performance

The system shall support up to 2000 simultaneous users against the central database at any given time, and up to 500 simultaneous users against the local servers at any one time.

Supportability

None.

Security

The system should prevent employees from changing any timecards other than their own. Additionally, for security reasons, only the Payroll Administrator is allowed to change any employee information with the exception of the payment delivery method.

Design Constraints

The system shall integrate with an existing legacy system, the Project Management Database, which is a DB2 database running on an IBM mainframe.
The system shall interface with existing bank systems via an electronic transaction interface
The system shall provide a Windows-based desktop interface.

Maintain Timecard UC

Brief Description

This use case allows the Employee to update and submit timecard information. Hourly and salaried employees must submit weekly timecards recording all hours worked that week and which projects the hours are billed to. An Employee can only make changes to the timecard for the current pay period and before the timecard has been submitted.

Flow of Events

Basic Flow

This use case starts when the Employee wishes to enter hours worked into his current timecard.

1. The system retrieves and displays the current timecard for the Employee. If a timecard does not exist for the Employee for the current pay period, the system creates a new one. The start and end dates of the timecard are set by the system and cannot be changed by the Employee.
2. The system retrieves and displays the list of available charge numbers from the Project Management Database.
3. The Employee selects the appropriate charge numbers and enters the hours worked for any desired date (within the date range of the timecard).
4. Once the Employee has entered the information, the system saves the timecard.

Submit Timecard

1. At any time, the Employee may request that the system submit the timecard.
2. At that time, the system assigns the current date to the timecard as the submitted date and changes the status of the timecard to "submitted." No changes are permitted to the timecard once it has been submitted.
3. The system validates the timecard by checking the number of hours worked against each charge number. The total number of hours worked against all charge numbers must not exceed any limit established for the Employee (for example, the Employee may not be allowed to work overtime).
4. The system retains the number of hours worked for each charge number in the timecard.
5. The system saves the timecard.
6. The system makes the timecard read-only, and no further changes are allowed once the timecard is submitted.

Alternative Flows

Invalid Number of Hours
If, in the Basic Flow, an invalid number of hours is entered for a single day (>24), or the number entered exceeds the maximum allowable for the Employee, the system will display an error message and prompt for a valid number of hours. The Employee must enter a valid number, or cancel the operation, in which case the use case ends.

Timecard Already Submitted
If, in the Basic Flow, the Employee's current timecard has already been submitted, the system displays a read-only copy of the timecard and informs the Employee that the timecard has

already been submitted, so no changes can be made to it. The Employee acknowledges the message and the use case ends.

Project Management Database Not Available
If, in the Basic Flow, the Project Management Database is not available, the system will display an error message stating that the list of available charge numbers is not available. The Employee acknowledges the error and may either choose to continue (without selectable charge numbers), or to cancel (any timecard changes are discarded and the use case ends). Note: Without selectable charge numbers, the Employee may change hours for a charge number already listed on the timecard, but he/she may not add hours for a charge number that is not already listed.

Special Requirements

None.

Pre-Conditions

The Employee must be logged onto the system before this use case begins.

Post-Conditions

If the use case was successful, the Employee timecard information is saved to the system. Otherwise, the system state is unchanged.

Run Payroll UC

Brief Description

The use case describes how the payroll is run every Friday and the last working day of the month.

Flow of Events

Basic Flow

4. The use case begins when it's time to run the payroll. The payroll is run automatically every Friday and the last working day of the month.
5. The system retrieves all employees who should be paid on the current date.
6. The system calculates the pay using entered timecards, purchase orders, employee information (e.g., salary, benefits, etc.) and all legal deductions.
7. If the payment delivery method is mail or pick-up, the system prints a paycheck.
8. If the payment delivery method is direct deposit, the system creates a bank transaction and sends it to the Bank System for processing.
9. The use case ends when all employees receiving pay for the desired date have been processed.

Alternative Flows

Bank System Unavailable
If the Bank System is down, the system will attempt to send the bank transaction again after a specified period. The system will continue to attempt to re-transmit until the Bank System becomes available.

Deleted Employees
After the payroll for an Employee has been processed, if the employee has been marked for deletion (see the Maintain Employee use case), then the system will delete the employee.

Special Requirements

None.

Pre-Conditions

None.

Post-Conditions

Payments for each employee eligible to be paid on the current date have been processed.