Tersus Tutorial

Table of Contents

Stage 1 - Introduction

About the Tersus Platform

Welcome to the Tersus Platform.

With Tersus, you can easily create web applications by drawing diagrams instead of writing code.

The Tersus technology has already been used successfully to create a range of software solutions, from small tactical applications to high-end, mission critical systems for processing financial transactions.

Tersus is especially appropriate for composite applications assembled from a combination of built-in components, self developed components and Web Services.

The **Tersus Platform** comprises three major components:

- The **Tersus Studio**, an extension of the Eclipse platform, used by modelers (developers and business experts) to graphically define the functionality of applications;
- The **Tersus Model Libraries**, containing building blocks for assembling applications;
- The **Tersus Server**, which executes the modeled solutions and performs the required database updates (can run over a J2EE application server).

Creating an application is done by defining a Model Hierarchy, in which each model is composed of lower level components. The developer starts at a top-level diagram representing the whole system, and then continues with an iterative top-down refinement process – drilling down from each model to specify its components. Employing an “infinite drawing board” that represents graphically the whole model hierarchy, it is possible for the developer to fully and precisely specify the required business logic in a visual and intuitive manner.

Deploying the application, once modeled, is immediate. The models are saved as a hierarchy of XML files, which are read by the Tersus Runtime Engine. The engine then performs the functionality defined by the models at all levels – user interface, server-side processing and database operations. It is possible to record the full details of the execution if tracing is required for auditing purposes or for root cause analysis of problems.

Maintaining an existing application is done by amending its model – changing the business flow, adding new components, or disabling redundant components. Upon completion of the required modifications, the application can be redeployed immediately.

About this Tutorial

This tutorial outlines the development of a complete sample application - a Purchase Requisition Management system - using the Tersus Modeling Tool.

The tutorial provides a step-by-step, hands-on example of the development of such a system, starting with an employee issuing a requisition, and concluding with the delivery of requested items.

Within minutes, and then at each step as you progress with building the application, you can start the application from your browser using the Tersus Runtime Engine.

Document Conventions

This tutorial uses the following style conventions:

Convention	Description
Double-click the root model	Step-by-step modeling instructions you may follow and perform
View	Objects available in the Tersus Modeling Tool
New Requisition	Recommended names for models you create
A data structure is …	Noteworthy information which elaborates beyond the direct subject at hand

Using the Tutorial

The tutorial is divided into multiple stages, each covering several new concepts of the Tersus development methodology.

In each stage, the tutorial covers:

- The development process (modeling, or "how to model it");

- The resulting application definition (model, or "how the model looks when done"); and

- The outcome (application, "what the user gets to use").

Each stage is accompanied by a pre-built, functioning sample project implementing all stages up-to and including the current stage.

The aim of the sample projects is two-fold:

1. Provide a consistent reference when reading through the tutorial stages.
2. Provide additional functionality which is similar to functionality modeled in detail in the tutorial. You can use the samples and skip the modeling of the additional functionality, or decide to model it yourself (the additional functionality is described in short rather than in a detailed step-by-step manner, but given what you have done earlier in the same stage, these descriptions should be sufficient for you to model the functionality yourself).

It is highly recommended that you follow the tutorial stage by stage and in the order outlined.

The Sample Application

Acme corp. wants to replace its manual, paper-intensive purchase requisition process with a computerized web-based system that will allow employees to request the purchase of products (e.g. PCs, furniture, office supplies).

A purchase requisition always goes through, at the very least, the following steps:

1. An **employee** enters a new requisition;
2. The requisition is approved/rejected by the employee's **manager;**
3. The approved requisition is handled by a **purchaser,** who issues a purchase order;
4. When the items ordered are delivered, the requisition is considered fulfilled and closed.

Stage 2 – Modeling a Basic Display

Stage Goals

This stage introduces the **Tersus Studio**.

You will learn how to create a new Tersus project

Tersus Concepts Covered

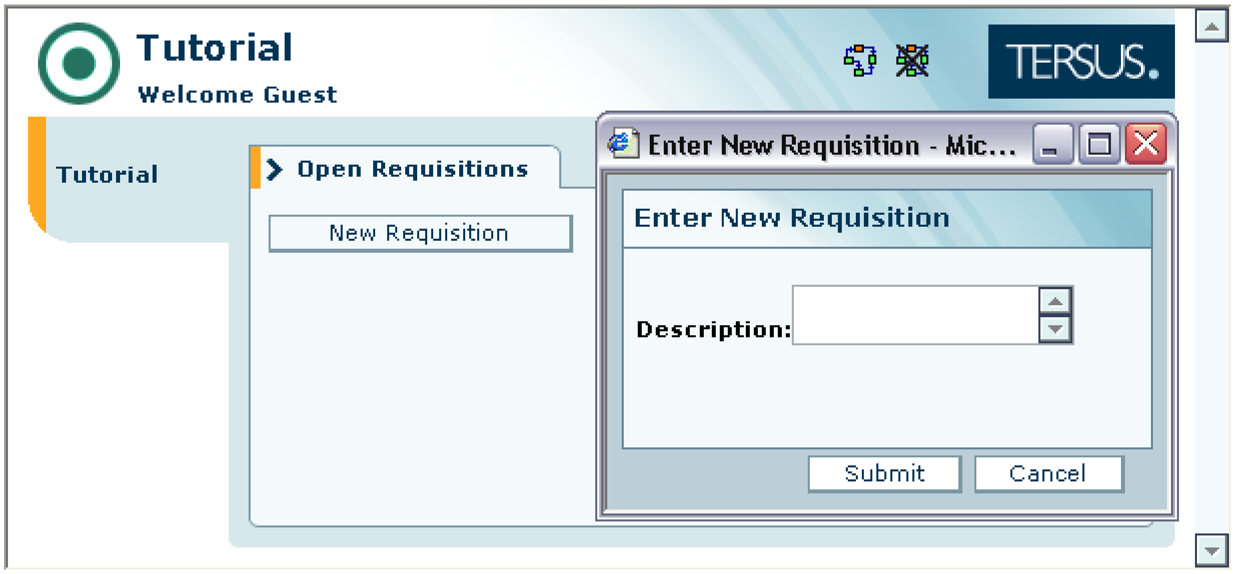
On completion of this stage you should be familiar with the following concepts:

Modeling notions:	Model, Display, Template. Model vs. Element Name.
Modeling techniques:	Creating & managing the display. Renaming a model.
Useful display templates:	View, Button, Popup, Label, Text area.

Application Functionality Modeled

In this stage you will model a browser form for entering requisitions.

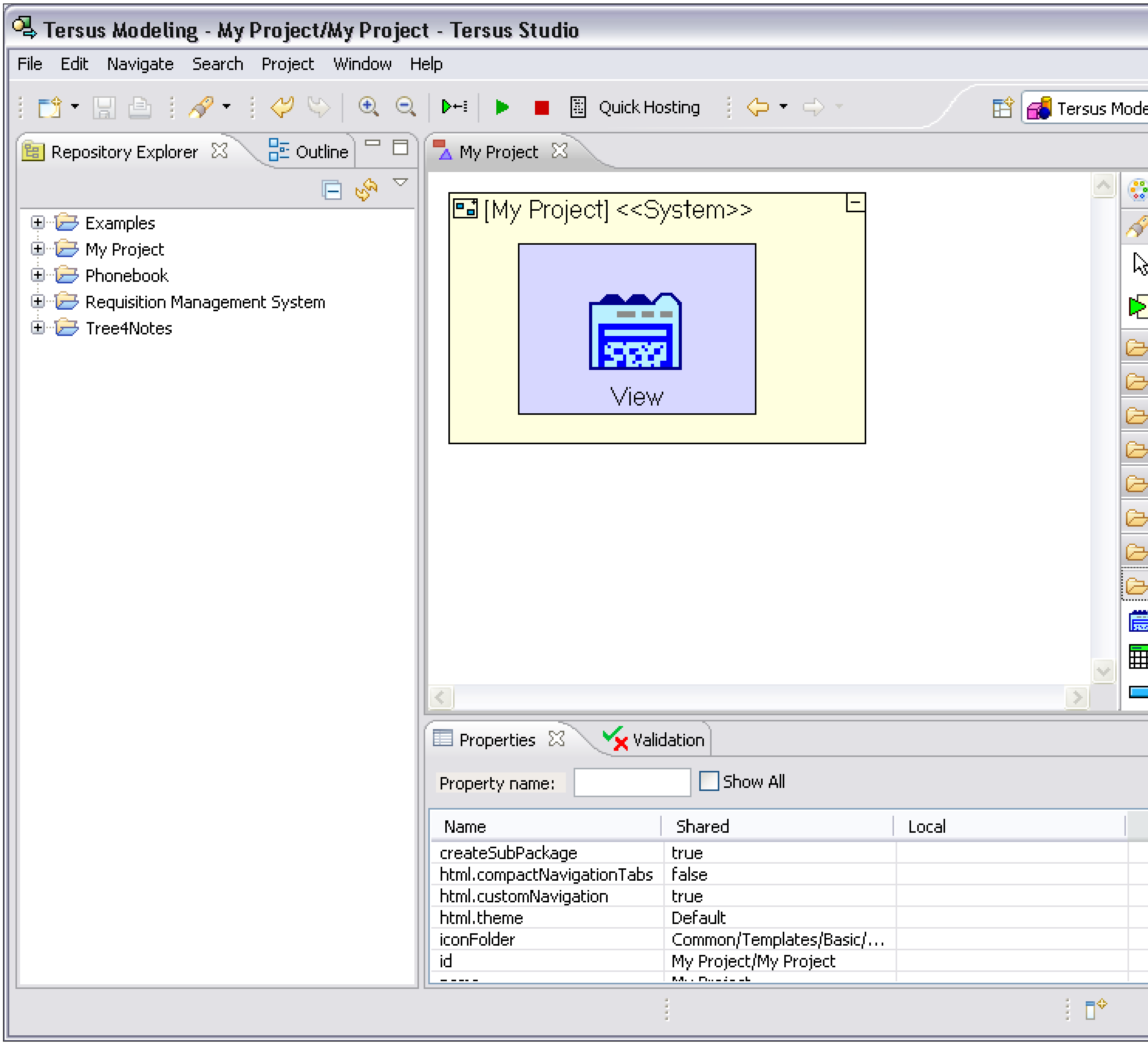
The resulting web application will include a button that opens a pop-up form used to enter a new requisition, similar to the following:



The Tersus Studio

The following screenshot displays the default appearance of the **Tersus Studio**. It includes the **Model Editor** (with integrated **Palette**) on the right, and a tabbed view containing the

Repository Explorer and **Outline** on the right.



Name	Shared	Local
createSubPackage	true	
html.compactNavigationTabs	false	
html.customNavigation	true	
html.theme	Default	
iconFolder	Common/Templates/Basic/...	
id	My Project/My Project	

If you want to learn more on the **Tersus Studio** before starting your first project, refer to Appendix A, which covers the following topics:

- Tersus Studio and the Eclipse Platform
- The Palette
- Inserting New Elements to the Model
- Selecting, Moving and Resizing Elements

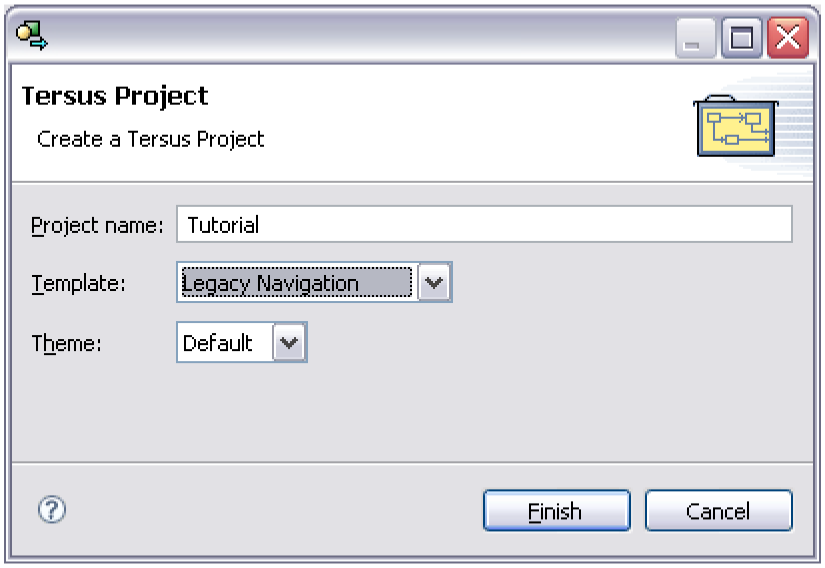
- Drill-down
- Zoom-in/out
- Undo/Redo
- The Outline
- The Repository Explorer
- The Application Server

Create a Simple Web Application

We start by creating a new application project:

Select **File -> New ->** **Tersus Project**.

Note that **Tersus Project** is the first option in the **New** sub-menu, followed by **Project...**, which is a generic eclipse option (see Appendix A for more information).



Project name: Tutorial

Template: Legacy Navigation

Theme: Default

Enter a **Project name** for your new project: ***Tutorial***.

Select the **Template**: ***Legacy Navigation***

Press the **Finish** button.

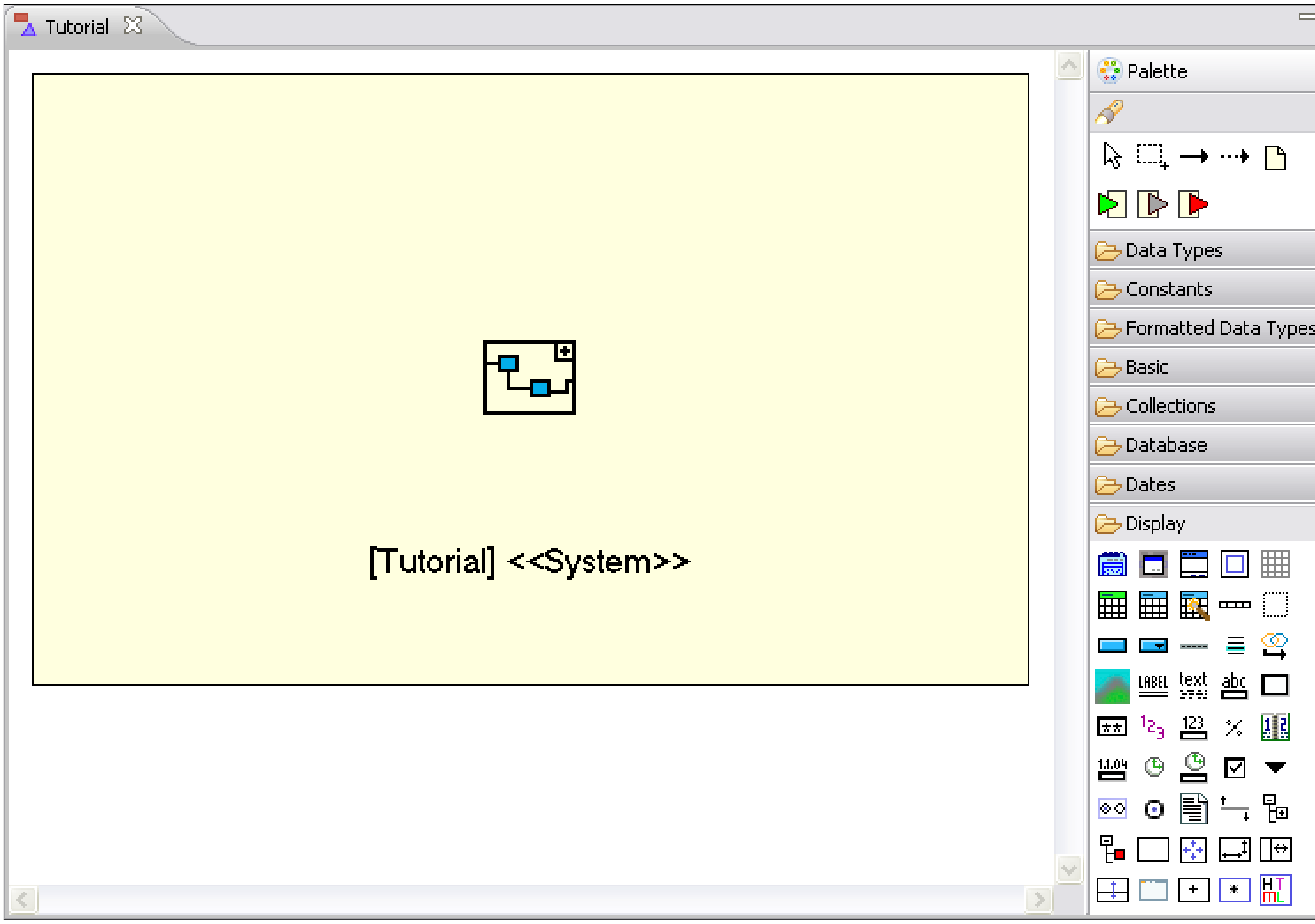
Start Modeling – Create a Form for Entering a Requisition

To start modeling we need to open the application root model in the model editor.

If you've just created a new project, **Tutorial**, it should already be open in the model editor. If not, do the following:

Locate the **Tutorial** project root (folder) in the Repository Explorer view, and double-click it.

This should open a new editor window, which should look as in the following screenshot, displaying a yellow rectangle, representing the application. Since we have not modeled anything yet, the rectangle is empty, except for its name:



Create a View (Open Requisitions)

A web application is something you can see in your browser, so first we need to define a **View**. A **View model** defines what is displayed in the browser, and contains other display elements (labels, buttons, tables, etc.).

To create the View called **Open Requisitions** (showing all your open requisitions), do the following:

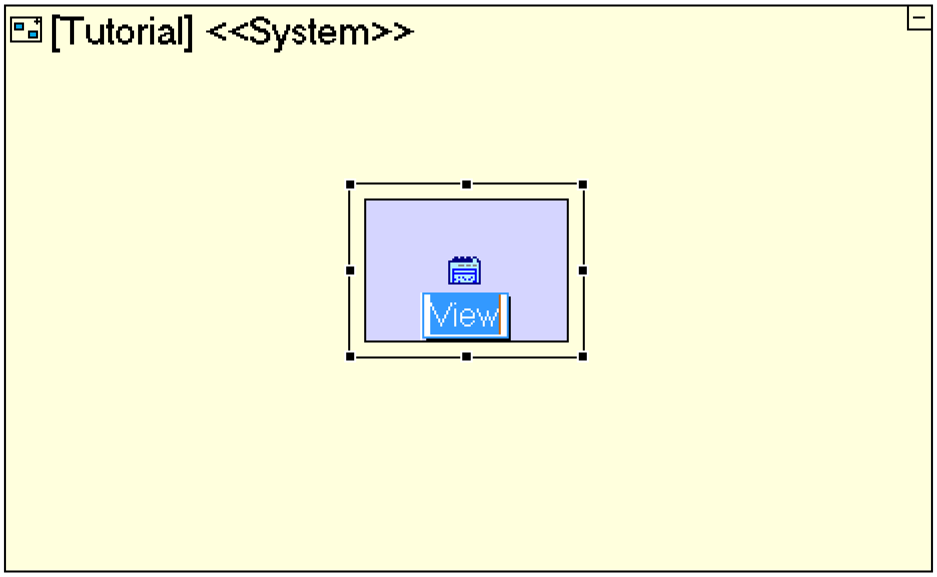
In the **Palette** (to the right of the model editor), make sure that the **Display** category is open (if it is not, just click it to open).

Click on the **View** template (▤) to select it.

Notice that when you now move the mouse pointer over the editor, it changes to signify where it is legal to drop **View** (by displaying a small, gray rectangle).

Position the mouse pointer inside the **Tutorial** root model and click to insert the view.

A view model is created, with the default name **View**, and the editor enables you to rename it immediately, as follows:

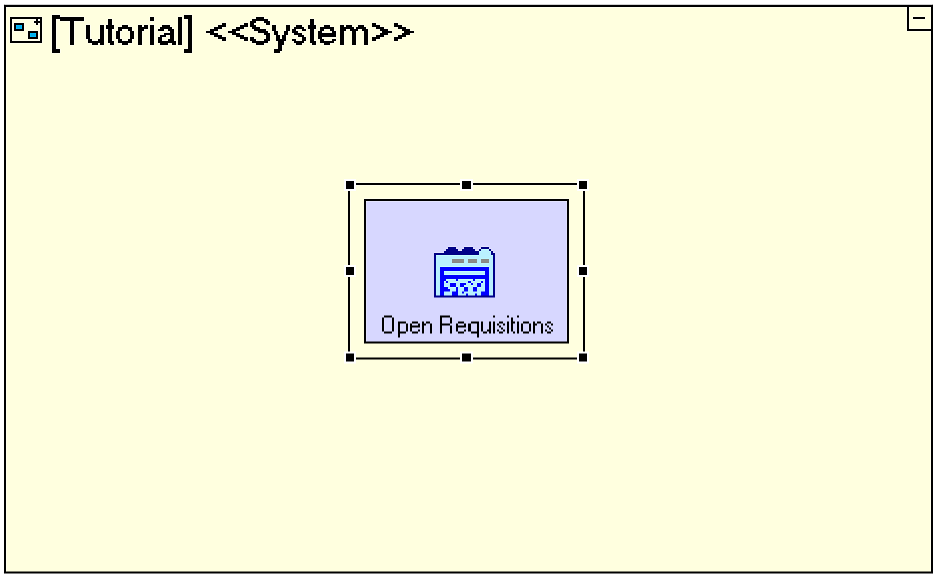


Type ***Open Requisitions*** and press **[Enter]** when finished.

If the editor exits name edit mode before you have changed it, you can rename it as follows:
1. Make sure the view model is selected
2. Open the **Rename** dialog by pressing **[F2]** (or right-click -> **Rename**)
3. Enter the new name , ***Open Requisitions***, in the **Element name (local):** field
4. Click **Finish**

For more details regarding rename and the **Rename** dialog, see the **Rename a model** section of this stage, below.

The simple model you have just created should be similar to the following:



The **Open Requisitions** view (the green/blue rectangle) is now a sub-model of the root model **Tutorial** (the yellow rectangle).

Any model in the hierarchy may contain any number of sub-models.

Note that the model screenshots provided in this tutorial may differ from the models you create, mainly because of differences in positioning and sizing of elements in the model, as well as the fact that elements which contain other elements may be expanded (marked with ⊟) or collapsed (⊞). This difference does not have an effect on application functionality, except in the case of display models, as explained later in this stage.

Add a Button (New Requisition)

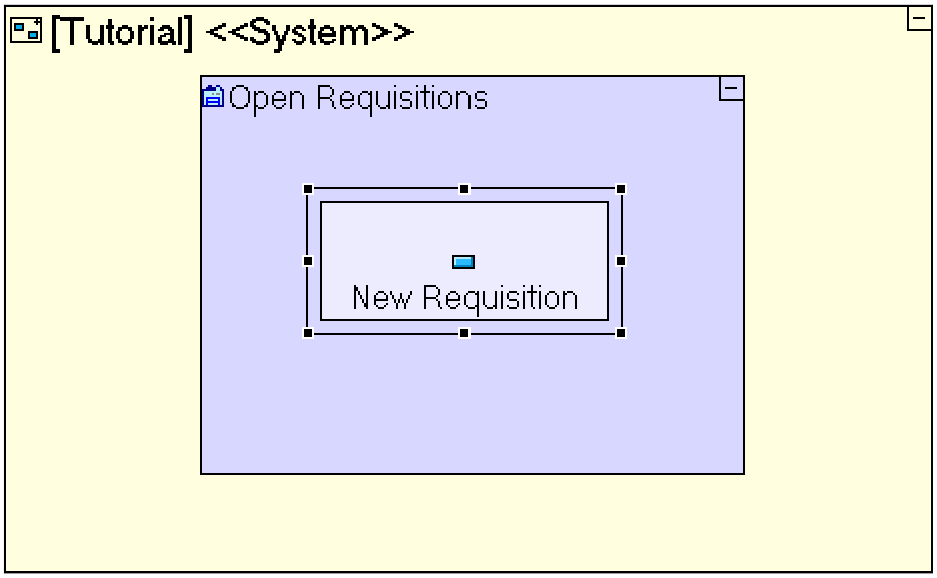
Let's continue with the modeling of the **Open Requisitions** view. We will now add a **New Requisition** button (allowing the user to enter a new requisition in a popup form).

Select the **Button** template (▭) from the **Display** category in the palette (by clicking on the template).

Insert it into the **Open Requisitions** view (by clicking inside **Open Requisitions**).

Name it ***New Requisition*** (Type ***New Requisition***).

The model should now look similar to the following:



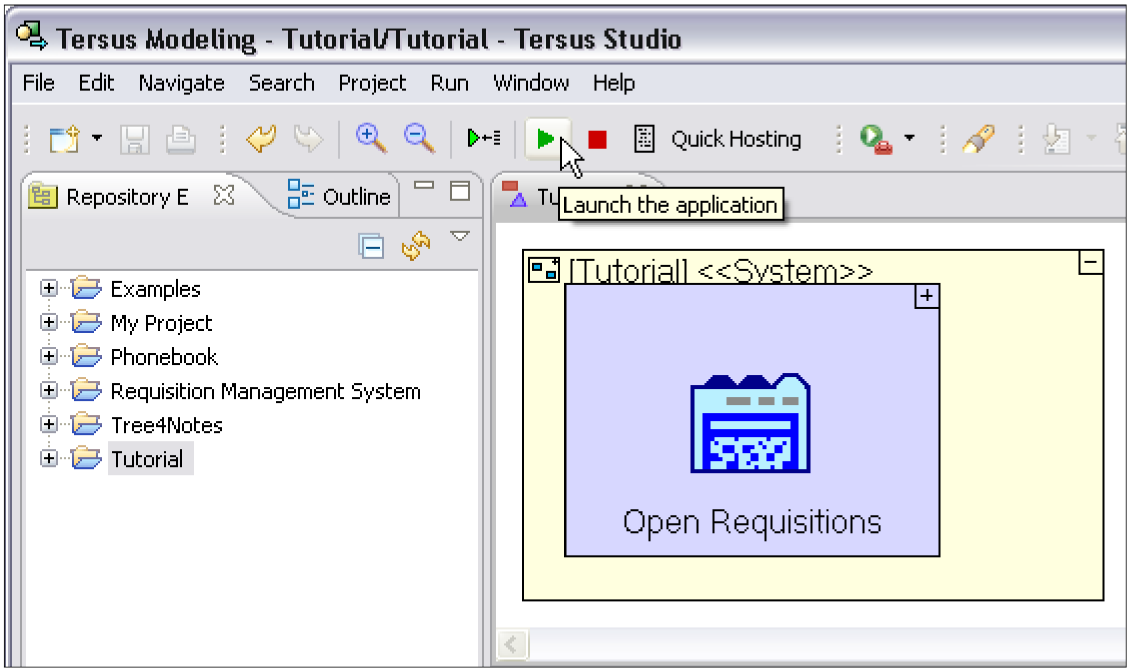
Both the **Repository Explorer** and the **Model Editor** captions display an asterisk next to them (e.g. *Tutorial). This indicates that the latest changes have not been saved.

Save your work by clicking 💾 on the toolbar.

Whenever you save your model, the Tersus Studio checks (validates) your models. If any errors are found a message will appear, and the errors will be displayed in the **Validation** view. See Completing Stage 12 for more details regarding validation.

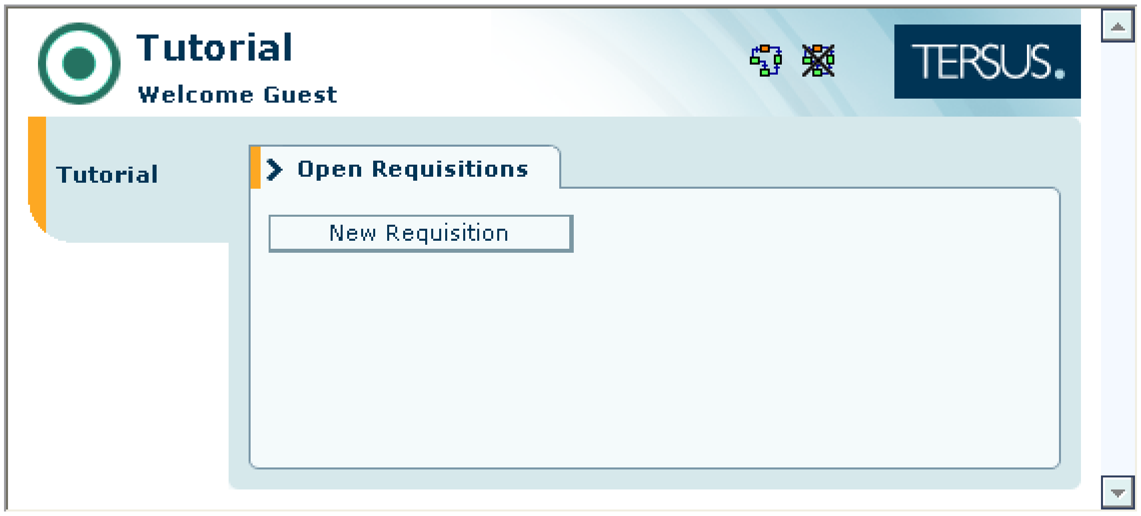
Although we have modeled very little, we can already have the first glimpse of our application in the browser:

Click on the **Launch the application** button in the studio's main toolbar to load your application in the embedded Tersus Server and open it in a web-browser.



If the **Launch the application** button is disabled, clicking on the model editor should enable it.

Your browser should display a page similar to the following:



Notice that the **Open Requisitions** view appears as a single tab. Later on, when you insert additional views, they will appear as additional tabs. Inside the view, we see the **New Requisition** button.

Switch back to the **Tersus Studio**. You may leave the browser open in the background.

Create a Popup (Enter New Requisition)

People often wonder what happens in the computer when they press a button. It would be nice if we could look through the button and see the insides of the application's logic, and this is exactly what Tersus let’s you do by using a “zoom in” technique – modeling "within" the button the actions that take place when the button is pressed.

So let’s zoom into the ***New Requisition*** button and model the popup form that appears when the button is clicked:

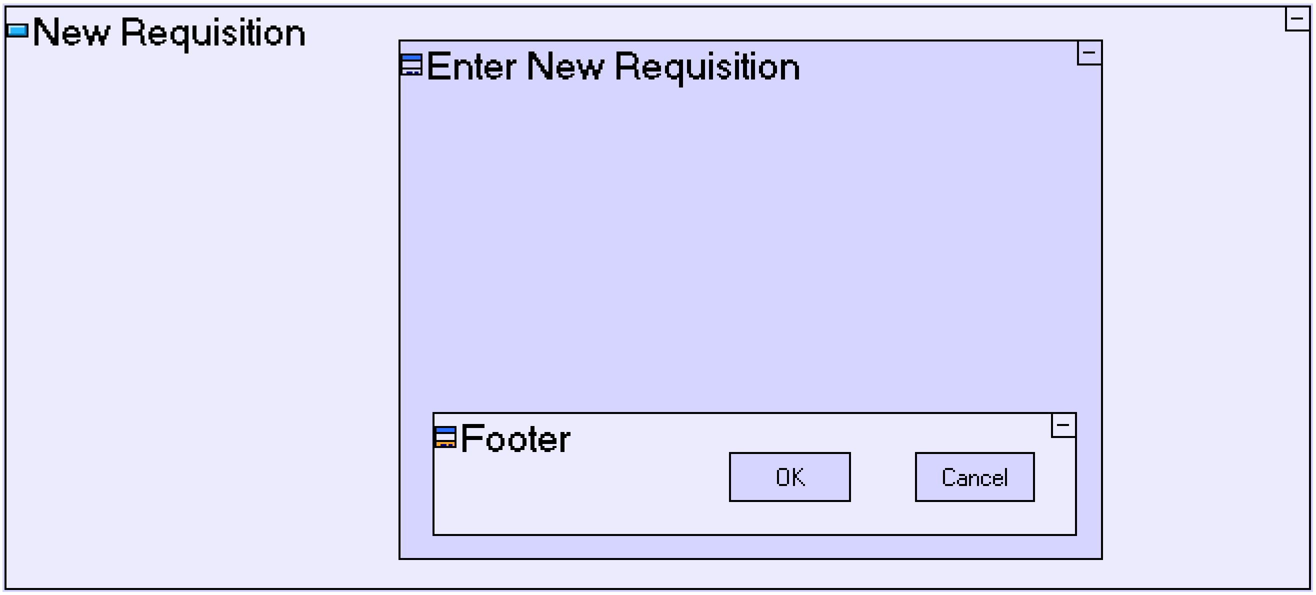
Double-click the **New Requisition** button to zoom into it.

Select the **Display/Popup** template (▤) from the palette.

Insert it into the **New Requisition** element.

Name it ***Enter New Requisition***.

Your **New Requisition** button model should now look similar to the following:

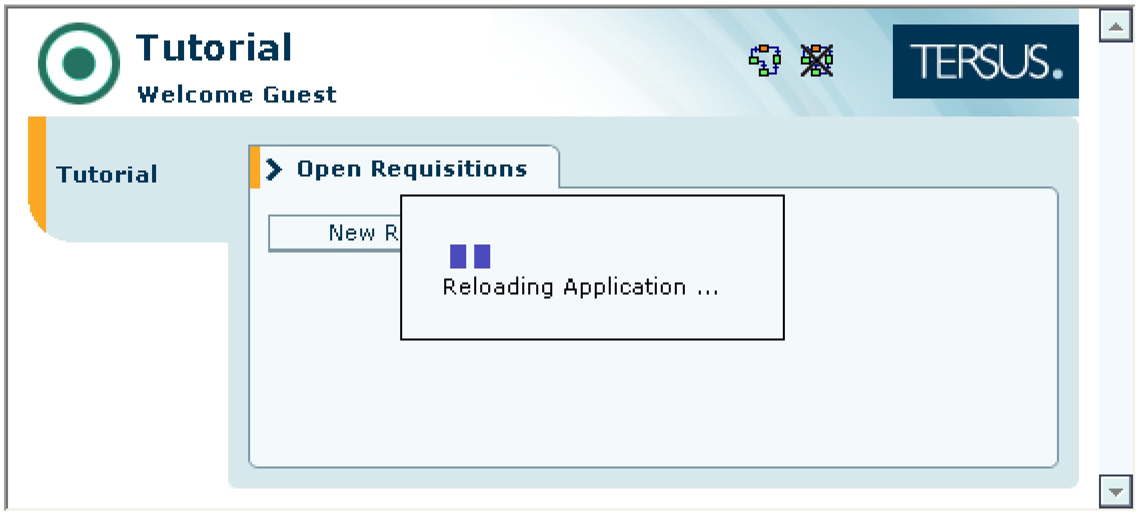


The **Popup** template is an example of a template which provides additional pre-built functionality out of the box.
In the case of a **Popup** the pre-built functionality consists of:
1. A **Footer**, which will cause its contents to be displayed at the bottom part of the popup.
2. An **OK** button, which does nothing at the present.
3. A **Cancel** button, which contains a **Close Window** model, so that when Cancel is pressed the popup will close
In general, you are free to change (or remove) this functionality as you see fit.

Save your work 💾.

Now let's see how your application looks when deployed in the browser:

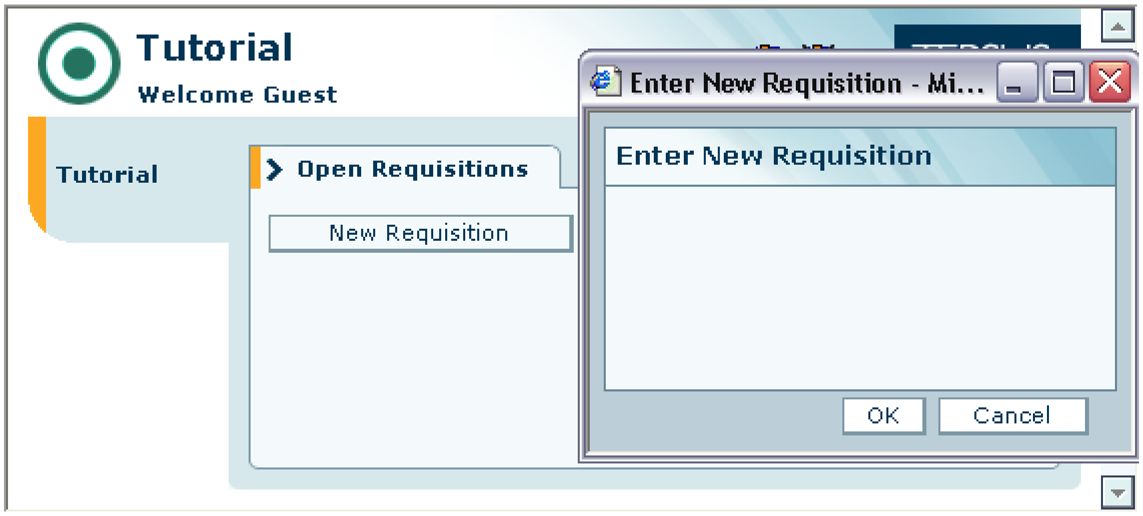
Switch back to the browser.



The browser recognizes that the application has changed since the last time it was loaded into browser and reloads the application automatically.

Once reload has finished, press the **New Requisition** button

You should see the following:



Click the **Cancel** button to verify that it does close the popup as expected.

Add Display Elements to the Popup

We need a place for the user to type the requisition details, and this should suffice to begin with.

First, create a label (“**Description:**”), which prompts the user to describe the requisition:

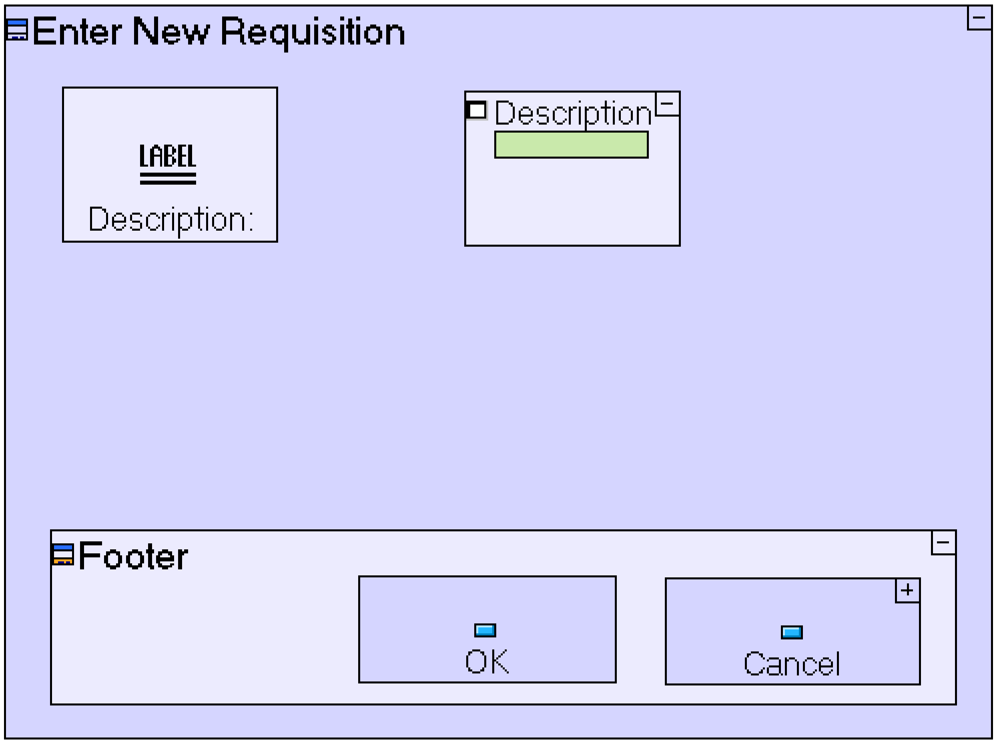
Double-click the **Enter New Requisition** popup to zoom into it.

Select the **Display/Label** template (LABEL) and drop it into the upper part of the popup. Name it ***Description:*** (the colon being part of the name).

Next, insert a text area element in which the user can type the description of his/her requisition:

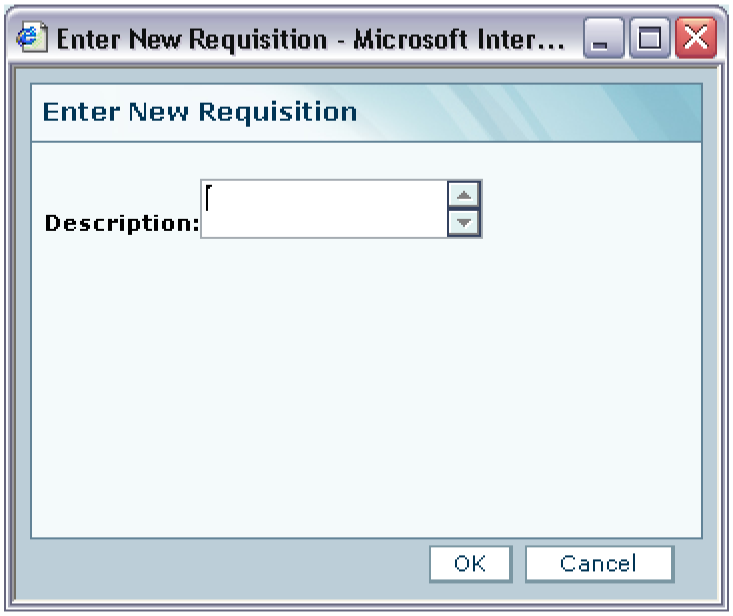
Select the **Display/Text Area** template (☐) and drop it to below the label. Name it ***Description***.

Your model should now look similar to the following:



Save your work, and switch back to the browser.

Your popup should look as follows:



Note that the order by which display elements appear in the form is determined by the relative positions of the corresponding sub-models in the popup's model. In general, the order is governed by a top-to-bottom, left-to-right rule (as if reading English).

More generally, the way display elements are arranged in the model hierarchy defines how they show on the screen. We have already seen two different examples:
1. Placing a popup inside a button causes the popup to appear when the button is pressed.
2. Positioning display elements next to each other (inside a parent model) defines the order in which they are displayed on the screen.

Rename a model

The **Enter New Requisition** popup contains the pre-built button, **OK**, which we shall use in the next stages to submit the new requisition. Let's rename it accordingly:

Zoom to the **OK** button in your model.

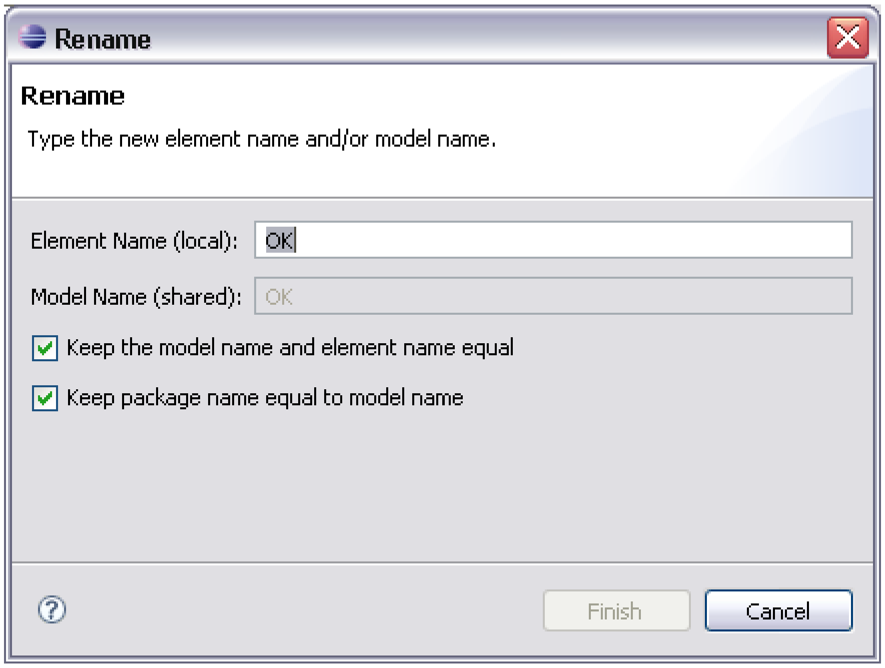
Next, either

Right-click it and select **Rename** from the menu.

or

Click to select the model, and press **[F2]**.

This will open the **Rename** dialog:


Rename

Rename

Type the new element name and/or model name.

Element Name (local): OK

Model Name (shared): OK

[✓] Keep the model name and element name equal

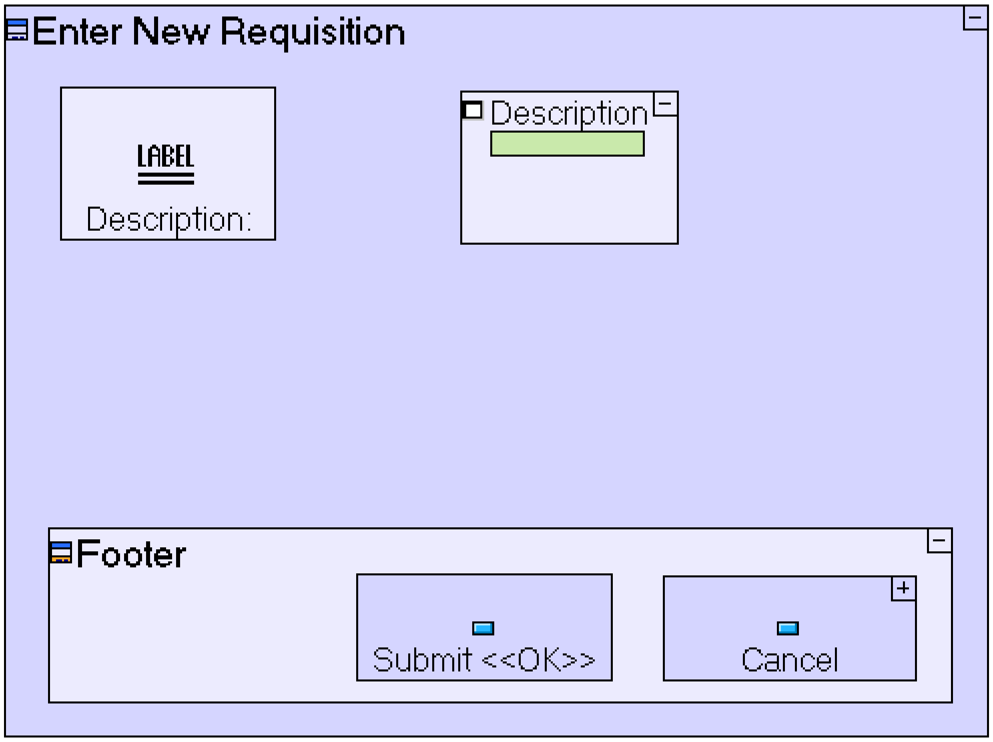
[✓] Keep package name equal to model name

Enter ***Submit*** replacing **OK**, and click **Finish**.

The **Rename** dialog is used to specify two different names for the model element you are renaming, an **Element Name** and a **Model Name**.
Model Name is the shared name of a reusable model (see **Stage 4** for further details regarding reuse). A model name must be unique in a given package.
Element Name is the local name of a model. An element name must be unique in a given parent. In the case of display models (such as the **View** we have just created), the element name is the name which will be displayed to the user.
Usually, the names are identical, and in these cases, the **Keep the model name and element name equal** check-box is checked by default so that both names are changed simultaneously.
There are however, situations in which a certain element or model name cannot be used, and in these cases you may decide to uncheck the check-box, and set different names. For further details, see the

discussion regarding automatically generated names in Stage 14.
The **Keep package name equal to model name** check box appears when the model you are renaming has the same name as the package containing it. It is checked by default, but you may clear it if you do not wish to rename the package.

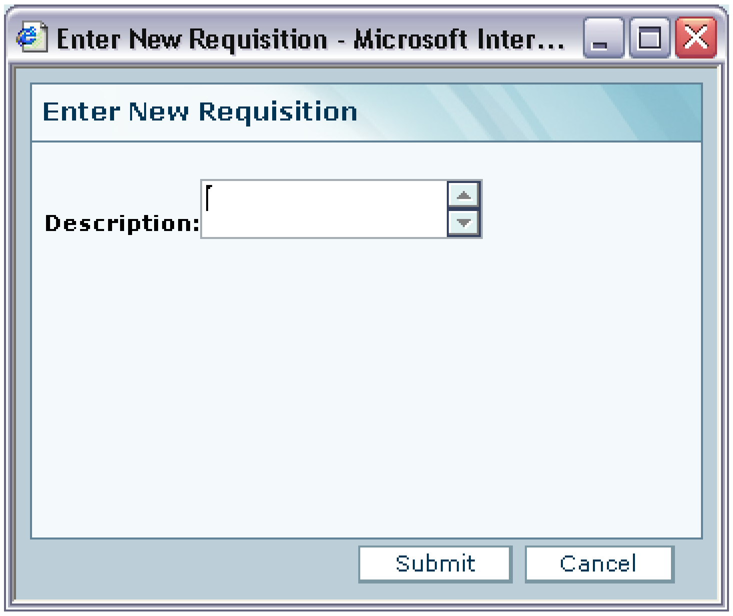
Your model should now look similar to the following:



The button we have renamed, still displays its original name in angled brackets (<<**OK**>>). This is a reminder that the button originates from the Popup template. It has no real effect, and will not be displayed at runtime, therefore you may ignore it.

Save your work, and switch back to the browser.

Your popup should look as follows:



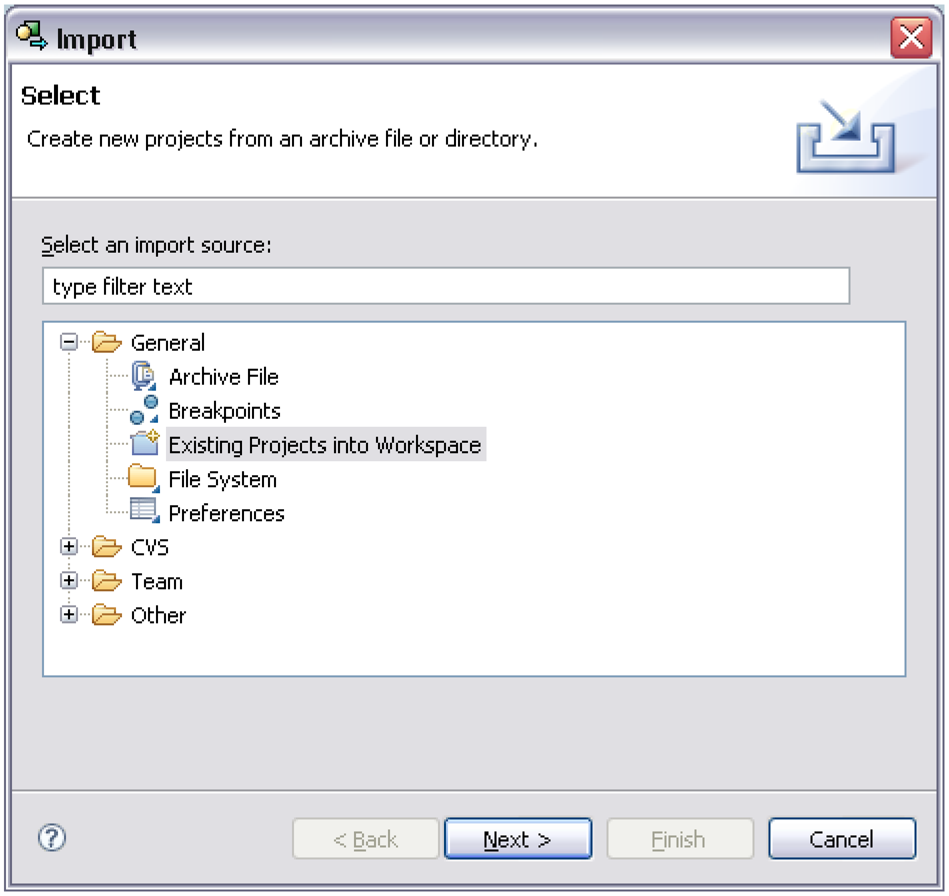
Completing Stage 2

We have completed modeling of the form, however pressing the **Submit** button does not save the data anywhere. We shall handle this in the next stage.

Importing a Sample Project

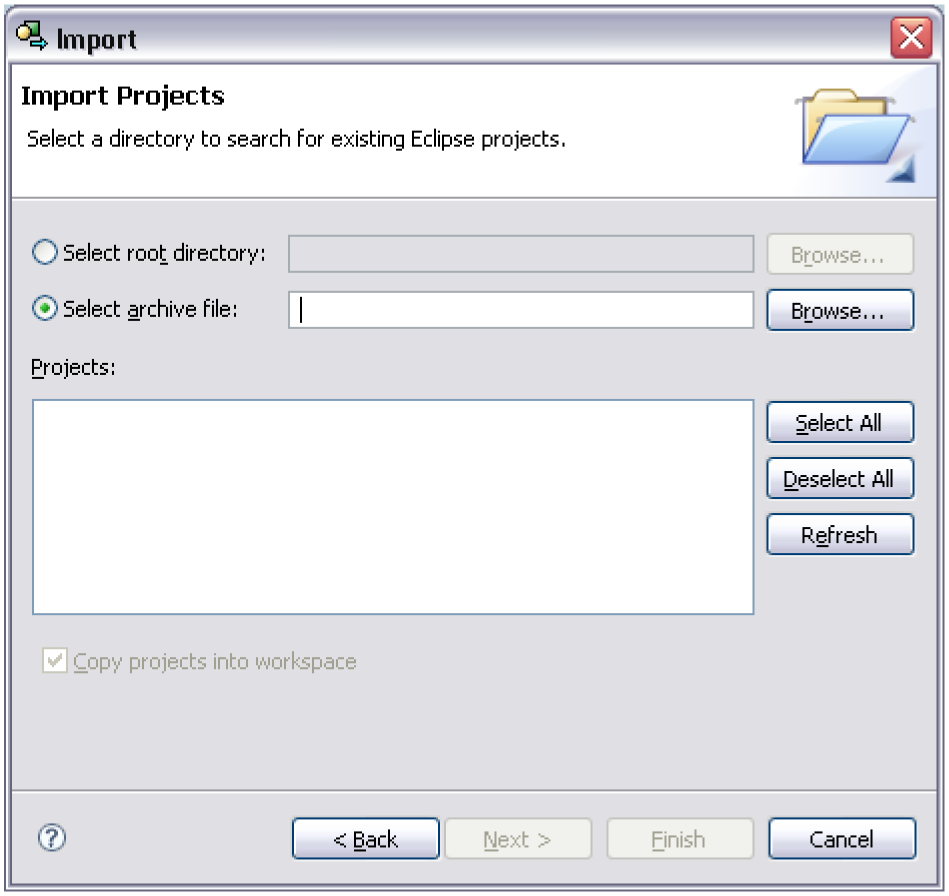
We shall now import a ready-made sample project, **Tutorial 2-3**. This sample project contains all the functionality modeled thus far, and will serve as the basis for the next stage of the tutorial:

Select **File**| **Import…** to open the **Import** wizard.

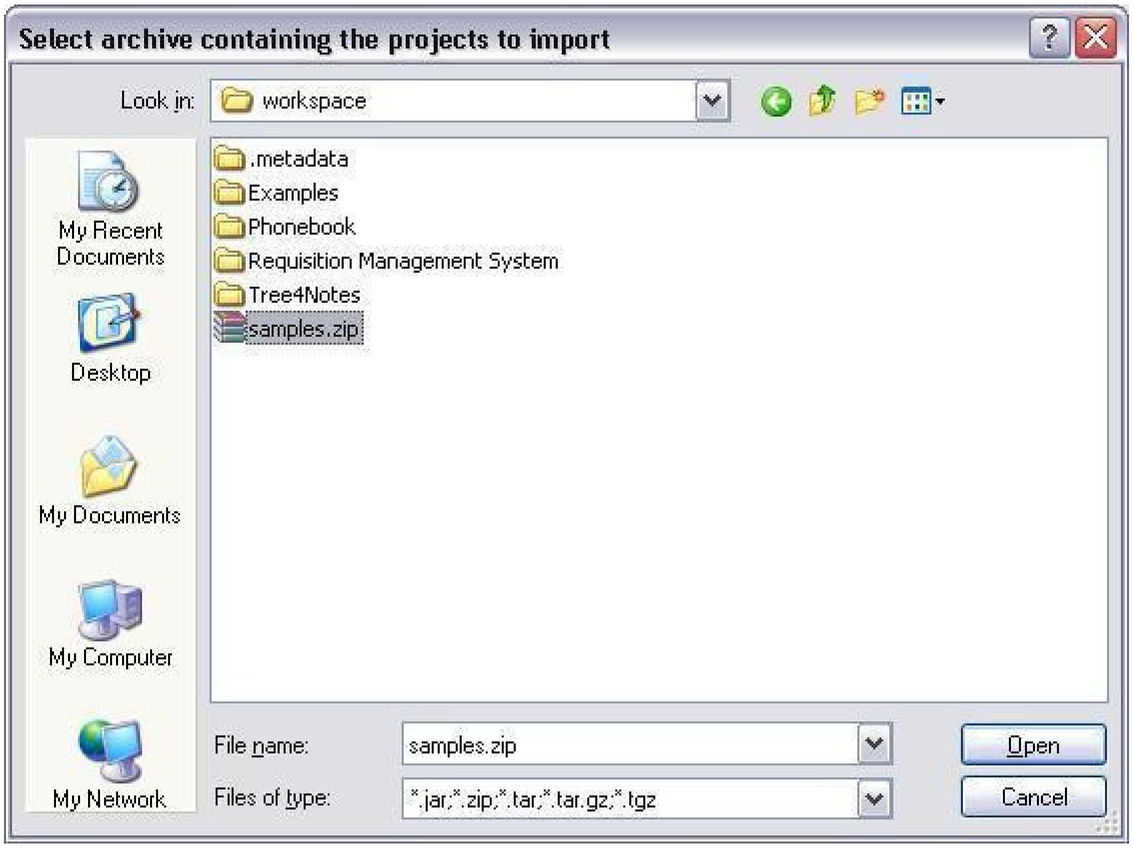


Select **Existing Project into Workspace** as the import source.

Click **Next>**.



Make sure the **Select archive file:** radio button is selected, and click the **Browse…** button to open the **Select archive containing the projects to import** dialog.



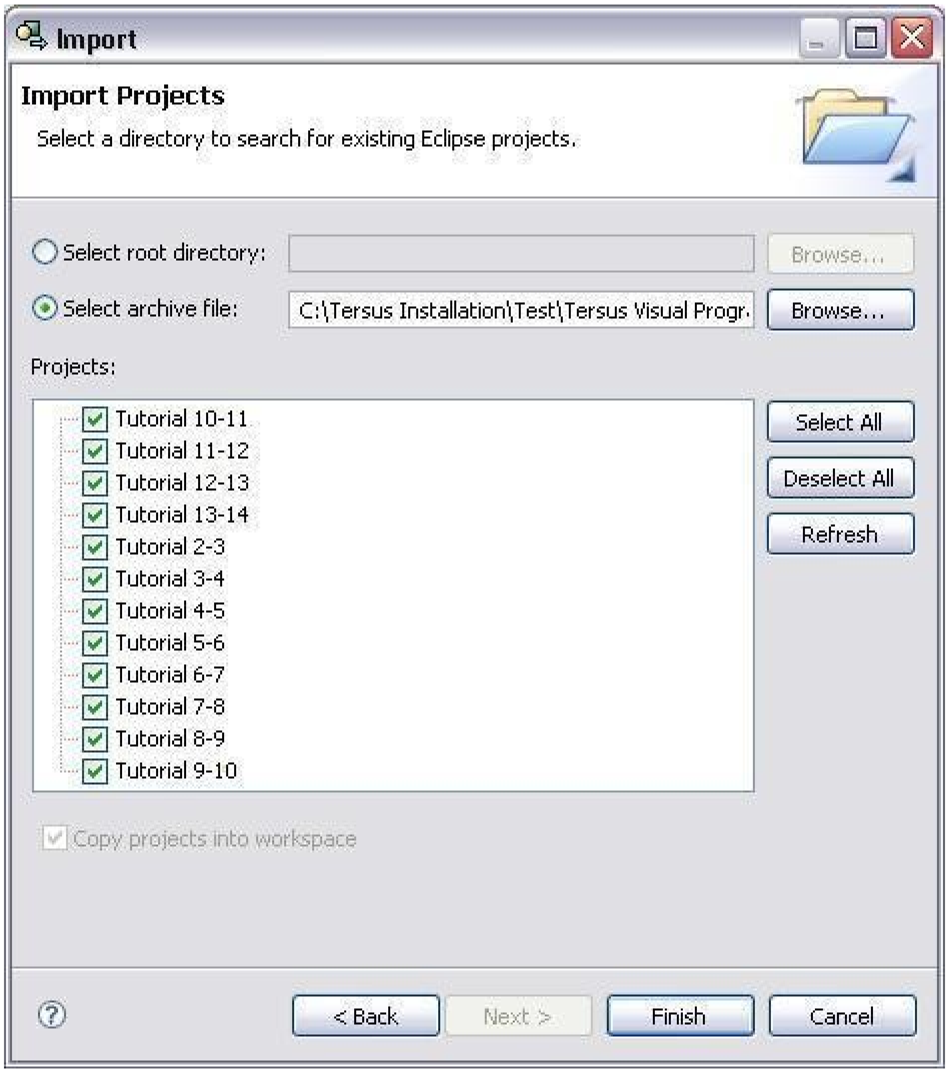
Select archive containing the projects to import

.metadata
Examples
Phonebook
Requisition Management System
Tree4Notes
samples.zip

File name: samples.zip
Files of type: *.jar;*.zip;*.tar;*.tar.gz;*.tgz

By default the dialog should open in the **workspace** folder which contains the sample project archives.

Select (or double-click) the **samples.zip** archive.



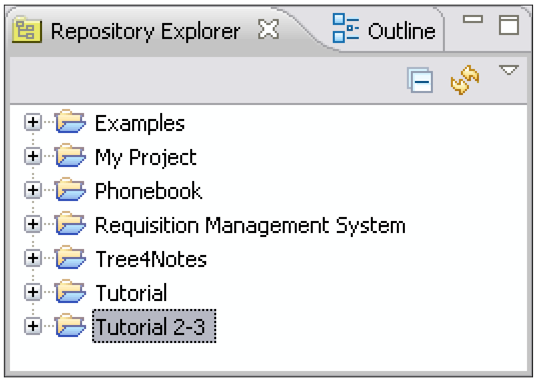
Note the following:
The **samples.zip** archive contains sample projects for all stages of the tutorial. You can import as many samples as you wish, but it is recommended that you import only the sample you need (**Tutorial 2-3** at this stage).
The full path of the archive you selected may be different, depending on the location in which you installed Tersus.

Click the **Deselect All** button.

Check the check box next to the **Tutorial 2-3** project.

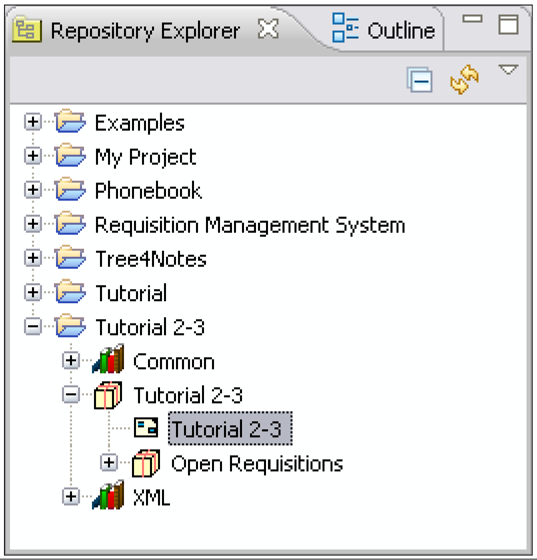
Click **Finish** to import the project.

The wizard will import the project, and add it to the **Repository Explorer**.

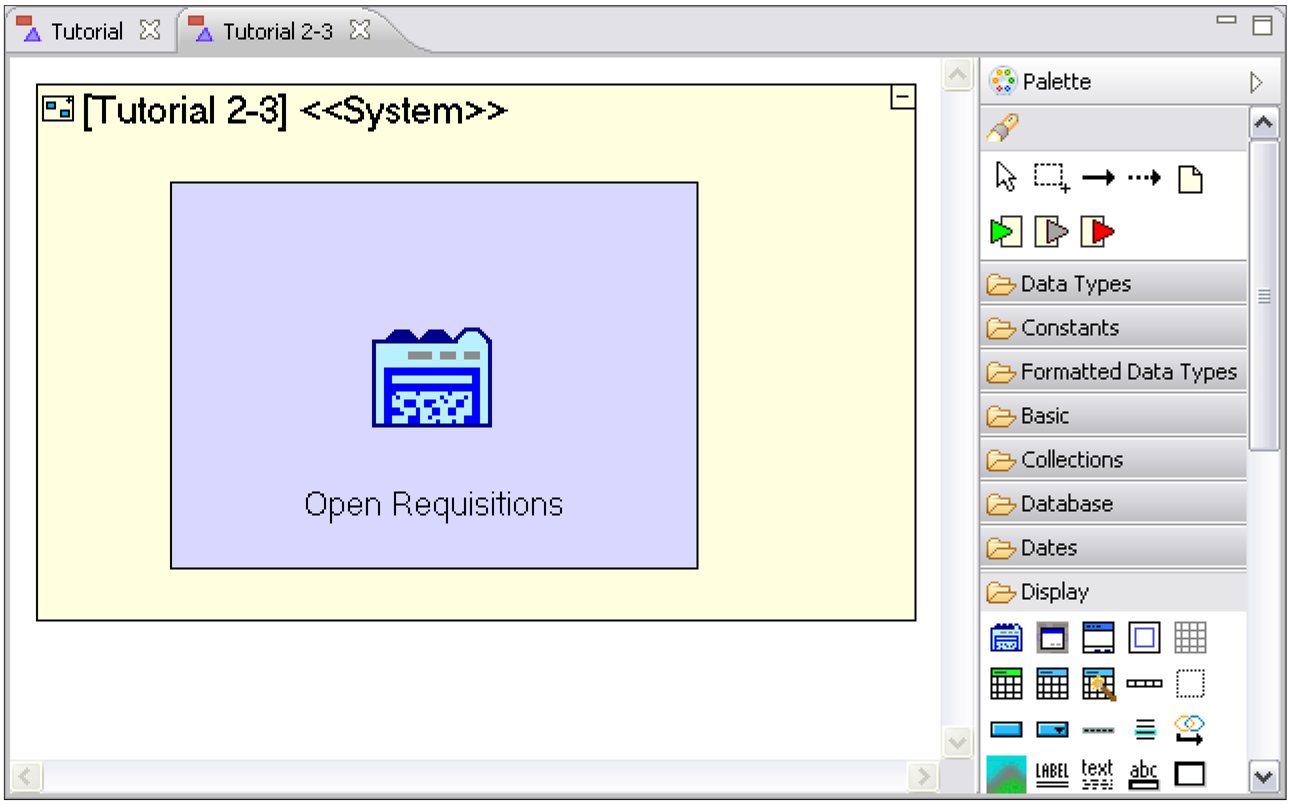


Double-click the **Tutorial 2-3** project to open its root model in the model editor.

Alternatively, expand the imported project, **Tutorial 2-3**, and the **Tutorial 2-3** package found in it, and double-click the . root model in it.



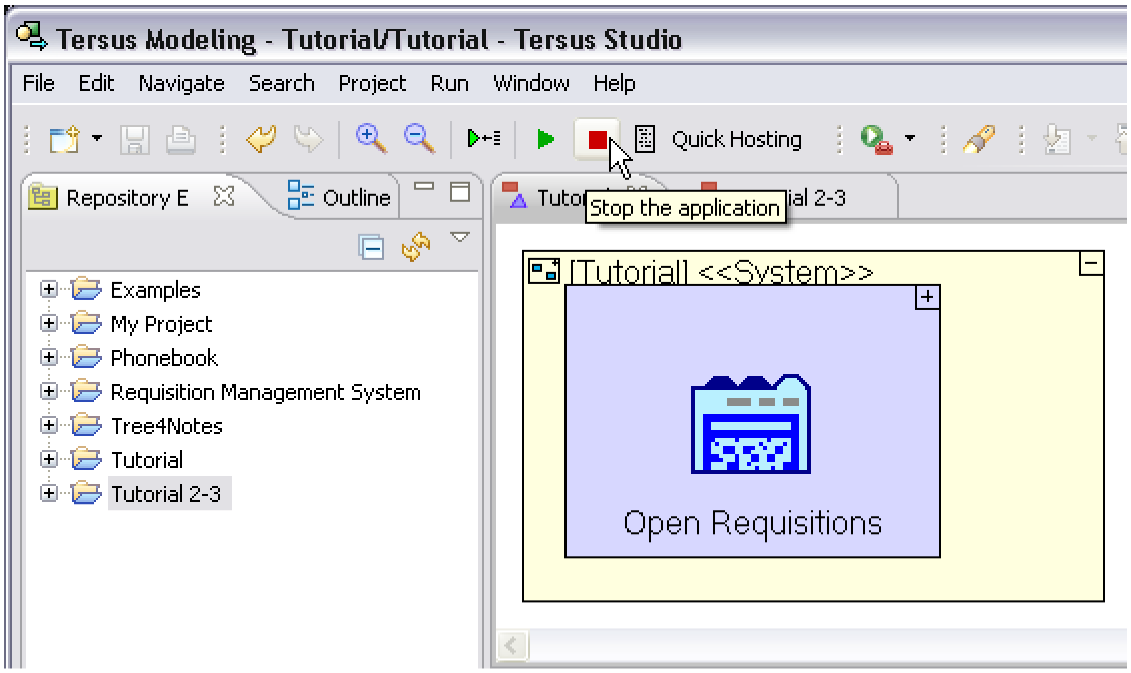
Doing this will open the model in a new model editor window.



Note that the project you previously worked on, **Tutorial**, is still open for editing in a separate model editor appearing as the left-most tab, in the screenshot above), and in the application server.

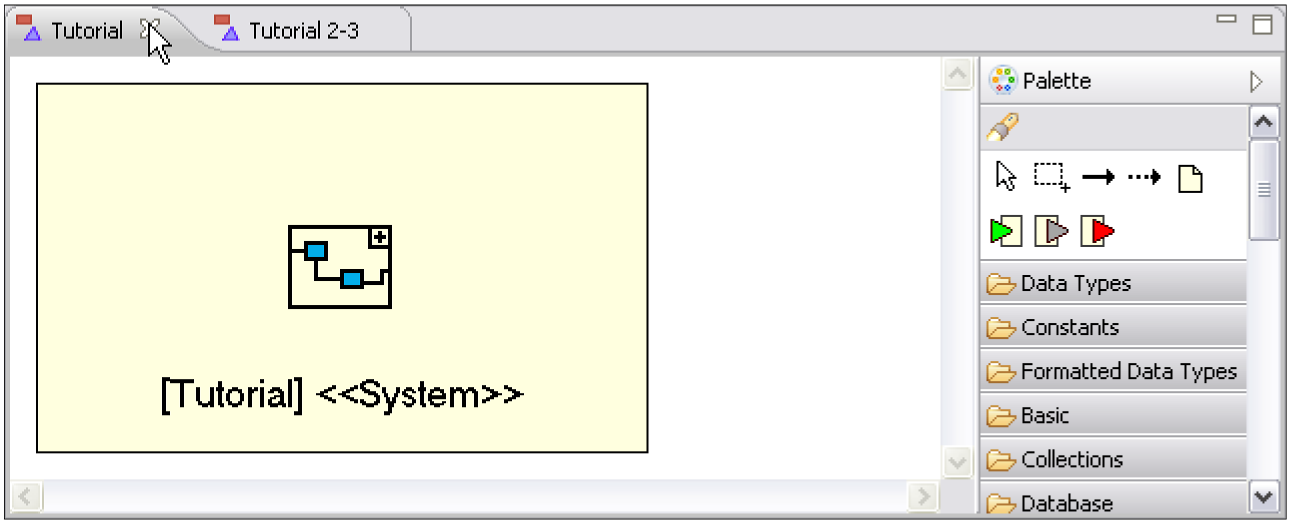
Switch back to the **Tutorial** model editor, by clicking on the **Tutorial** tab.

Click on the **Stop the application** button in the studio's main toolbar to unload your application from the embedded Tersus Server.

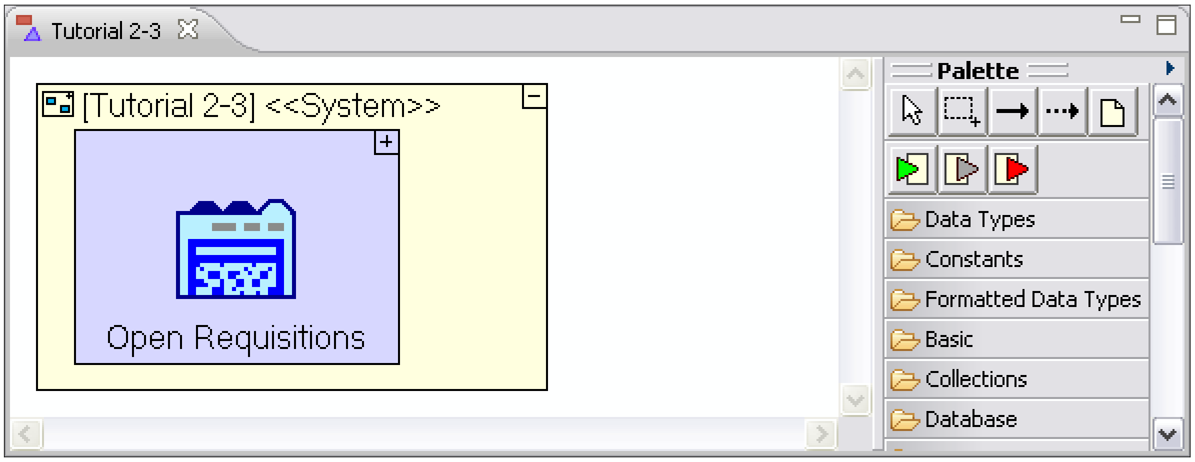


Close the browser window running the application.

Close the **Tutorial** model editor by pressing the Close button on the editor's tab:



The model editor should now look as follows:

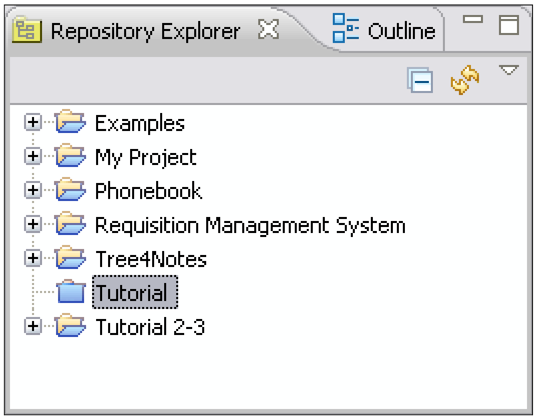


It is also recommended that you close the (old) **Tutorial** project you created at the beginning of this stage:

In the repository explorer, right-click the **Tutorial** project.

Select **Close Project** from the menu.

The repository explorer should look similar to the following:



You may now proceed to **Stage 3**, in which we will implement a process which saves the submitted requisition to the database.

See It Live

Object 1

Click here to open the live project in a separate window.

Stage 3 – Modeling the Logic behind the Screen

Stage Goals

Tersus Concepts Covered

On completion of this stage you should be familiar with the following concepts:

Modeling notions:	Data-element, Data Types, Flow, Slot, Display Data-element, Ancestor Reference, Reserved Names, Prototypes
Modeling techniques:	Retrieving user input from the display, Storing data in a database table.
Useful process templates:	Sequence, Insert

Application Functionality Modeled

In this stage you will model some basic application logic - retrieving the requisition description entered by the user (through the popup form) and saving it in a database table.

This stage's modeling should be performed in the **Tutorial 2-3** project, you imported at the end of the previous stage.

Model Application Logic

In the previous stage we learned how to model a form for simple data entry. We mentioned that one crucial thing was missing – saving the data entered by the user.

So now, let's model the saving of the requisition's description when the user clicks the **Submit** button.

As mentioned above, Tersus modeling employs a zoom-in technique in which application logic is defined "inside" the display element which invokes it. In our case, saving of the requisition's details should be performed when the button is clicked, so it will be modeled inside the **Submit** button.

Zoom into the **Submit** button by double-clicking on it in the model editor (or in the **Outline** view).

Define a Data Structure

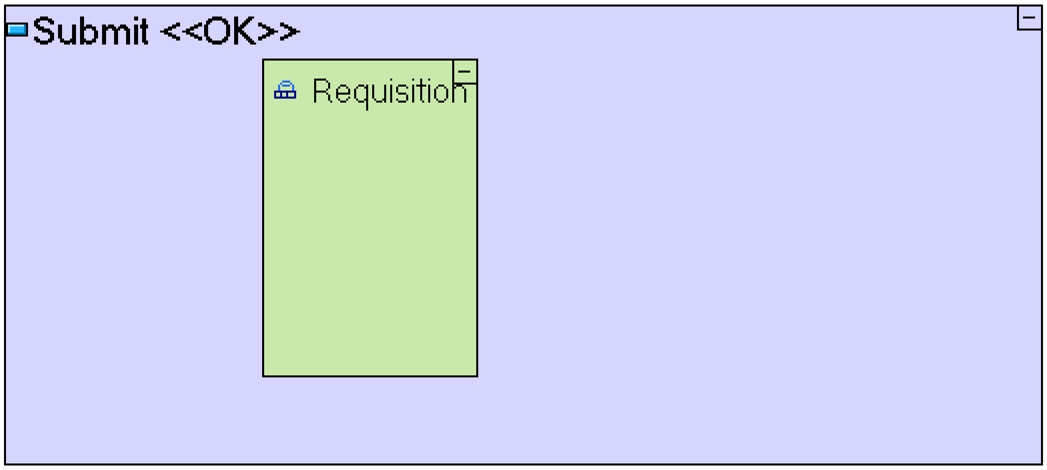
We will now define a data structure to store the details of a requisition:

Select the **Data Types/Database Record** template (🖧) and drop it into the button. Name it ***Requisition***.

A data structure is a collection of data items (either “primitives” like a number or a date, or other data structures). Data structures appear in the model as gray rectangles.

You may have noticed a **Data Structure** template appearing next to the **Database Record** template in the Palette. The two templates are practically identical, except for the fact that a **Database Record** structure is automatically associated with a matching database table, and since we would like the **Requisition** data to be stored in the database, as we will see later in this stage, we are basing it on a **Database Record**.

The **Submit** model should look similar to the following:

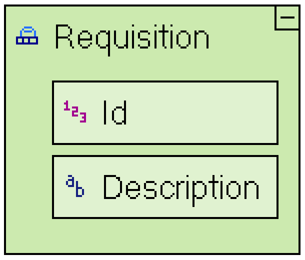


Next we should define the fields comprising the data structure. To insert data elements into the data structure, we simply drag the appropriate data types (Number, Text, etc.).

Select the **Data Types/Number** template (123) and drop it into the **Requisition** data structure. Name it ***Id***.

Select the **Data Types/Text** template (ab) and drop it into **Requisition** as well. Name it ***Description***.

The **Requisition** data structure should look similar to the following:



The Requisition data structure now contains 2 fields: **Id**, which is to be an automatically-generated unique identifier for each requisition, and **Description,** which is a free text description of the requisition as entered by the user.

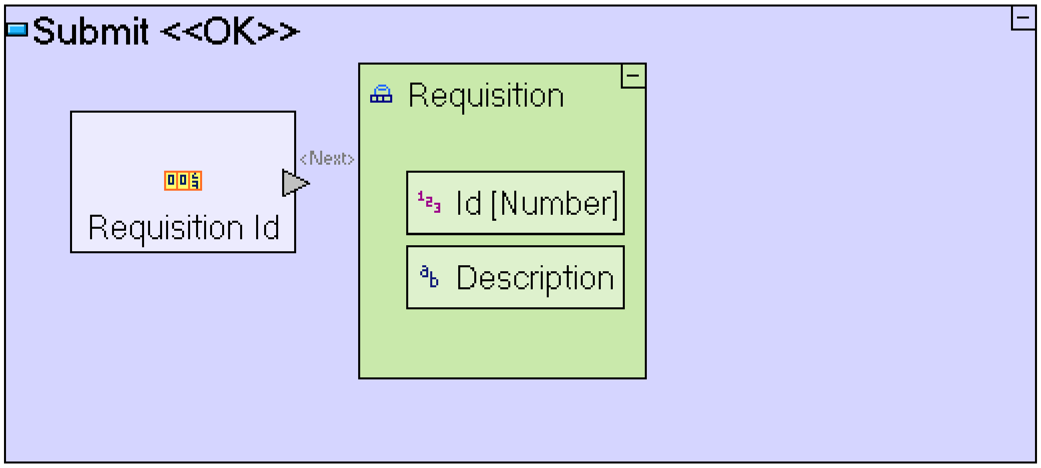
Use a Process Template (to generate a record identifier)

As explained previously, **Id** must be an automatically-generated, unique identifier. It must be unique, because we don't want any two requisitions to have the same identifier.

This can be accomplished by using the **Sequence Number** template:

> Select the **Database/Sequence Number** template ( ) and drop it into the **Submit** model. Name it ***Requisition Id***.

The model should look similar to the following:



Sequence Number is an example of a built-in process (action) that generates a new unique identifier. A process is something that is being performed.
A process can be a built-in action provided with the Tersus platform, and accessible through the palette (e.g. a basic mathematical calculation), or a more complex process modeled by a user (either you yourself or someone else).
Process modeling is the major activity required for creating a solution (a solution is comprised of display models, data models and process models).

When a process (in our case **Sequence Number**) generates some output (the unique identifier), the output needs to "exit" the process in order to be used by another process or populate a field of a data structure. An **Exit** slot is modeled as a gray triangle ( ) that appears on the frame of the process model.

There are two major types of **Slots**: **Triggers** that receive data when a process starts (discussed later), and **Exits** that expose data generated by the process while executing.

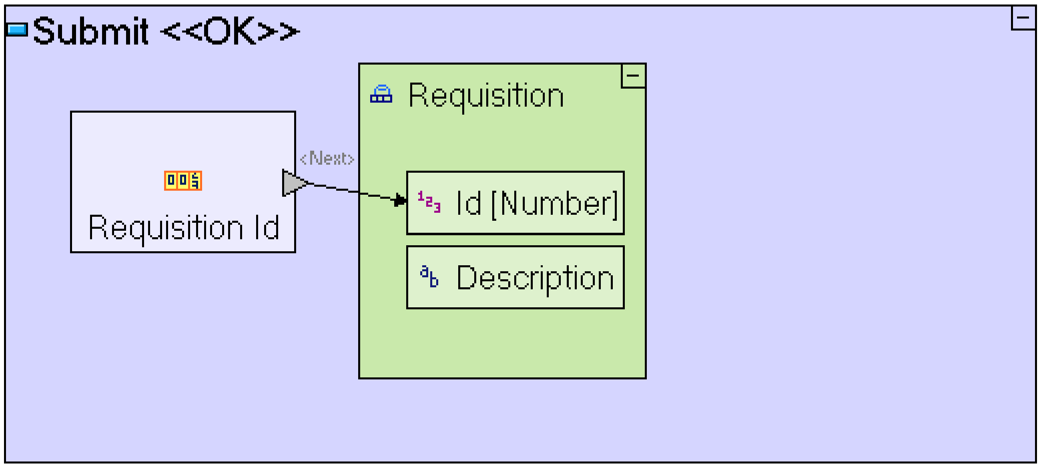
The **Sequence Number** template outputs the unique identifier it created through the predefined **<Next>** exit.

Create a Flow

Of course, we still need to pass this generated identifier from the **<Next>** exit of the **Sequence Number** process to the **Id** field of the **Requisition** data structure. To do this we shall use a **Flow**.

> Select the **Flow** tool (→) at the top row of the palette, click on the **<Next>** exit slot (▷) of **Requisition Id** to specify the **Source**, and then click on the **Id** field in **Requisition** to specify the **Target**. An arrow should appear linking the slot with the field.
>
> While using the **Flow** tool, the mouse pointer changes to signify whether the current position can serve as the **Source** or **Target** of a flow definition.

The **Submit** model should now look as follows:



A **Flow** is modeled by an arrow between two model elements (**Source** and **Target**). A **Flow** defines the relation between the **Source** and **Target**, in two ways:

Order of Execution: When should each process be executed (and depending on what conditions).

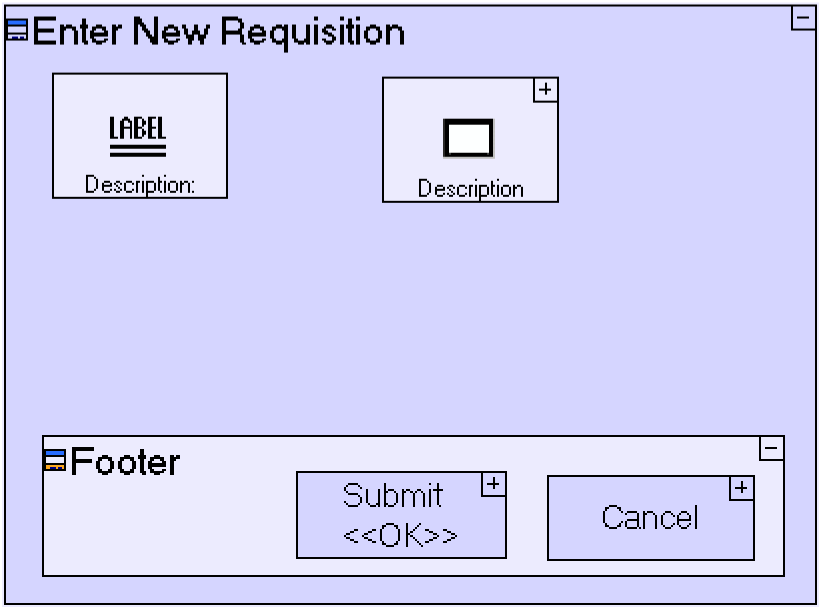
Data Flow: When and how are data items passed.

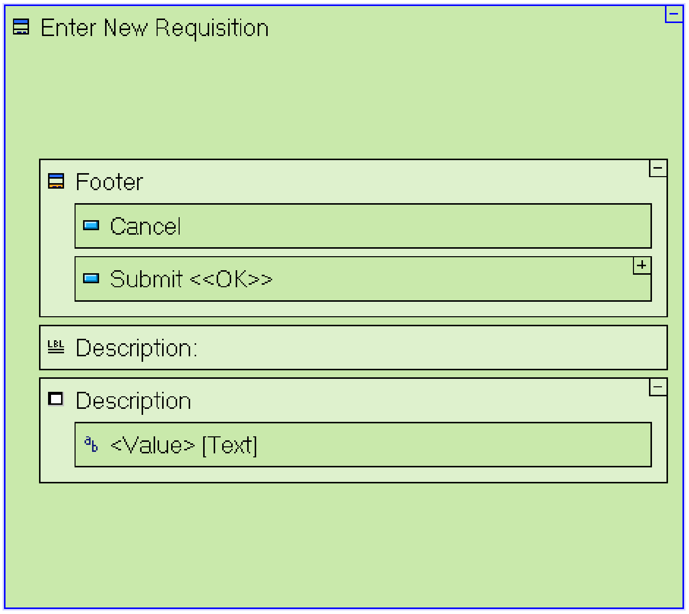
Use a Display Data Element (to retrieve user input)

Now that we've stored the automatically generated identifier in the **Requisition** data structure, we need to store the **Description** in the same data structure – after all, this is the real information we are interested in. To accomplish this we shall define a new type of element in the model called a **Display Data Element**.

A **Display Data Element** is an alternative representation of a **Display** element (and its sub-elements) as a data structure. It's main purpose is to provide access to the contents of the display so that it can be read from or written to.

For example, look at the display hierarchy of the popup we are modeling, which contains a label, a text area, a footer and 2 buttons (see screenshot of the **Display** - below left). This is represented by a data structure of the popup, which contains data structures for the text area, button and label; the text area contains another data structure for its value (see screenshot of the **Display Data Element** - below right).





The order of display elements in the Display Data Element screen shot (above right) may be different from the one you see in your model. The order of appearance is defined by the order in which the elements were actually added. As long as all the elements appear (at any order) the runtime behavior will be consistent with the tutorial.

We would like to access the contents of the **Description** text area, and so we must add to the **Submit** button a **Display Data Element** that references the **Enter New Requisition** popup. Since **Enter New Requisition** is the "father" of the **Submit** button, we use the **Add Ancestor Reference** operation:

Right-click on the **Submit** button, select **Add Ancestor Reference** from the menu, and select **Enter New Requisition**.

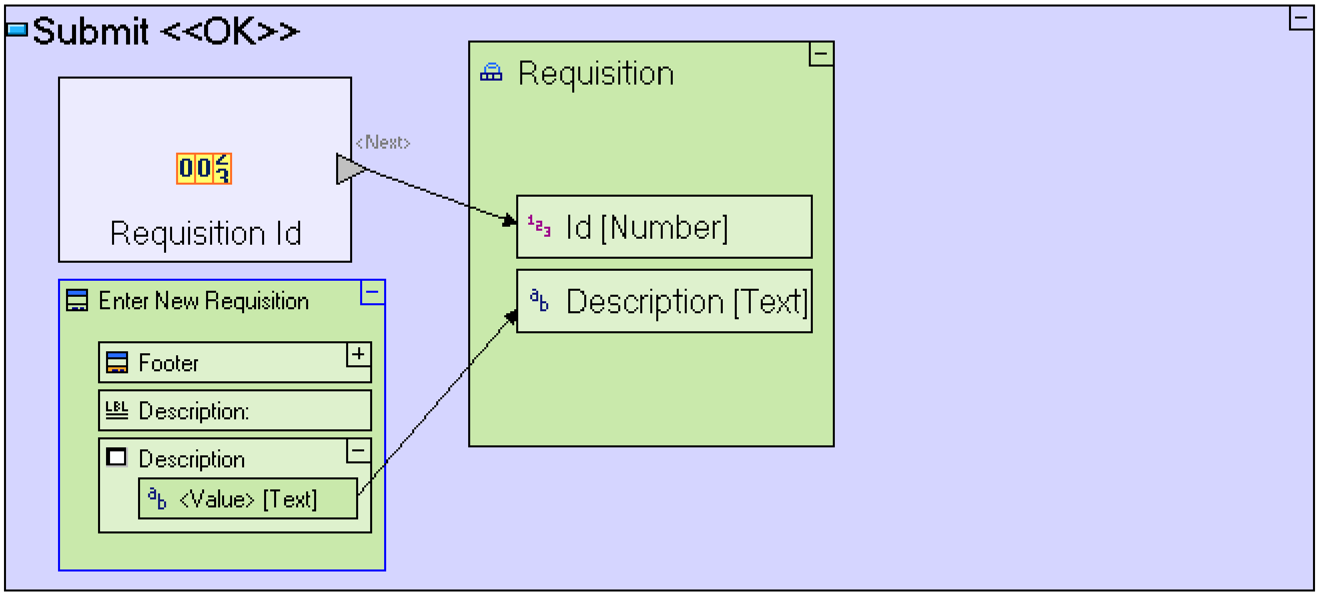
Notice that the inserted data structure has a blue frame (as opposed to the regular black frame of other elements). The blue frame indicates the fact that the data structure is a reference to another element (the **Enter New Requisition** popup), so that as data changes in one the change is automatically mirrored in the other.

Next you need to create a flow from the **Description** text area within **Enter New Requisition** to the **Description** field in the **Requisition** Data structure. The actual text entered by the user is available through the **<Value>** data element of the **Description** text area within **Enter New Requisition** data element:

Use the **Flow** tool (→) to link **Enter New Requisition/Description/<Value>** to **Requisition/Description**.

Most display elements contain default data elements through which they can be manipulated. These data elements are identified with a descriptive name (such as **Value**) surrounded by angled brackets (<**...**>).

The **Submit** model should now look as follows:



Let’s summarize what this model does so far:

The **Requisition Id** sub-process generates an identifier that is passed to the **Id** field of **Requisition**, and the content of the **Description** text area (entered by the user) is passed to the **Description** field of **Requisition**.

Use the Insert Template (to store data in the database)

Now that the **Requisition** data structure has been created and populated with the relevant data, the data has to be stored in the database. To accomplish this, we shall use another process template, **Insert**:

Select the **Database/Insert** template ( ) and drop it next to the **Requisition** data model.

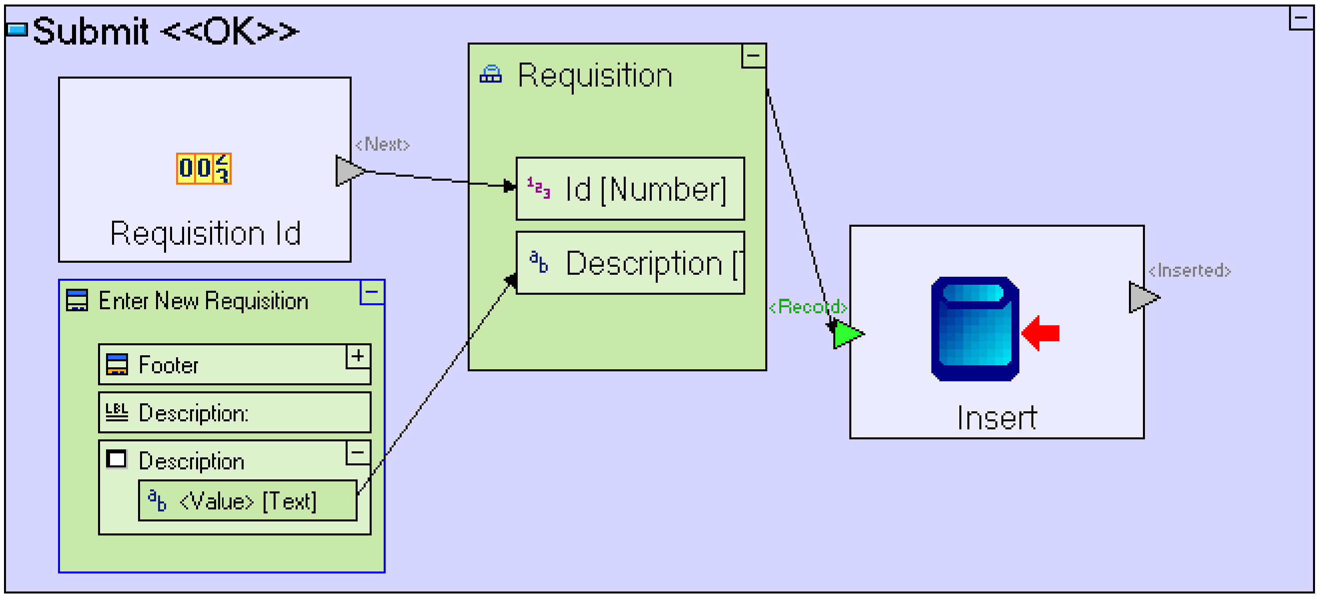
The **Insert** template includes a **<Record>** trigger slot ( ) through which it receives the data structure to be saved in the database.

Trigger slots are used as the entry point through which a process receives input data when its starts executing.

To send the **Requisition** data structure to **Insert**, let's create a flow:

Select the **Flow** tool (→), then click on the **Requisition** data structure (anywhere outside the **Id** and **Description** fields) to define it as the source of the flow, and then click on the **<Record>** trigger of the **Insert** model to define it as the target of the flow.

The **Submit** model should look as follows:



The meaning of the last change should be straight forward – the **Requisition** database record data structure is inserted as a new record to the database, into a database table with an identical name: **Requisition**. If the table does not exist in the database, it will be created automatically by the application server.

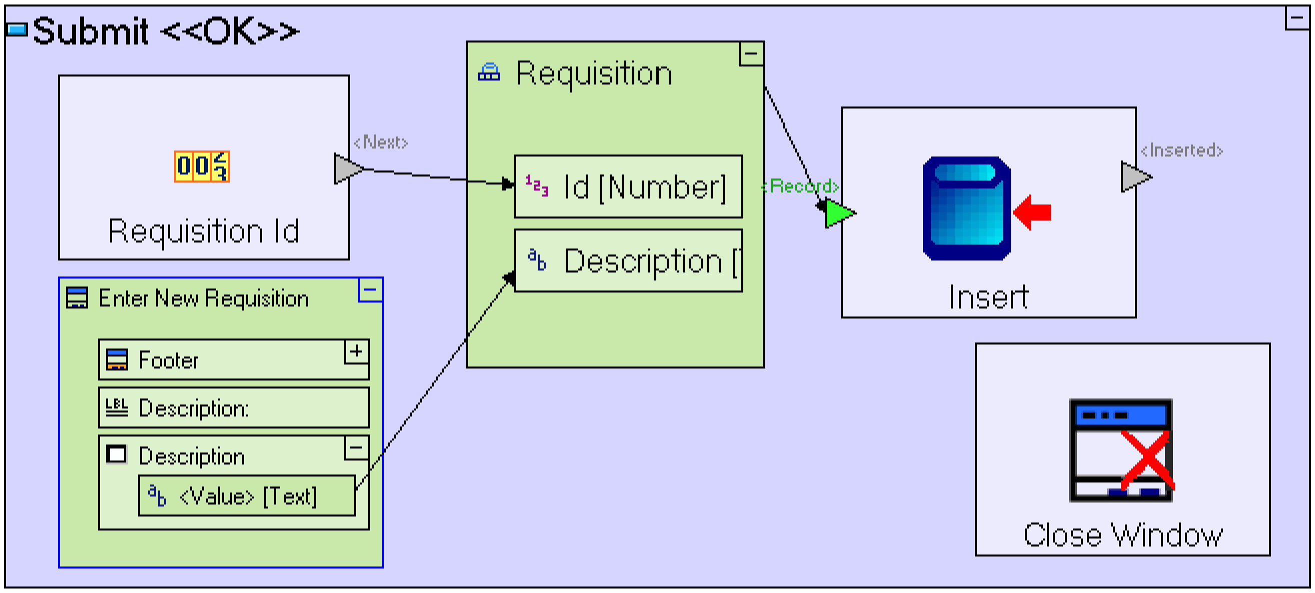
Notice that many slot names in templates (such as **Sequence** and **Insert** in the above screenshot) use a specific naming convention, where the name is surrounded by angled brackets (‘<**…**>’). These are reserved names on which template functionality may rely. If a reserved name is changed, the template may fail or function incorrectly.

Use the Close Window Template (and make sure processes occur in the right order)

Once the requisition has been written to the database, the popup window should close:

Select the **Display Actions/Close Window** template () and drop it into the **Submit** button.

The **Submit** model should look as follows:



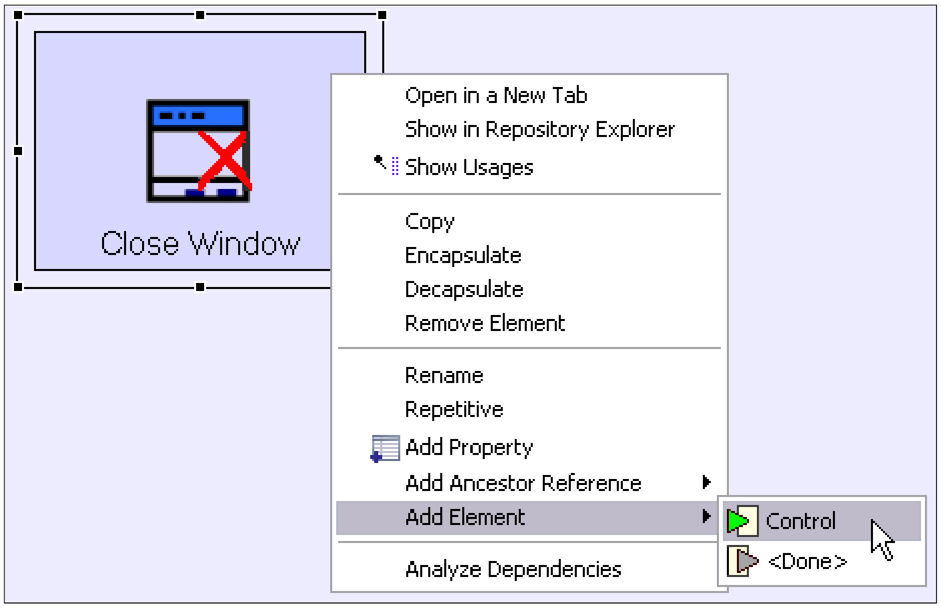
The **Submit** model has 2 parts occurring in an unspecified order, one is the **Close Window** process, and the other is the chain of processes which saves the requisition data in the database.

Sometimes it is mandatory to specify which process occurs first, because the order of processes is an essential part of the application's logic. In other cases the order in not important, but nevertheless, it is a good practice to make sure processes occur in a well defined order even when this is less critical.

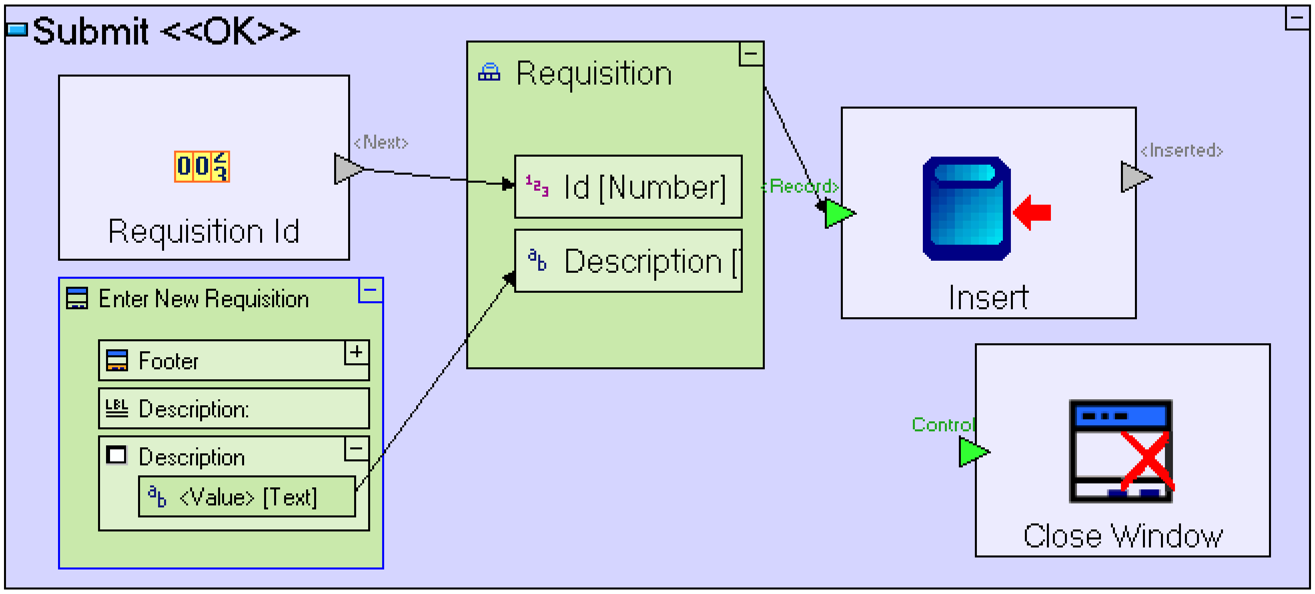
We want to make sure that the **Enter New Requisition** popup window closes only after data has been saved, therefore we should define a flow which specifies that the **Close Window** process occurs following the **Insert** process, and only if insertion was successful.

To do so, we must first add a trigger to **Close Window**. We can do so, either by adding a new trigger slot selected from the palette, or by taking advantage of the fact that **Close Window** is "hiding" a pre-defined trigger which is there for exactly that purpose:

> Right-click on the **Close Window** element, open the **Add Element** sub-menu, and select the **Control** trigger element.



The **Submit** model should look as follows:

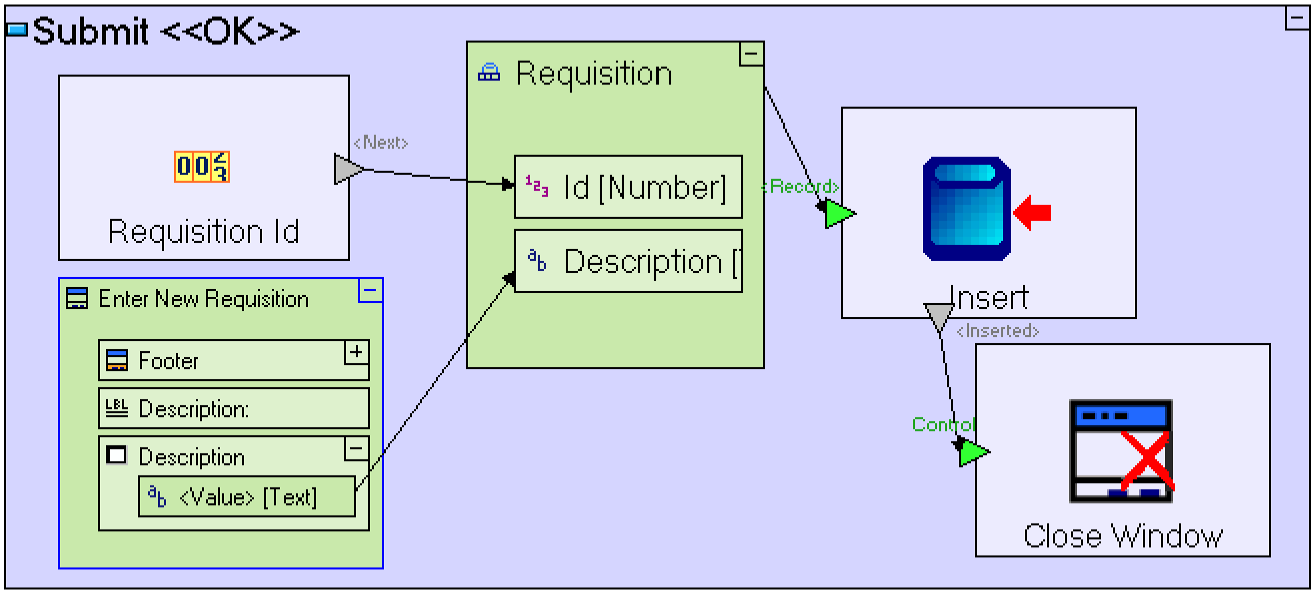


The “hidden” **Control** trigger, is available due to the fact that the **Close Window** template is implemented as a **Prototype**.
Prototypes define the elements making up a given model, and what parts of it are mandatory. In **Close Window**'s case the trigger is not required in order for **Close Window** to function (as demonstrated by the **Cancel** button in our popup), but in most use cases (such as the one we have implemented here) the trigger is needed, and so it is provided as an optional element of the **Close Window** prototype.
It should be noted that the other templates are also implemented as prototypes. For Example: **Insert**'s prototype includes a “hidden” exit, **<Duplicate>**, which is useful in cases where the record to be inserted has a duplicate key, in our case, if the **Requisition** data structure contains an **Id** which already exists in the database. But as this will never occur in our model (we are generating a unique **Id** using the **Sequence Number** template), the **<Duplicate>** exit may remain “hidden”.
It is important to point out that **all** elements available through the **Add Element** sub-menu are there for 'shortcut' purposes only. Each element can actually be created manually (in the specific case that is by dragging a trigger from the palette onto **Close Window**, and naming it **Control**), and will work just as well.

We want to close the window only after the record has been inserted to the database, i.e. when **Insert** is successful and exits through the **<Inserted>** exit:

Select the **Flow** tool to link the **<Inserted>** exit of **Insert** to the **Control** trigger of **Close Window**.

The **Submit** model should look as follows:



You may have noticed that the **Insert/<Inserted>** trigger in the preceding screenshot has been moved from its default position. This has been done to make the model more "readable" to the modeler, and has no effect on the model's functionality.

Save your work.

To view the application in the browser:

Click the ▶ **Launch the application** toolbar button.

Try entering requisitions.

At this stage we still cannot see the content of the database to verify the requisitions have been successfully stored in it. We will model this later.

If you wish, you may use an external tool (usually provided by the DBMS vendor) to verify that the **Requisition** table has been created in the database and to view its content.

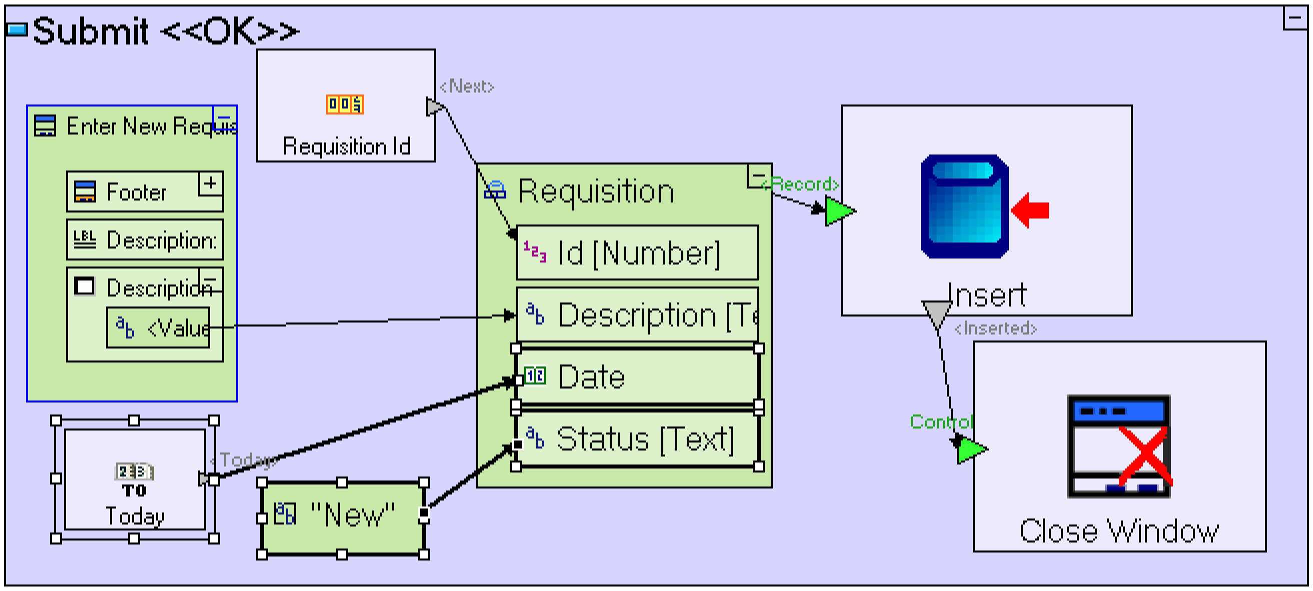
Completing Stage 3

Import the sample project **Tutorial 3-4** and use it as the basis for the next stage of the tutorial.

For a reminder on how to import a sample project, see the **Importing a Sample Project** section at the end of **Stage 2**.

This sample project contains all the functionality modeled thus far.

The sample project also includes additional functionality, as follows:



Functionality	How to Model	Located in
Add fields to the **Requisitions** table for future use	Add a **Date** field (drag the **Data Types/Date** template) Add a **Status** field (drag the **Data Types/Text** template)	**Requisition** data structure
Set a default value (today's date) to the requisition's **Date** field	Drag the **Dates/Today** template Define a flow from **Today/<Today>** to **Requisition/Date**	**Submit** button
Set a default value (**New**) to the requisition's Status field	Drag the **Constants/Text** template Name it ***New*** Define a flow from "**New**" to **Requisition/Status**	**Submit** button

We have completed the modeling of the logic performed behind the scenes, retrieving user input and saving it in the database.

You may now proceed to **Stage 4**, in which we will implement a table display of all the requisitions stored in the **Requisitions** table.

See It Live

Object 2

Click here to open the live project in a separate window.

Stage 4 – Modeling a Simple Table Display

Stage Goals

Tersus Concepts Covered

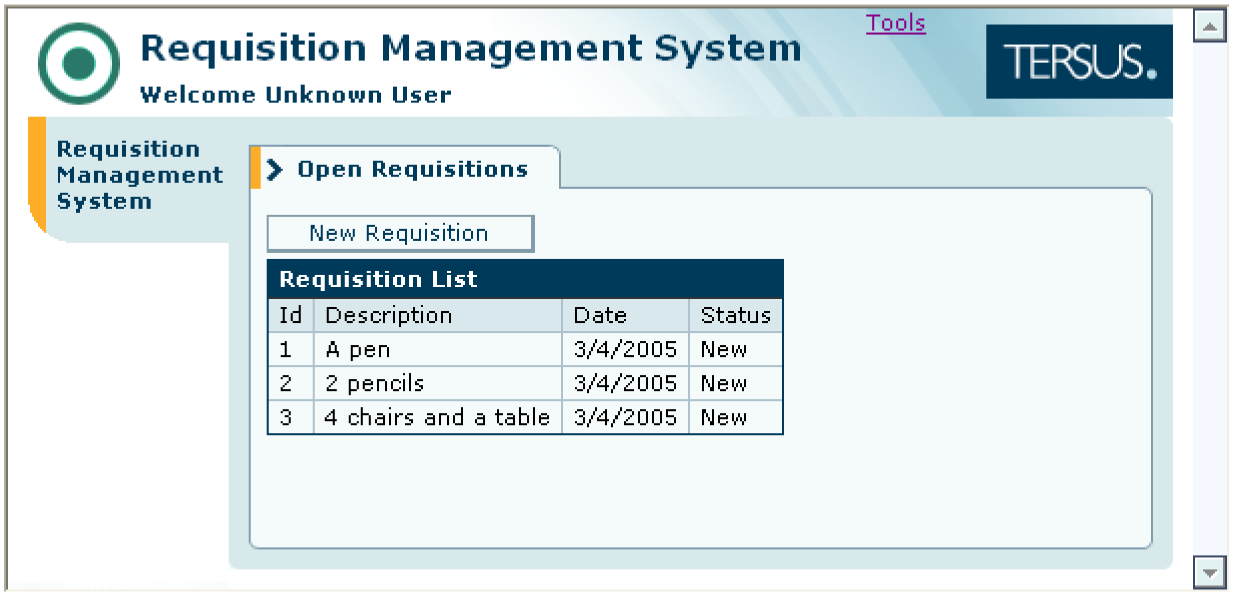
On completion of this stage you should be familiar with the following concepts:

Modeling notions:	Reuse, Process
Modeling techniques:	Retrieve and display data from the database in a table
Useful process templates:	Action, Find
Useful display templates:	Simple Table

Application Functionality Modeled

In this stage you will model the display of a list of requisitions retrieved from the database.

The resulting web application will look as follows:



Requisition List

Id	Description	Date	Status
1	A pen	3/4/2005	New
2	2 pencils	3/4/2005	New
3	4 chairs and a table	3/4/2005	New

This stage's modeling should be performed in the **Tutorial 3-4** project, you imported at the end of the previous stage.

Model a Tabular Display of Data

In the previous stages we have learned how to display a form used for data entry and store the entered data in a database.

Next we would like to be able to view the data previously entered, in a tabular manner – one row per record.

The table will be part of the **Open Requisitions** view we have created previously.

Zoom to the **Open Requisitions** view by either:

Double-clicking on it in the editor

or

Locating it in the outline and double-clicking on it there

The table will be located below the New Requisition button, so make sure there's room for the new model we're about to create in the **Open Requisitions** model:

Resize the **New Requisition** button and position it near the top of the **Open Requisitions** view.

Create a Table Display

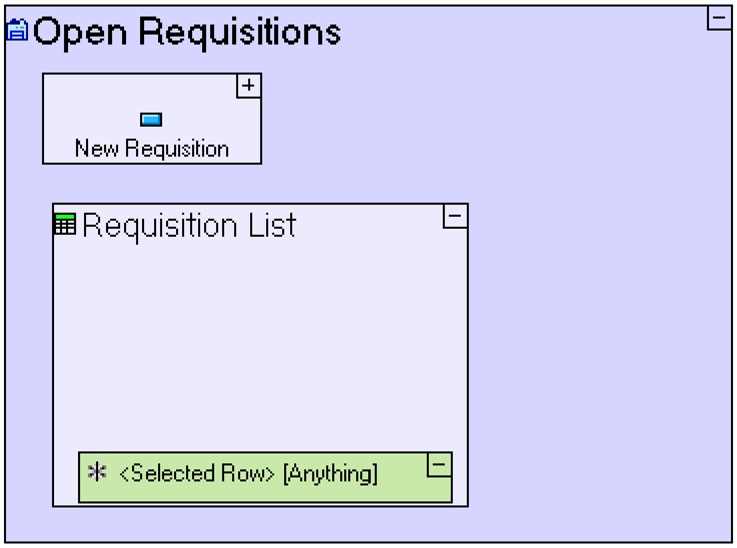
We shall start of by adding a **Table Display** to **Open Requisitions**.

Select the **Display/Simple Table** template (▦) from the palette, and insert it into **Open Requisitions**.

Name it ***Requisition List***.

The **Simple Table** template includes by default, a **<Selected Row>** element. We will ignore this element for the time being, but will make use of it in **Stage 6** of the tutorial.

The **Open Requisitions** view model should now look similar to the following:

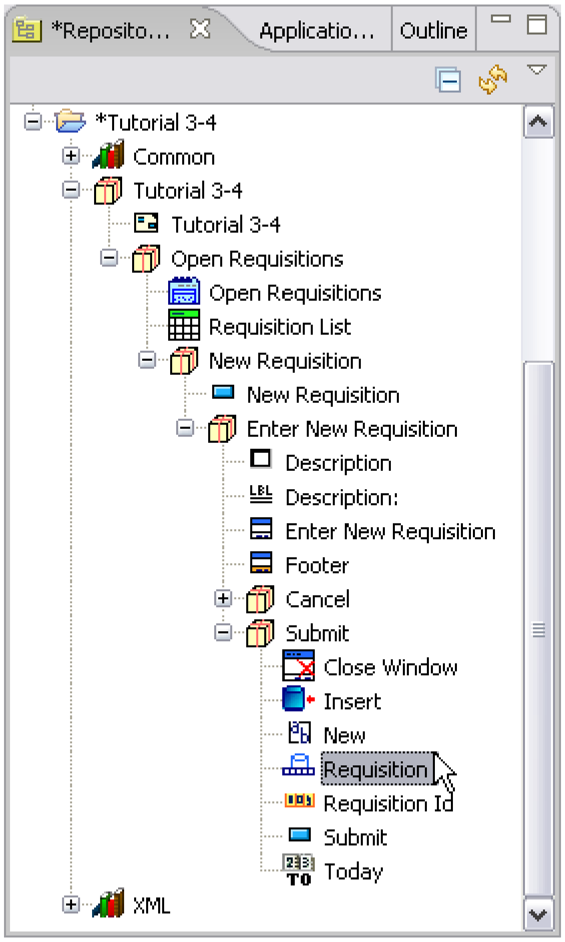


Reuse a Data Structure to Define Table Contents

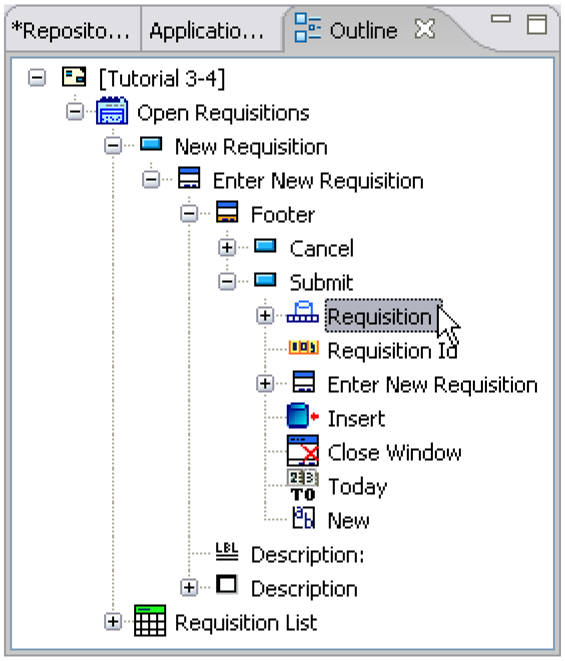
We would like the table to be as straightforward as possible and show all fields of **Requisition** (**Id**, **Description**, **Date**, **Status**), so we can simply **reuse** the **Requisition** data structure:

Reusing previously defined models is an integral part of the modeling process, and significantly improves both development speed and model clarity.
When you reuse a model, any change made to it affects all its occurrences. This ensures model consistency.
You can reuse any type of model (data model, display model or process model).

Locate the **Requisition** data structure in the **Repository** view.

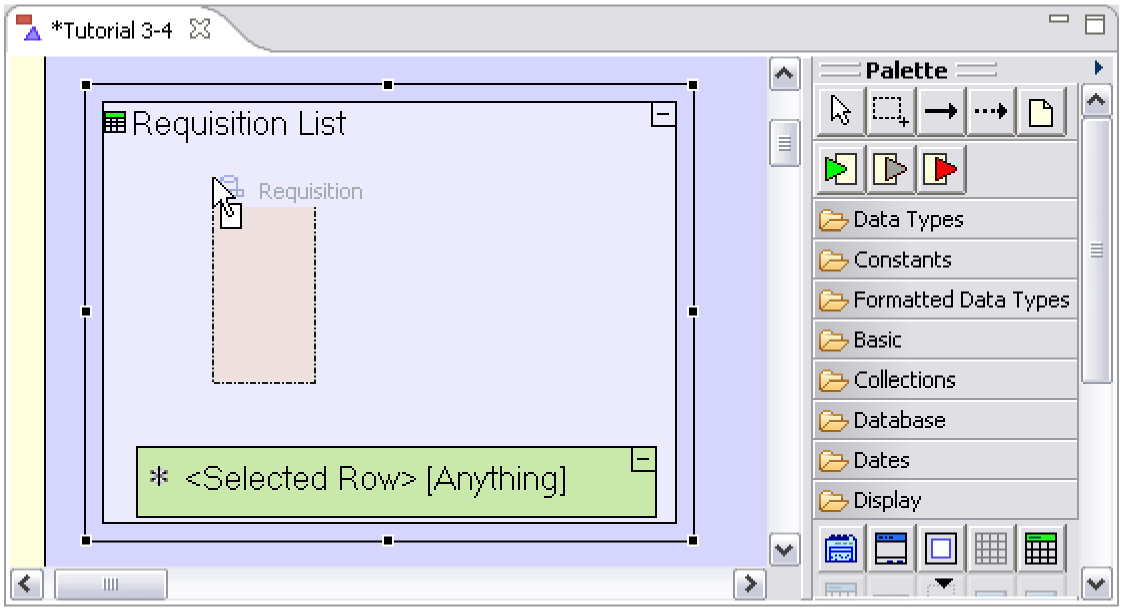


Or, in the **Outline** view.

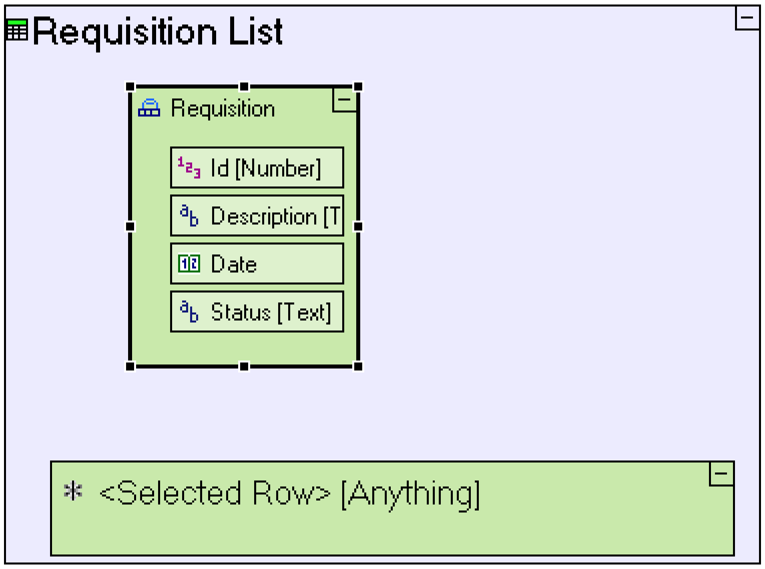


Zoom into the **Requisition List** table.

Drag it into the model editor.



And drop it inside the **Requisition List** element.

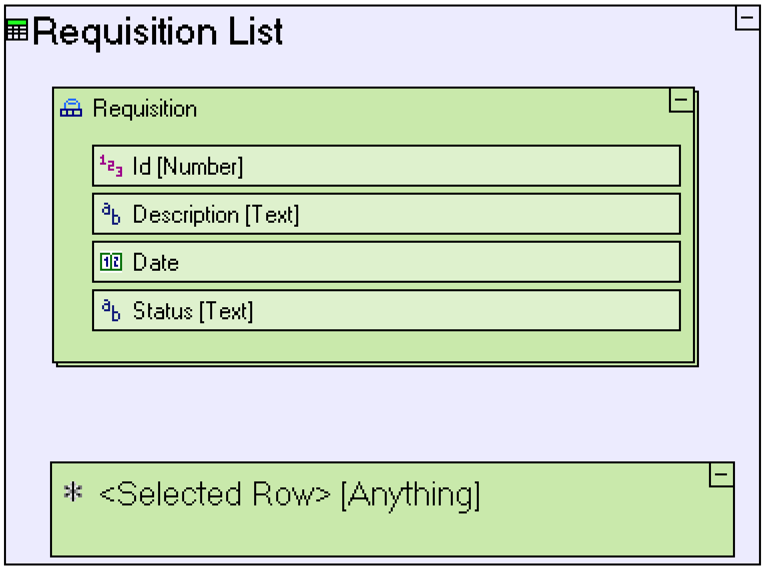


Now, since the table contains multiple rows, each repeating the same data structure (**Requisition**), we need to define the data structure as repetitive:

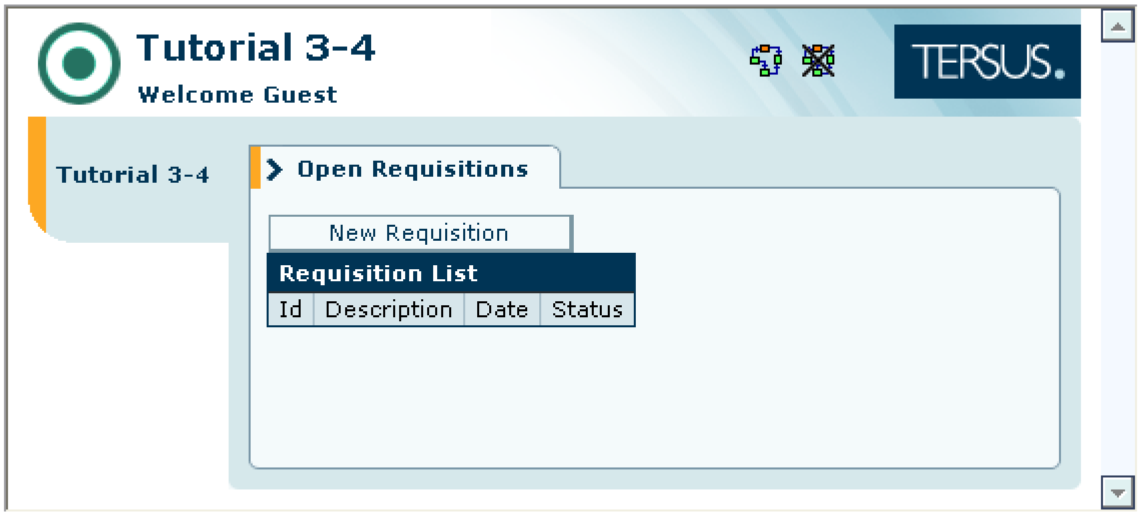
Right-click on the **Requisition** element we have just created, and select **repetitive** from the menu.

You can verify that the **Requisition** element has been defined as repetitive by:
1. Right-clicking on it again, and verifying that **repetitive** has a check mark next to it.
2. Checking the **repetitive** property in the **Properties** pane.
3. Verifying that the **Requisition** element in the model editor, appears stacked. (see in the screenshot below)

The **Requisition List** table model should now look similar to the following:



Save your work and view the application in the browser. It should look similar to the following:



Requisition List			
Id	Description	Date	Status

Notice that the **Requisition List** table is empty – the column titles appear, but there are no rows in the table. We still need to model the retrieval of data from the database to populate the rows of the **Requisition List** table.

Use Action models to Define a Process (populating the display table with data from the database)

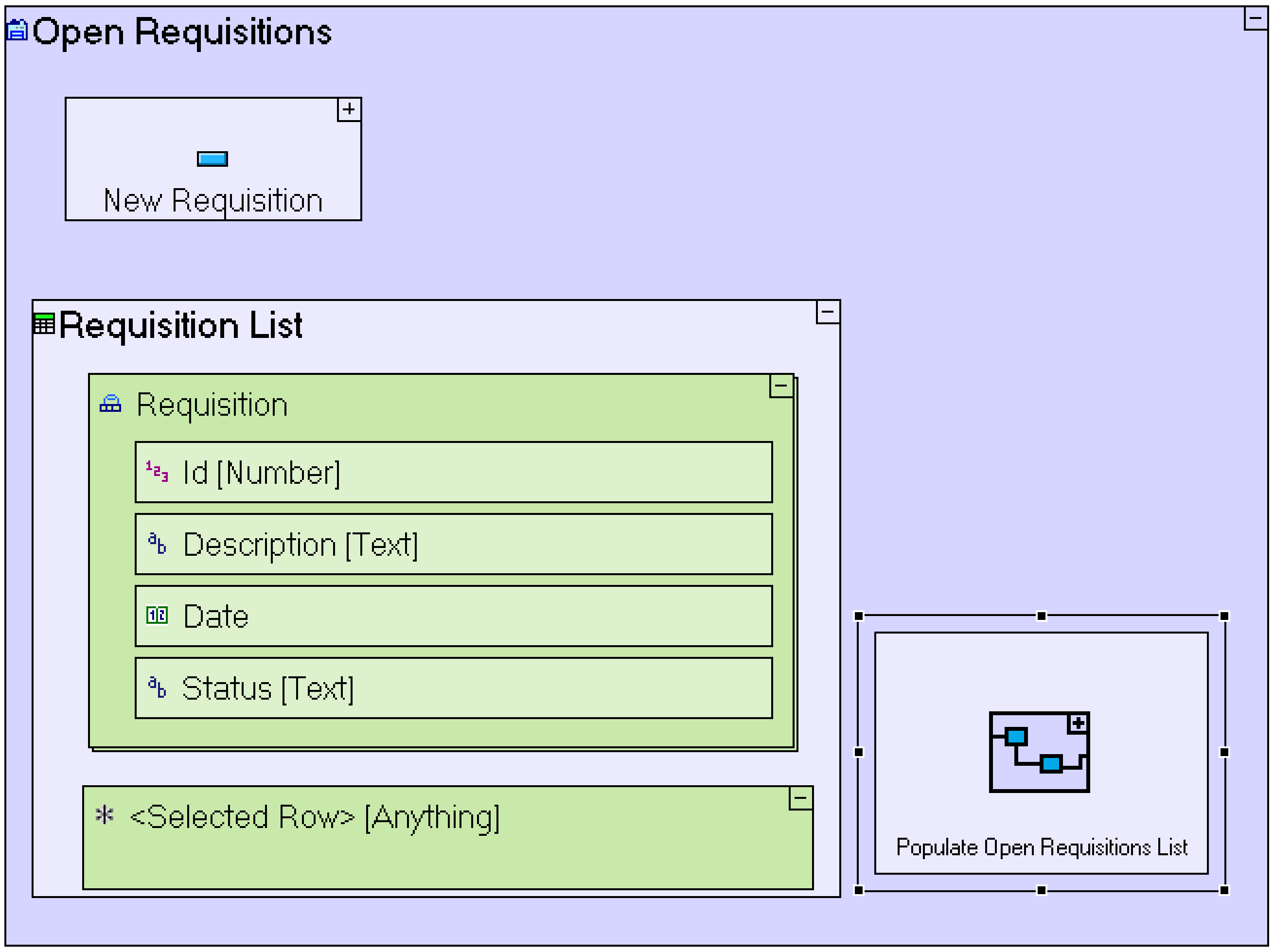
We shall now model retrieving data from the database and populating the table with it:

Select the **Basic/Action** template (), and drag it into the **Open Requisitions** model.

Name it ***Populate Open Requisitions List***.

The **Action** template is a container for defining composite processes.
Since the **Populate Open Requisitions List** has no trigger defined, it will be executed automatically when its parent (**Open Requisitions**) is executed.
The name **Populate Open Requisitions List** was chosen because it is self explanatory. It has **no** effect at runtime, and you can decide on any other name with no change to application behavior.

The **Open Requisitions** view model should now look as follows:



Notice that **Populate Open Requisitions List** is not a display model, but rather a process model. It is not part of the display of the table (as is the **Requisition** element of **Requisition List**), but rather some processing that takes place in order for the table to be properly displayed.
As explained above, it has no triggers and hence it is executed automatically, so we call it an Initialization Process. Any display model may contain such Initialization Processes.

The **Populate Open Requisitions List** process will generate a **Requisition List** data element (made up of **Requisitions** read from the database). The generated requisition list will then be sent to the display.

Define a Sub-Process (generating the table data element in memory)

To generate the **Requisition List** data element, we shall create a sub-process to the **Populate Open Requisitions List** process:

Zoom into **Populate Open Requisitions List**.

Select the **Basic/Action** template (), and drag it into the **Populate Open Requisitions List** model.

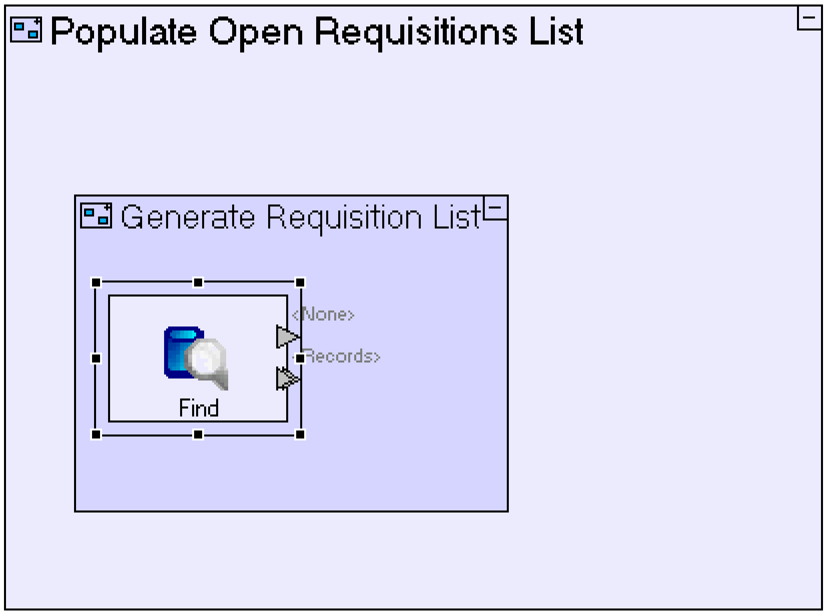
Name it ***Generate Requisition List***.

Use the Find template (to retrieve data from the database)

The **Generate Requisition List** starts by retrieving data from the database. This is performed by using the **Find** template.

> Select the **Database/Find** template ( ), and insert it into **Generate Requisition List**.

The **Populate Open Requisitions List** action model should now look as follows:



The **Find** template includes two exits, **<None>** & **<Records>**, which are "activated" (and expose output) depending on whether **Find** finds any records or not. If no record is found, the **<None>** exit is activated. If at least one record is found, the records are exposed through the **<Records>** exit.

If you look closely at the <**Records**> exit, you'll notice that it is marked as repetitive ( ), meaning that **Find** can output multiple records.

We still need to define the database table from which **Find Requisitions** should retrieve records. This is defined through defining the **Data Type** of its **<Records>** exit - the data structure that the exit outputs. This is defined next.

Use a Display Data Element (to define the data type & create the table data element)

The records retrieved by **Find** should be stored in the correct data structure, which in our case should be a data structure representing the **Requisition List** display we have previously created.

> Drag the **Requisition List** model from the outline, and drop it next to the **Find Requisitions** element.

This creates a **Requisition List** Display Data Element, which represents the **Requisition List** model.

Note that the **Display Data Element** we have just created is similar, but not identical to the Display Data Element we created through the **Add Ancestor Reference** function (in Stage 3).
The similarity lies in that both are data representations of display models. The difference is that through **Add Ancestor Reference**, you create a reference to the actual runtime display (providing access to the displayed data), whereas in the current situation we are creating a new display data element in memory which has the same structure as the display, but is not the actual runtime display. Another way to put it is that we have created a logical (or in-memory) display data element.

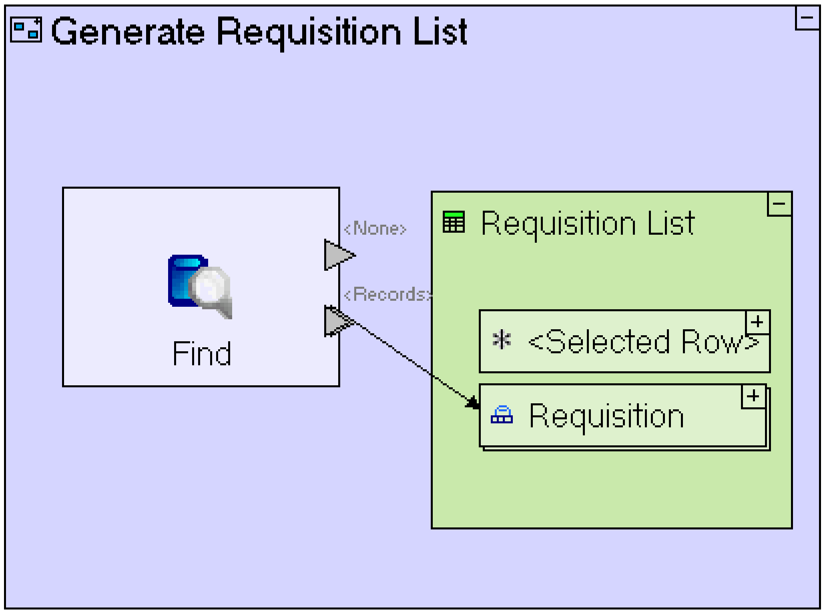
Next add **Flow** that will create the **Requisition List** table from the records provided by **Find Requisitions**:

Expand the **Requisition List** element (by clicking the ⊞ sign at its upper right corner) to display its content.

Select the **Flow** tool to link the **<Records>** trigger of **Find** to the repetitive **Requisition** data element in **Requisition List**.

By creating the above flow you have done two things:
1. Explicitly defined where the output of **Find** should go (the **Requisition** element of the **Requisition List** table);
2. Implicitly defined the data type of the **<Records>** exit of **Find**, i.e. **Requisition**. Since the **Requisition** data type is based on a **Database Record** template, **Find** knows which database table it should access (the **Requisition** table).

The **Generate Requisition List** action model should now look as follows:



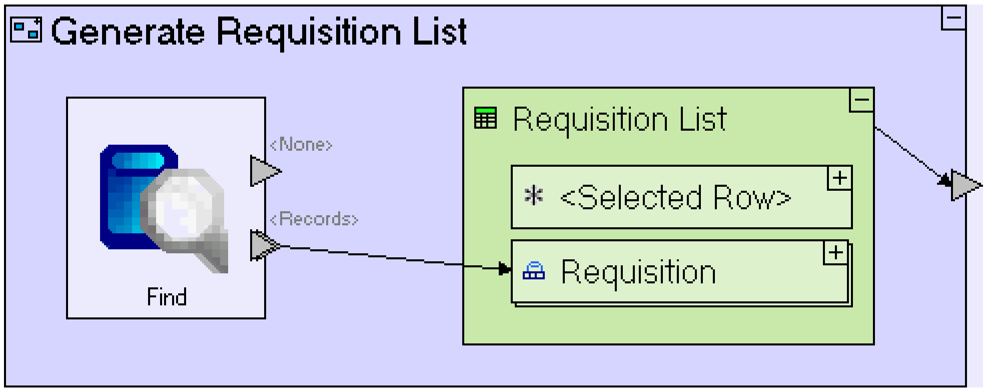
Output Data from the Sub-Process

The **Generate Requisition List** sub-process is meant to output data (the **Requisition List** display data element), so we need to define what data will be output, and where the output will be stored.

Select the **Exit** slot (▷) from the palette and click on the frame of the **Generate Requisition List** element.

Select the **Flow** tool to link the **Requisition List** display data element to the **Exit** slot we added.

The **Generate Requisition List** sub-process should look as follows:



Use a Display Data Element (to output data to the display)

In the previous stage we showed how to use an **Ancestor Reference** display data element to retrieve user input from the display. We shall now use the same technique to perform the opposite, i.e. output data to the display.

Zoom to the **Populate Open Requisitions List** element.

Right-click on the **Populate Open Requisitions List** element, select **Add Ancestor Reference** from the menu, and select **Open Requisitions**.

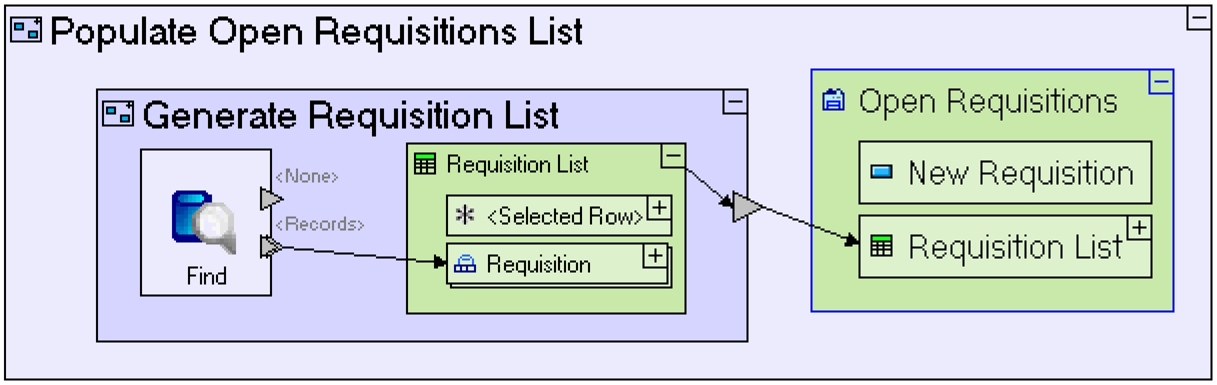
Now we can use the **Open Requisitions** ancestor reference we have created as the target to which **Generate Requisition List** will send its output table.

Double-click on the **Open Requisitions** data element to zoom into it.

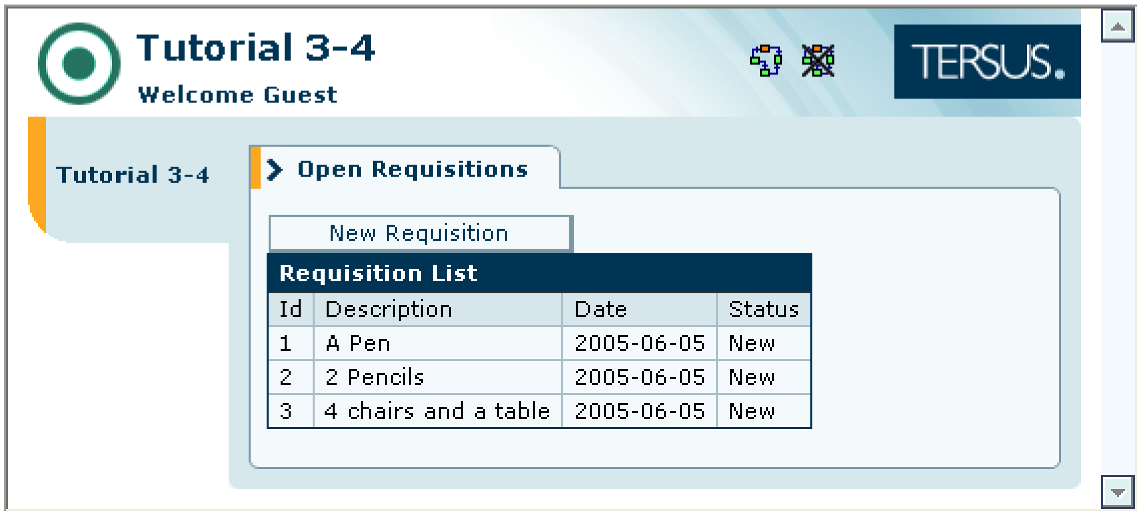
Expand the **Requisition List** element (by clicking the ⊞ sign at its upper right corner) to display its content.

Use the **Flow** tool to link the **Generate Requisition List** exit to **Open Requisitions/Requisition List**.

The **Populate Open Requisitions List** model should now look as follows:



Save your work and view the application in the browser. It should look similar to the following:



Requisition List			
Id	Description	Date	Status
1	A Pen	2005-06-05	New
2	2 Pencils	2005-06-05	New
3	4 chairs and a table	2005-06-05	New

The data displayed in the table, is data you previously entered (or data bundled with the **Tutorial 3-4** sample application).

Now try entering a new requisition:

Click the **New Requisition** button.

In the **Enter New Requisition** popup, enter a **Description**, and click **Submit**.

Notice that the table is not updated, although the data is saved, as you may verify by refreshing your browser.

Reuse the Action Process (to refresh the table display)

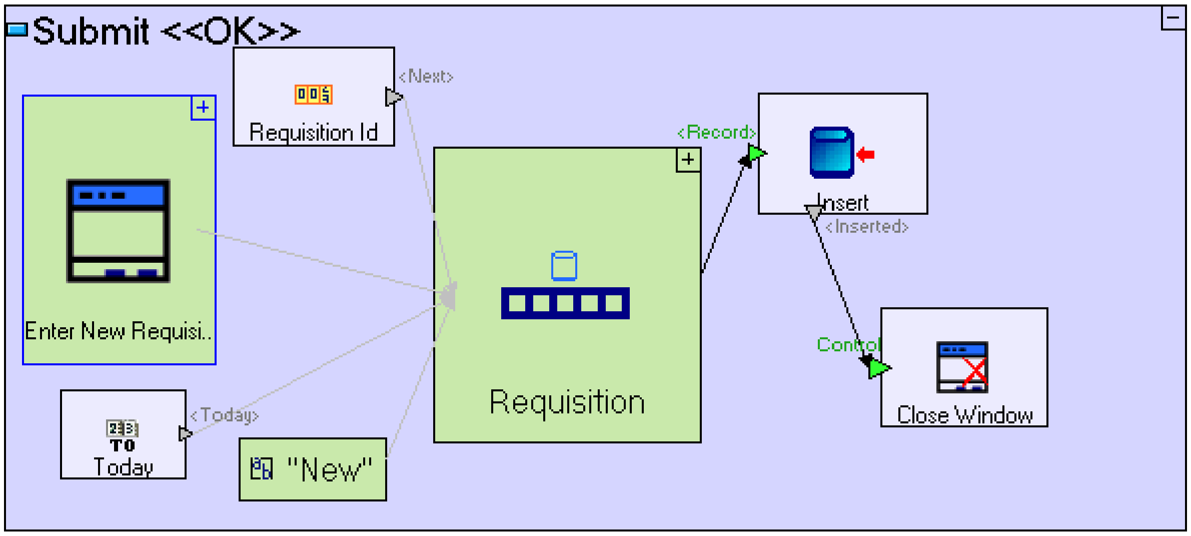
Expecting the user to refresh the browser display manually is unacceptable; the table must be refreshed automatically when a new requisition is submitted.

Refreshing the display means populating the **Requisition List** table with the most up-to-date data available, which is exactly what we have just finished modeling, so we can **reuse** the **Populate Open Requisitions List** process.

Since refreshing should be performed when the requisition is submitted, it should be modeled within the **Submit Requisition** button:

Zoom to the **Submit** button we've created in a previous stage.

At this stage, the **Submit** model should look similar to the following:



The light-grey arrows that appear in the screenshot above (flowing into **Requisition**), signify that the source or target of the flow, are not displayed (because their parent has been collapsed)

Add a new action process to the button to perform the required refreshing, after the requisition has been inserted successfully:

Select the **Basic/Action** template (), and drag it into **Submit Requisition**.

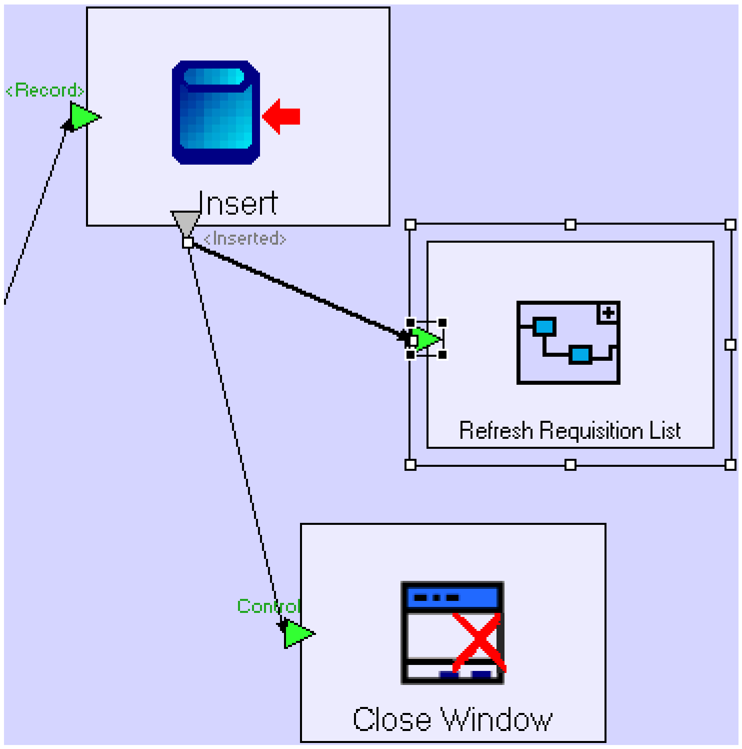
Name it ***Refresh Requisition List***.

Select the **Trigger** slot () from the palette and click on the frame of the **Refresh Requisition List** element.

Select the **Flow** tool, and link the **<Inserted>** slot of **Insert** to the **Trigger** slot of **Refresh Requisition List**.

Note that the trigger added to **Refresh Requisition List** demonstrates an alternative method for adding a trigger which controls the order in which processing occurs. The other method (using the **Add Element** context menu option) was demonstrated on the **Close Window** element we added (see **Stage 3**).
This also demonstrates that the name **Control** itself has no effect on the way the trigger acts. It just makes the model more readable (exposing the trigger's purpose via its name).

Your model should now look similar to the following:

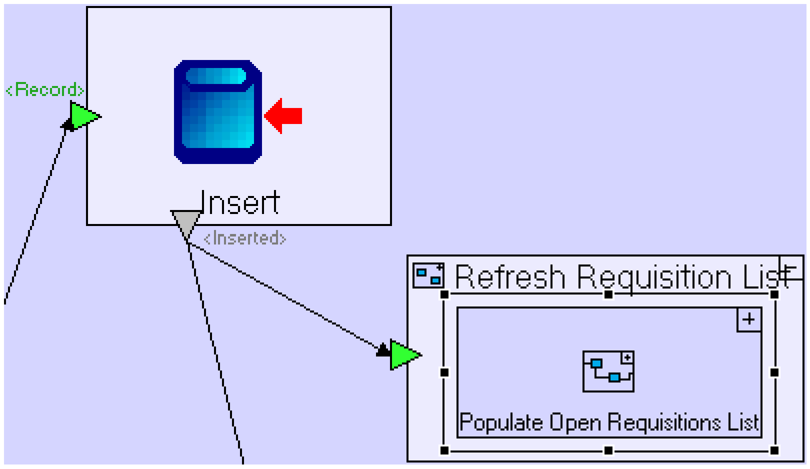


Notice that slots can serve as the source/target of multiple flow definitions. In our case, when the Insert model exits successfully, **Close Window** and **Refresh Requisition List** are both executed.

Now let's **reuse** the **Populate Open Requisitions List** process:

> Drag the **Populate Open Requisitions List** process from the Repository (or Outline) and drop it into **Refresh Requisition List**.

The **Refresh Requisition List** model should look similar to the following:



You may be wondering why we 'wrapped' **Populate Open Requisitions List** in another process (**Refresh Requisition List**), instead of placing it directly in the **Submit** button model, and creating the flow from **Insert** directly to it. The reason is that in that case we would need to add a trigger to the

Populate… process, and since the process is reused in the **Open Requisitions** view, the trigger will also appear there, and will stop functioning, as the trigger receives no flow in that scenario.

Save your work and view the application in the browser.

Try entering new requisitions and verify that they appear in the list.

Completing Stage 4

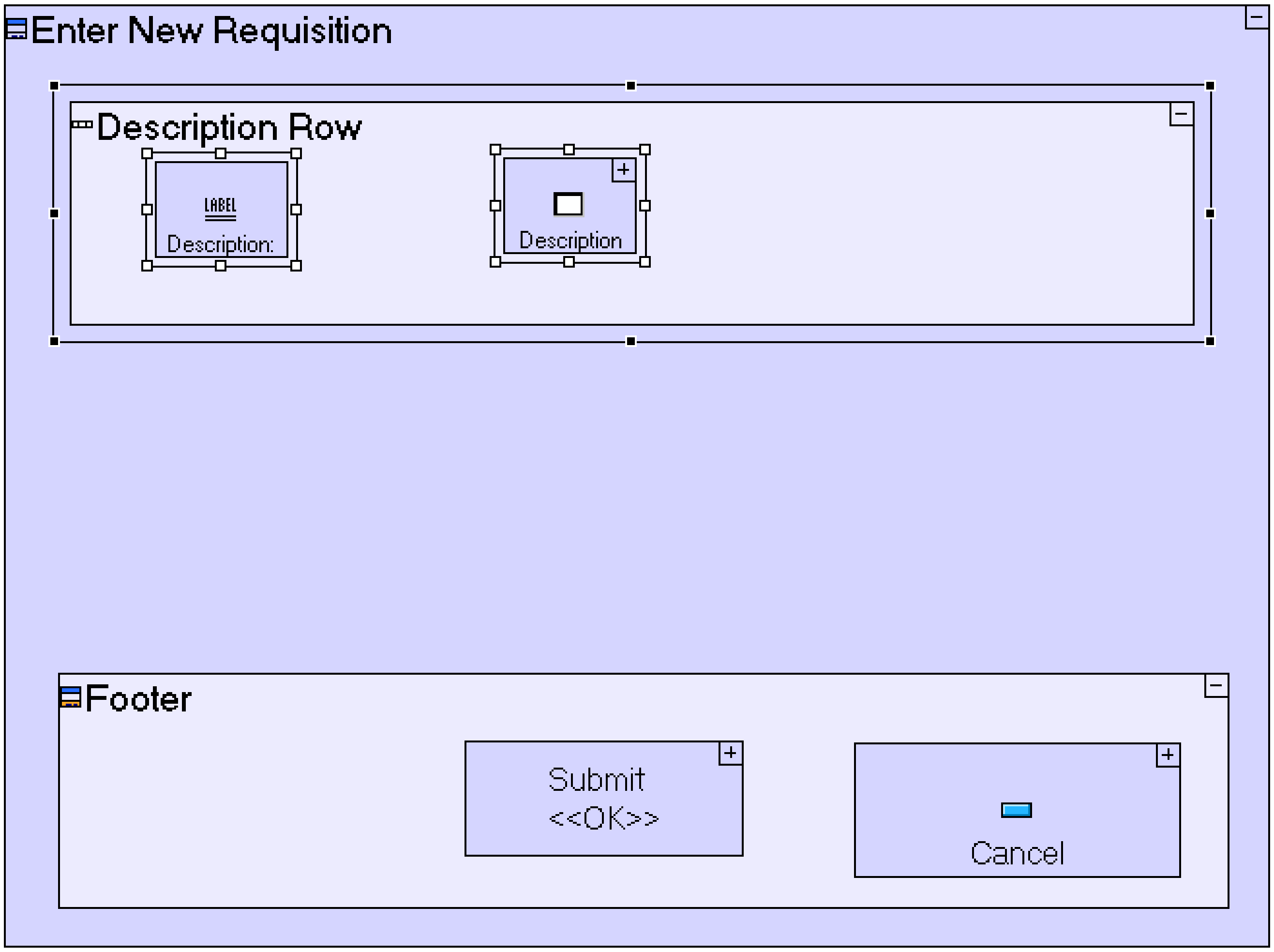
Import the sample project **Tutorial 4-5** and use it as the basis for the next stage of the tutorial.

For a reminder on how to import a sample project, see the **Importing a Sample Project** section at the end of **Stage 2**.

This sample project contains all the functionality modeled thus far.

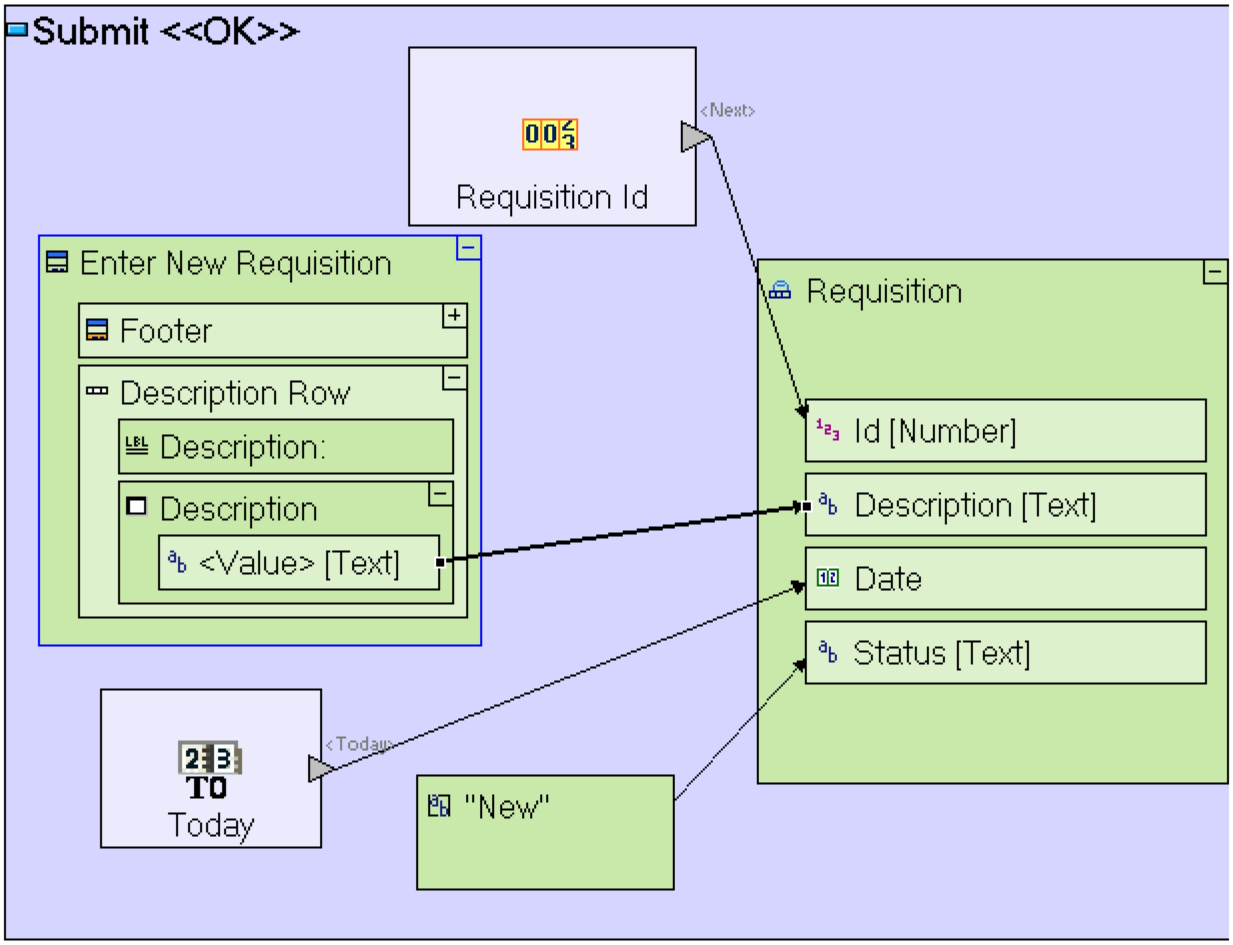
The sample project also includes additional functionality, as follows:

1. Place the **Description** text area and label in a row element, for improved formatting of the popup



How to Model	Located in
Add a **Display/Row**. Name it ***Description Row***.	**Enter New Requisition** popup
Drag **Description** text area from repository/outline. Drag **Description:** label from repository/outline.	**Description Row**
Delete **Description** text area. Delete **Description:** label.	**Enter New Requisition** popup

2. Fix flow in **Submit Requisition** model



How to Model	Located in

Click on red flow arrow. Drag source to **Enter New Requisition/Description Row/Description/<Value>**	**Submit Requisition** button

You may now proceed to **Stage 5**, in which we shall add support for an Urgency field to the requisition.

See It Live

Object 3

Click here to open the live project in a separate window.

Stage 5 – Modeling Choosers and Using Constants

Stage Goals

Tersus Concepts Covered

On completion of this stage you should be familiar with the following concepts:

Modeling notions:	Constant
Modeling techniques:	Creating a drop-down list (chooser)
	Updating a table structure
Useful display templates:	Row, Chooser

Application Functionality Modeled

In this stage you will model the addition of a field to an existing data structure (and the corresponding database table). The new field will have a predefined set of possible values, which the user can choose from a drop-down list.

This stage's modeling should be performed in the **Tutorial 4-5** project, you imported at the end of the previous stage.

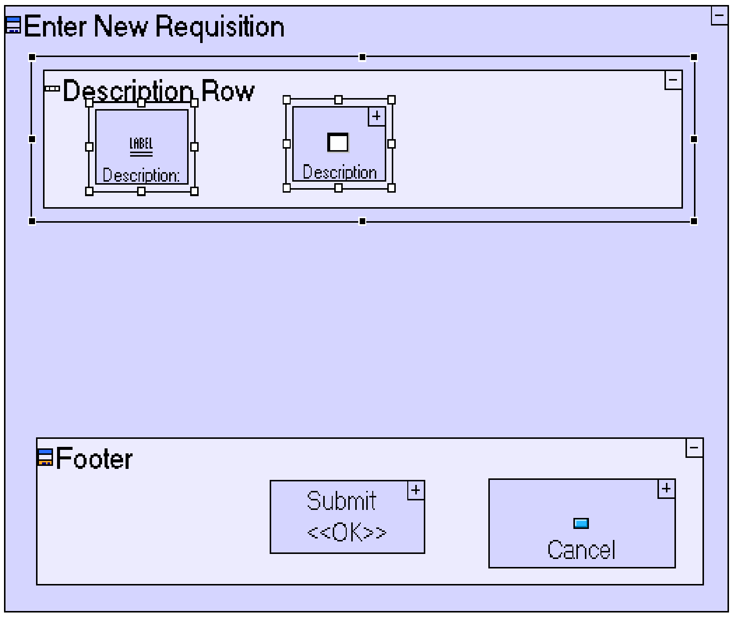
Model a Chooser

We want to enable the employee to distinguish urgent requisitions from regular one.

To do so, let's add an **Urgency** field to the requisition, which can have two possible values – **Regular** and **Urgent**. This is implemented using a **Chooser** display element which allows users to select from a list of predefined values.

Use a Row Element for Better Formatting (of the popup display)

Let's take a look at the **Enter New Requisition** popup, which has been updated in the model imported on completion of the previous stage:



Note that the **Description** text area and label have been moved into a **Row** display element named **Description Row**. This causes the popup to look neater, by ensuring the elements inside the row will always appear together in the same row.

We would like the **Urgency** chooser we are about to add, to be positioned in its own Row:

Zoom into the **Enter New Requisition** popup.

Select the **Display/Row** template (▭), and drag it into **Enter New Requisition**, below **Description Row**.

Name it ***Urgency Row***.

Add a Chooser Display (to the popup)

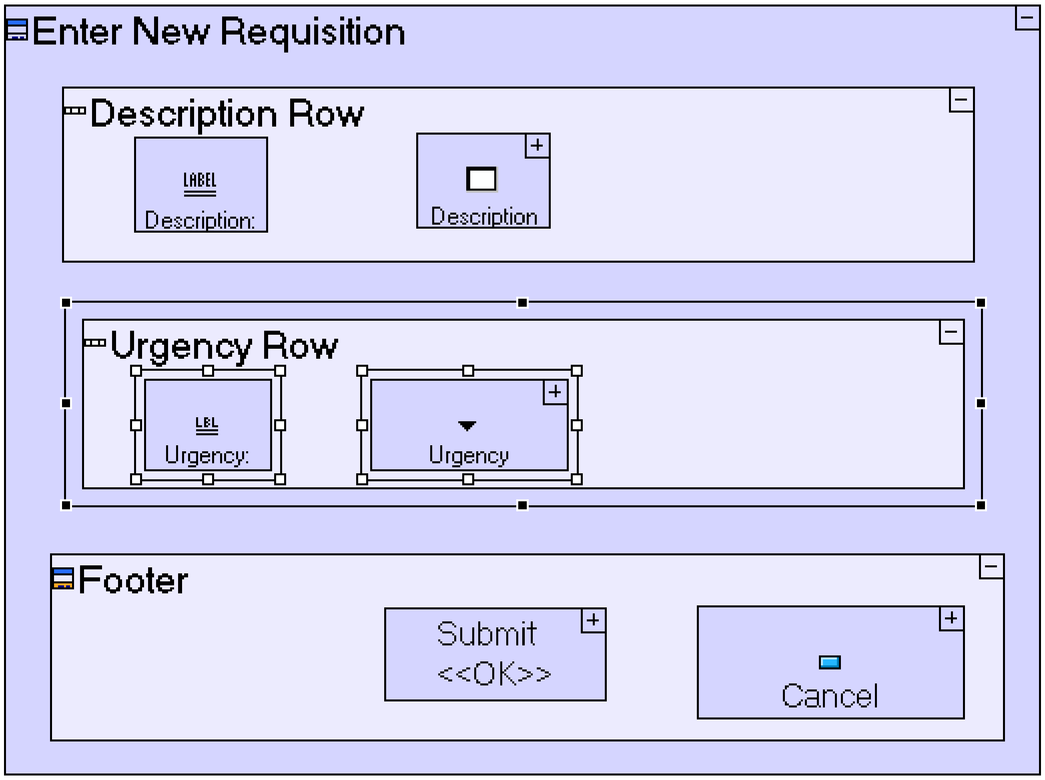
Next, we shall add the **Chooser** itself (plus a label):

Zoom into the **Urgency Row**.

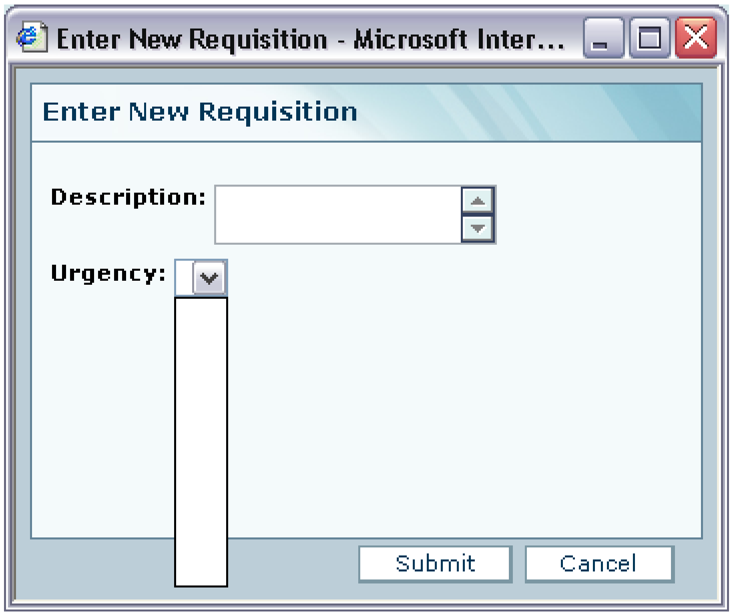
Select the **Display/Label** template (LABEL), and drag it into **Urgency Row**. Name it ***Urgency:***.

Select the **Display/Chooser** template (▼), and drag it into **Urgency Row**. Name it ***Urgency***.

The model should now look as follows:



If you now save your work, run the application and press the **New Requisition** button, you should see the row you've just modeled, which includes an empty drop down list (chooser), as in the following screen shot:



Create an Initialization Process (that populates the chooser with values)

To specify the possible values for **Urgency**, we'll need an initialization process in the **Enter New Requisition** popup. The initialization process will run each time the **Enter New**

Requisition popup is opened and will populate the **Urgency** drop down list with the possible values to choose from:

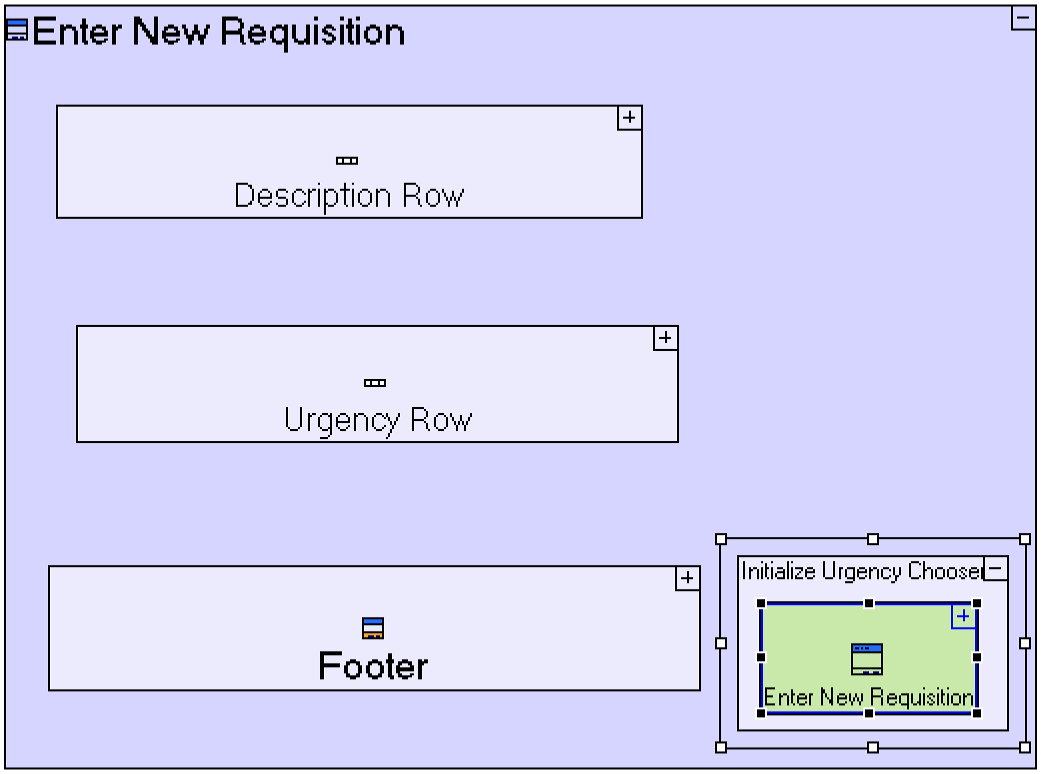
Select the **Basic/Action** template ( ), and drag it into **Enter New Requisition**; Name it ***Initialize Urgency Chooser***.

Zoom into **Initialize Urgency Chooser**.

The process must include a reference to the **Urgency** chooser display element, which is part of the **Enter New Requisition** popup:

Right-click on the **Initialize Urgency Chooser** process, select **Add Ancestor Reference** from the menu, and select **Enter New Requisition**.

The model should now look as follows:



Use Constants (to define the values displayed in the chooser)

Since there are only 2, pre-defined urgency values we wish to use, we can define those using Constants (which are data elements with a predefined fixed value).

Select the **Constants/Text** template ( ), and drag it into **Initialize Urgency Chooser**; Name it ***Regular***.

Notice that the **Regular** data element we have just added is displayed as **"Regular"** instead of **Regular**. This indicates it is a **Constant**.

"Regular" is a textual constant (a string of characters). There are also numeric constants, date constants, etc.

Values for a chooser need not necessarily be defined using constants. The values may also come from any other data source, such as the result of a previous process, a database table or a spreadsheet.

Now flow is needed to indicate the constant should populate the chooser:

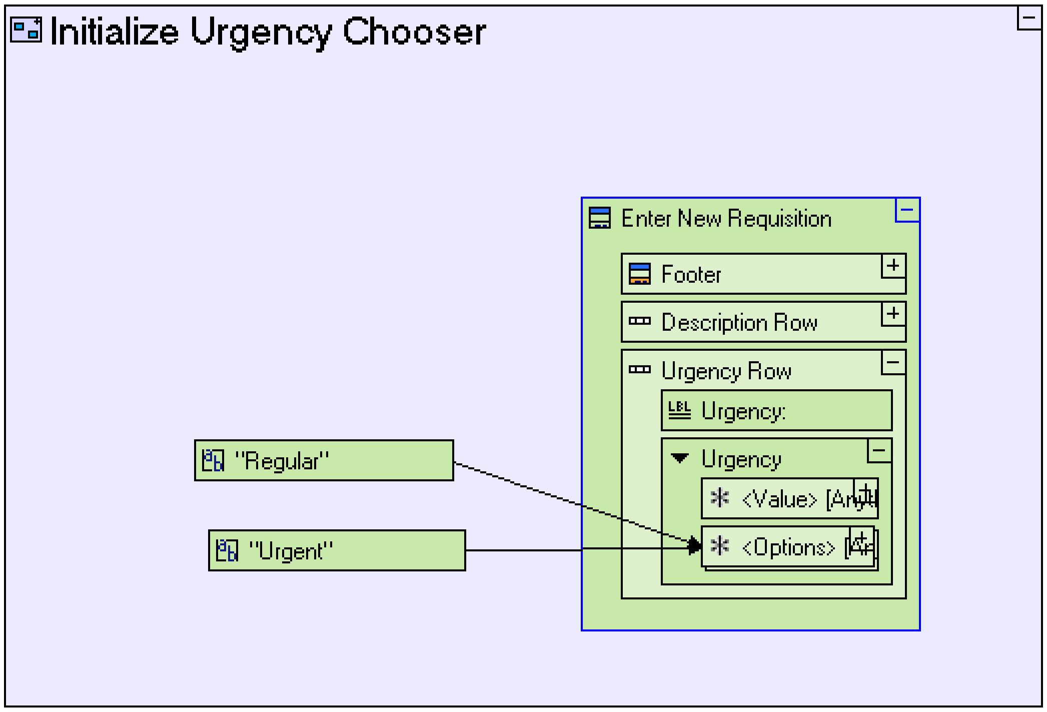
Create a flow from the "**Regular**" constant to **Enter New Requisition/Urgency Row/Urgency/<Options>**.

To specify a second possible value of the chooser **Urgency**:

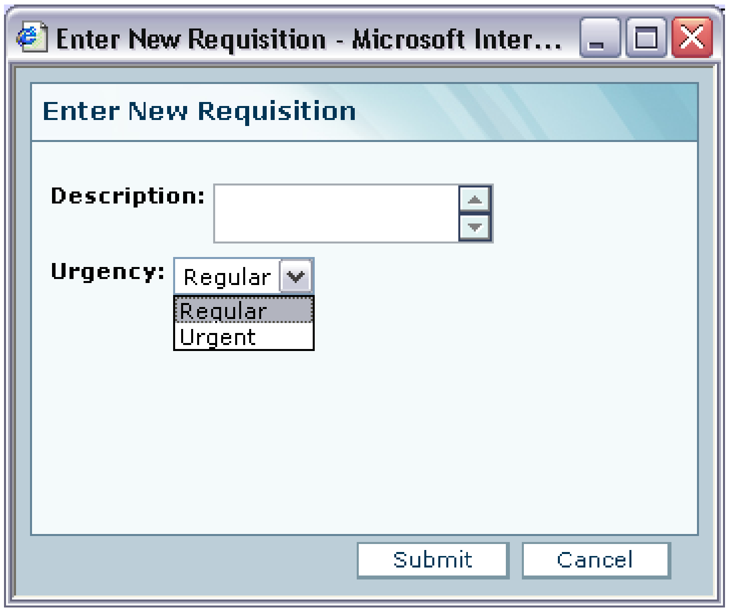
Select the **Constants/Text** template (ab), and drag it into **Initialize Urgency Chooser**; Name it ***Urgent***.

Create a flow from the "**Urgent**" constant to **Enter New Requisition/Urgency Row/Urgency/<Options>**.

The **Initialize Urgency Chooser** model should look similar to the following:



If you now save your work and run the application, the drop down list in the **Enter New Requisition** popup should allow you to choose between the two possible values **Regular** and **Urgent**, as in the following screenshot:



Add a Field to the Data Structure

The **Urgency** chooser now appears in the display, but the entered value is not saved with the requisition. We should also model the addition of the **Urgency** field to the **Requisition** database record.

Zoom to the **Requisition** database record in **Footer/Submit**.

Select the **Data Types/Text** template (^{a}b) and drop it into **Requisition**. Name it ***Urgency***.

Note that as the **Requisition** database record is reused in 2 models (**Submit** and **Requisition List**), you can actually perform the above steps in **Requisition List/Requisition**, with the same effect. In any case, the change will affect both.

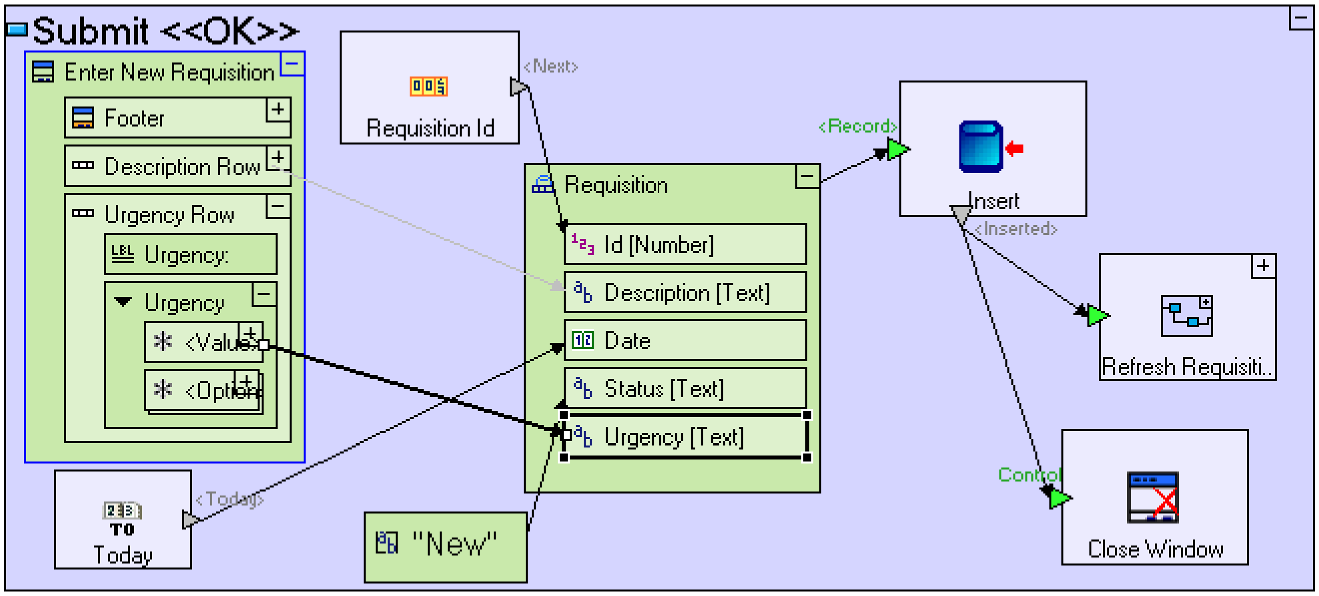
Next, add a Flow to populate the **Urgency** field in the **Requisition** data structure with the value the user selected in the chooser:

Zoom to the **Submit** button in **Footer**.

Create a flow from **Enter New Requisition/Urgency Row/Urgency/<Value>** to **Requisition/Urgency**.

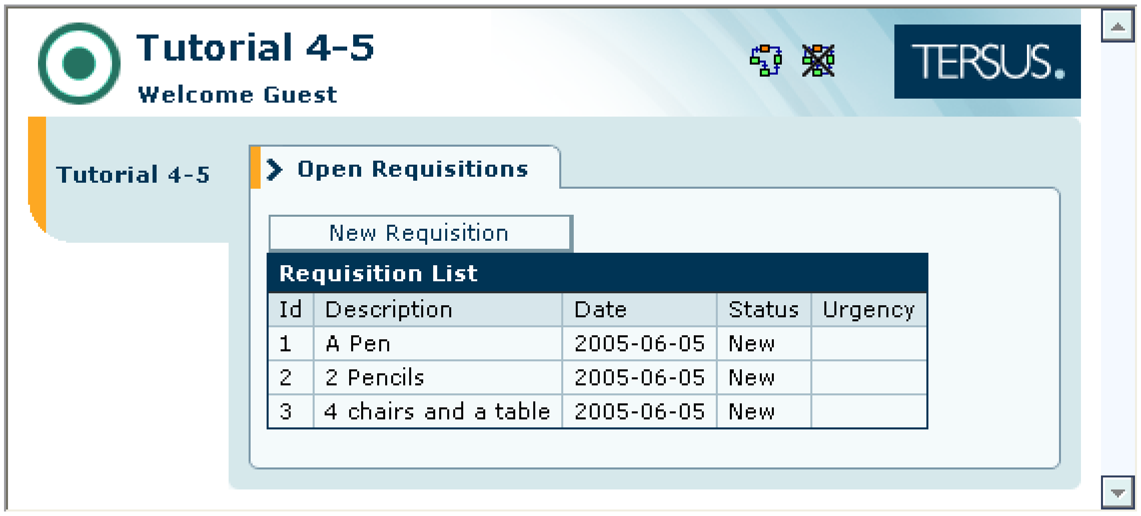
The Chooser has 2 predefined data elements, **<Options>** and **<Value>**. **<Options>** contains the values appearing in the chooser (which is why it is a repetitive element). **<Value>** contains the currently selected value.

The modeling we've performed in this last step should look similar to the following:



Reuse Means Faster Modeling (display & database structure are automatically updated)

If you now save your work, and launch your application in the browser, it should look as follows:



Requisition List

Id	Description	Date	Status	Urgency
1	A Pen	2005-06-05	New	
2	2 Pencils	2005-06-05	New	
3	4 chairs and a table	2005-06-05	New	

Without doing any additional modeling on the **Requisition List** table model, it now displays an **Urgency** column. This is because the **Requisition** database record is reused in the **Requisition List** model. And since the **Requisition** database record defines the actual structure of the database table, the structure of the **Requisitions** table in the database is also updated with a new field, **Urgency**.

Records that existed in the database before the addition of the **Urgency** field will have a NULL value, and will be displayed as an empty value as can be seen in the screen shot above.

Completing Stage 5

Import the sample project **Tutorial 5-6** and use it as the basis for the next stage of the tutorial.

For a reminder on how to import a sample project, see the **Importing a Sample Project** section at the end of **Stage 2**.

This sample project contains all the functionality modeled thus far.

You may now proceed to **Stage 6**, in which we shall model the next stage in a requisition's lifecycle, the approval of the requisition by the employee’s manager.

See It Live

Click here to open the live project in a separate window.

Stage 6 – Modeling an Additional View and Updating Data

Stage Goals

Tersus Concepts Covered

On completion of this stage you should be familiar with the following concepts:

Modeling notions:	Slot names (reserved & user-defined)
Modeling techniques:	Adding views to the application
	Processing a row selected in a display table
	Updating data stored in the database
Useful process templates:	Update
Useful display templates:	<Selected Row> (in Simple Table)

Application Functionality Modeled

In this stage you will add a new view to the application. In addition to the initial view, used to enter requisitions, the new view is used by managers to approve requisitions.

This stage's modeling should be performed in the **Requisition Management system (5-6)** project, you imported at the end of the previous stage.

User Modeling

Add a View to the Model

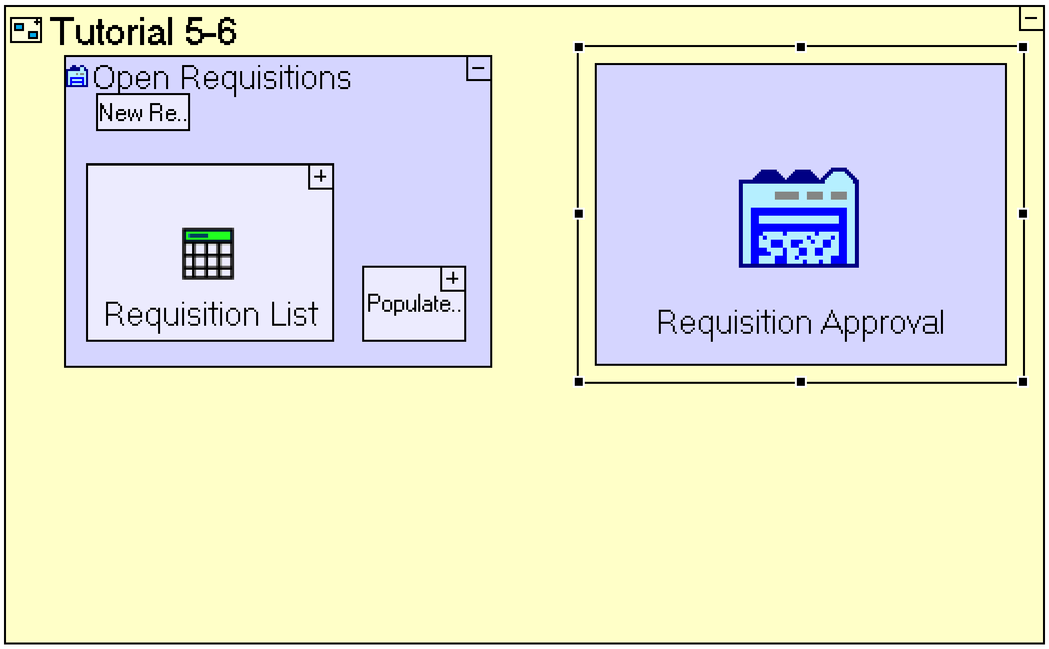
We shall now add a second view to our application, **Requisition Approval**. This view will be used by managers to view requisitions entered by their employees and decide whether to approve them (or, in the next stage, reject them).

Zoom to the root model (**Requisition Management System**).

Select the **Display/View** template () in the Palette and drop it next to the **Open Requisitions** view.

Name it ***Requisition Approval***.

The model should now look similar to the following:



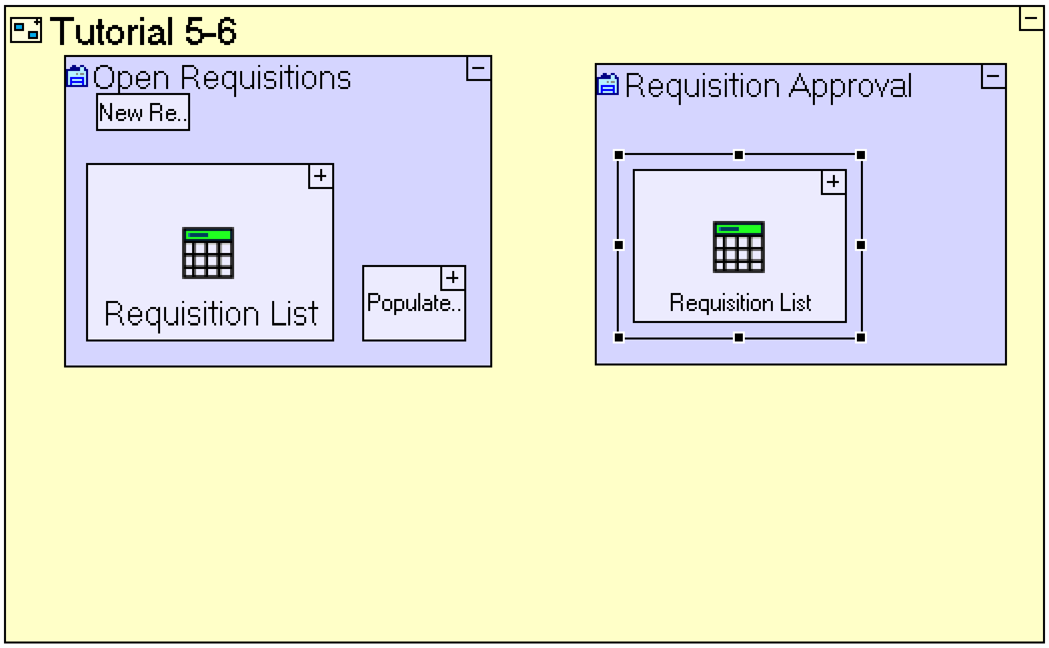
The **Requisition Approval** view will contain a list of open requisitions and buttons to Approve/Reject requisitions.

Reuse a Display Element (Requisition List)

We have already modeled the list of open requisitions in a previous stage, so let's reuse it:

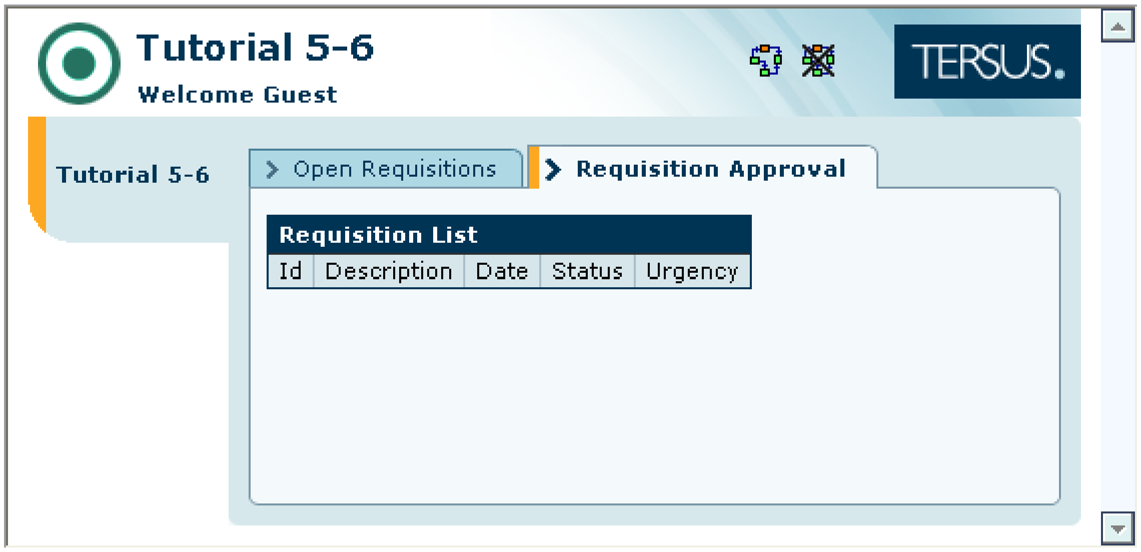
> Drag **Requisition List** from the Repository (or Outline) and drop it in the **Requisition Approval** view.

The model should look similar to the following:



Save your work and view the application in the browser. Note the additional tab **Requisition Approval** to the right of **Open Requisitions** view.

If you click **Requisition Approval**, you should see the following (the **Requisition List** table is displayed but is still empty):

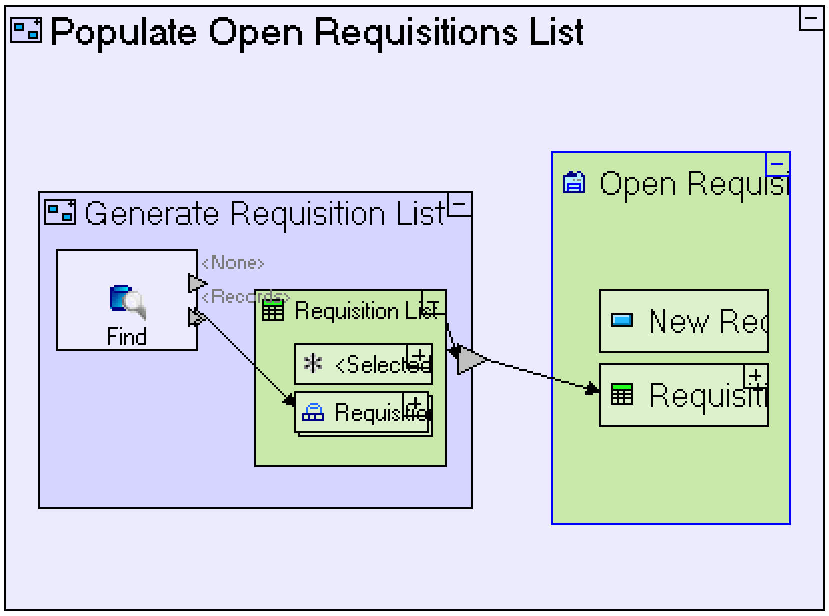


Id	Description	Date	Status	Urgency

Recreate a Process when it cannot be Reused in full (to populate the requisition list)

The **Requisition List** is empty because we have not reused the process which populates it originally (in the **Open Requisitions** view) - **Populate Open Requisitions List**.

Let's have a look at this process and recall what it does:



The **Populate Open Requisitions List** process includes a **Generate Requisition List** process which creates a **Requisition List** data element populated with **Requisitions** from the database. The generated **Requisition List** is then sent to populate the table display through the **Open Requisitions** ancestor reference.

This is nearly identical to what we want to perform here, except that the target table display is in the **Requisition Approval** view (the new view we have just defined) rather than the **Open Requisitions** view (the original view). So, we should create a new **Populate Requisition Approval List** process in the current view, reusing the **Generate Requisition List** process.

Zoom into **Requisition Approval**.

Select **Basic/Action** in the palette and drop it in **Requisition Approval**.

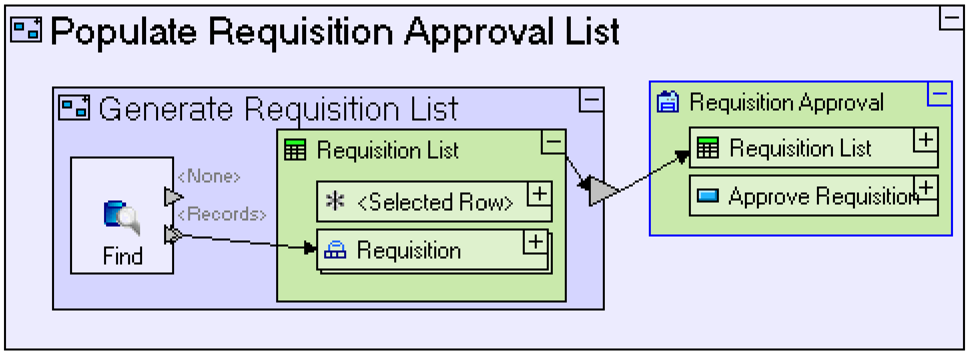
Name it ***Populate Requisition Approval List***.

To reuse **Generate Requisition List**, drag it from the outline and drop it into **Populate Requisition Approval List**.

Right-click on the **Populate Requisition Approval List** element, select **Add Ancestor Reference** from the menu, and select **Requisition Approval**.

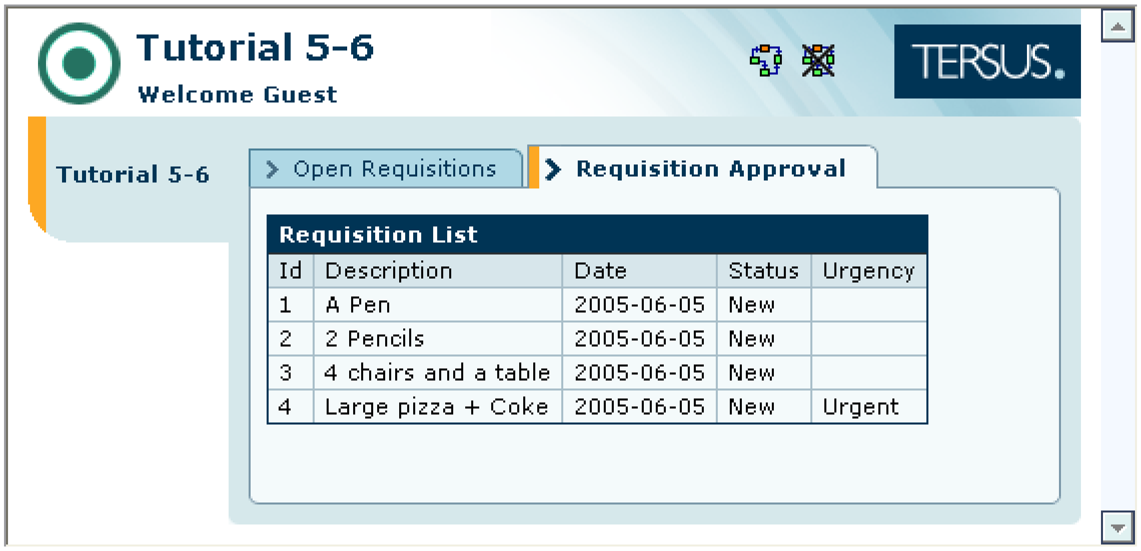
Use the **Flow** tool to link the exit of **Generate Requisition List** to **Requisition Approval/Requisition List**.

The model should look similar to the following:



Save your work and view the application in the browser.

If you click on the **Requisition Approval** tab, you should see the following:



Requisition List

Id	Description	Date	Status	Urgency
1	A Pen	2005-06-05	New	
2	2 Pencils	2005-06-05	New	
3	4 chairs and a table	2005-06-05	New	
4	Large pizza + Coke	2005-06-05	New	Urgent

You can verify that the requisition list appears in both views and updated when a new requisition is entered in the **Open Requisitions** view.

Add a Button (that updates an existing record, marking it Approved)

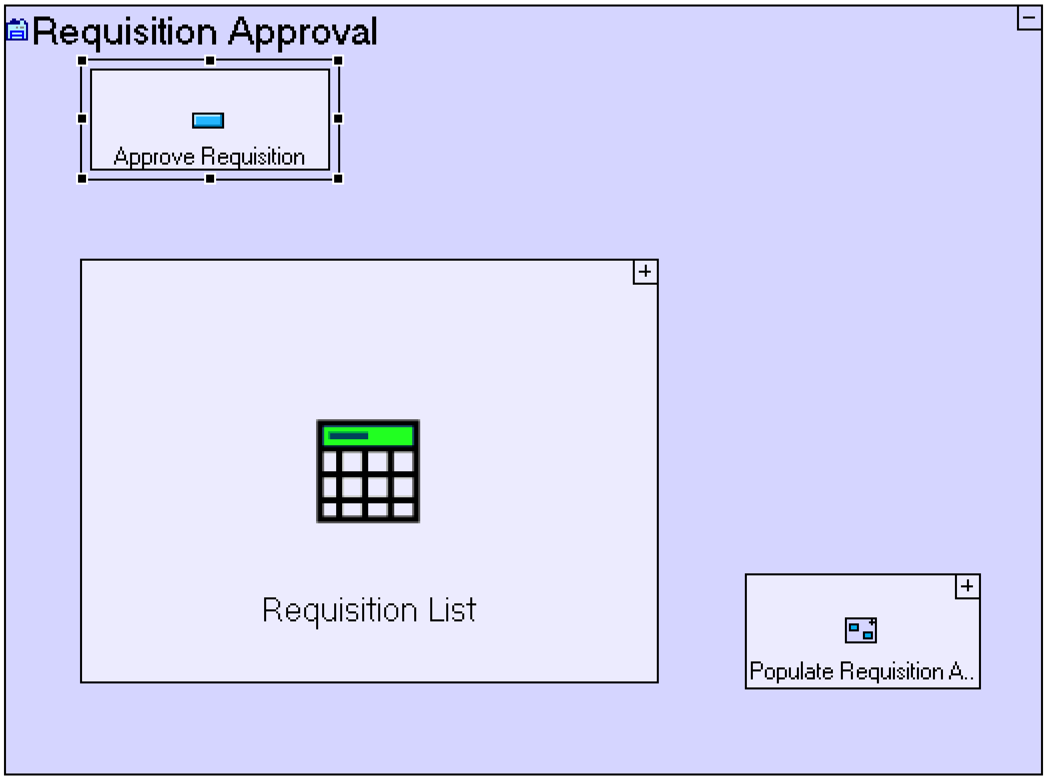
The **Requisition Approval** view should provide a manager with the ability to select a row in the table and mark it Approved (or Rejected).

Let's start with creating the button itself. To keep this view consistent with the **Open Requisitions** view, place the button above the table:

Select **Display/Button** in the palette and drop it in **Requisition Approval** (above the **Requisition List**).

Name it ***Approve Requisition***.

The **Requisition Approval** model should look similar to the following:



Now, let's model the button logic, which should consist of the following:

1. Retrieving the selected row from the table
2. Changing the **Status** field value for the row to **Approved**
3. Updating the corresponding record in the database
4. Refreshing the **Requisition List** table to display the updated record

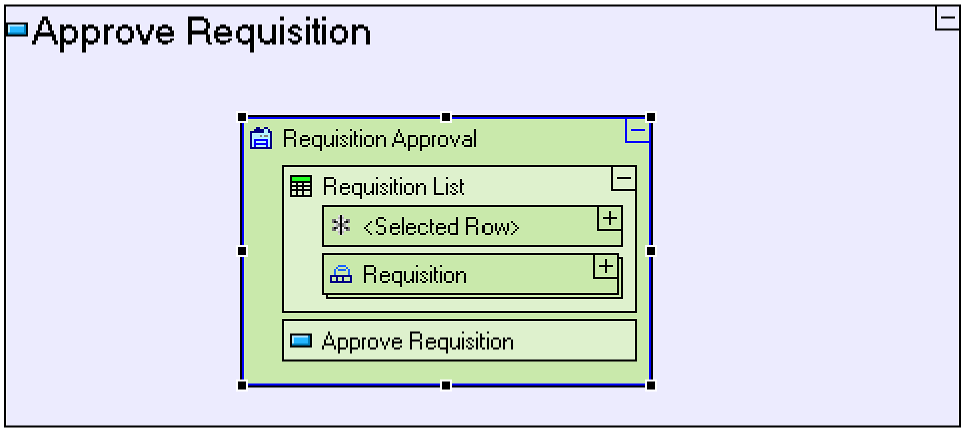
Retrieve the Selected Row from a Table Display

To retrieve the selected row from a table display, we first need to include a reference to the display.

Zoom into **Approve Requisition**.

Right-click on **Approve Requisition**, select **Add Ancestor Reference** from the menu, and select, **Requisition Approval**.

The **Approve Requisition** model should look similar to the following:



The **<Selected Row>** data element is included by default in any display element based on the **Simple Table** (or **Table**) template. Whenever the user clicks a row in the table (at runtime), the **<Selected Row>** element will contain the data in that row.

> The **<Selected Row>** element supports any row structure defined in it's parent model (the table). This is possible as it is based on the **Anything** data type.

Change a Field's Value

The **<Selected Row>** element returns the selected row as a **Requisition** data structure. We should now create a process which changes the **Status** of the selected requisition to **Approved**.

Select **Basic/Action** in the palette and drop it next to the **Requisition Approval** ancestor reference.

Name it ***Change Requisition Status***.

The **Change Requisition Status** process we are creating should receive the selected **Requisition** as input:

Select the **Trigger** slot ( ) in the palette and drop it on the frame of **Change Requisition Status**.

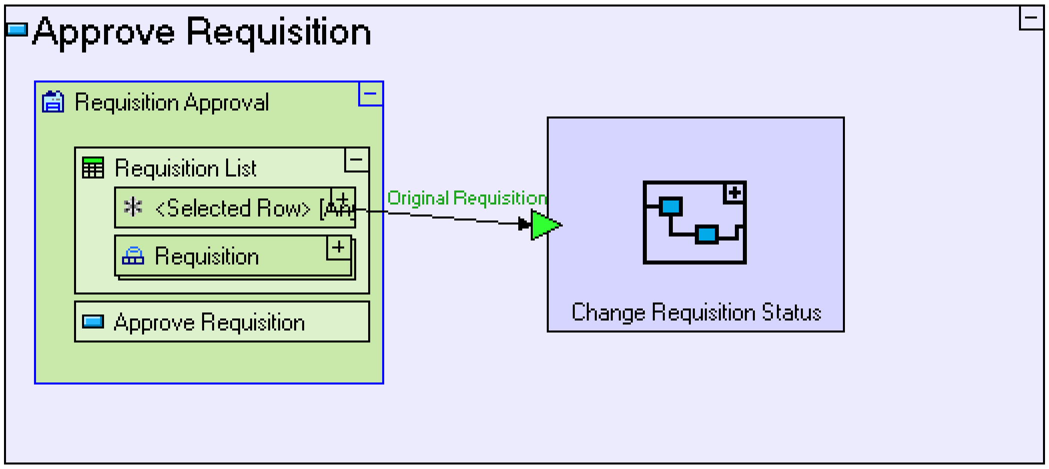
Add **Flow** linking **Requisition Approval/Requisition List/<Selected Row>** to the trigger just added to **Change Requisition Status**.

In a moment we will add additional slots, so it would be a good idea to name the slot we have just created so that its purpose (and content) is clear to anyone looking at the model.

Select the trigger we have just added and press **[F2]**.

In the edit box that appears, enter ***Original Requisition*** as the name of the trigger.

The **Approve Requisition** model should look similar to the following:



An additional input to **Change Requisition Status** should be the new value to set **Status** to:

Select **Constants/Text** in the palette and drop it next to the new trigger (outside **Change Requisition Status**). Name it ***Approved***.

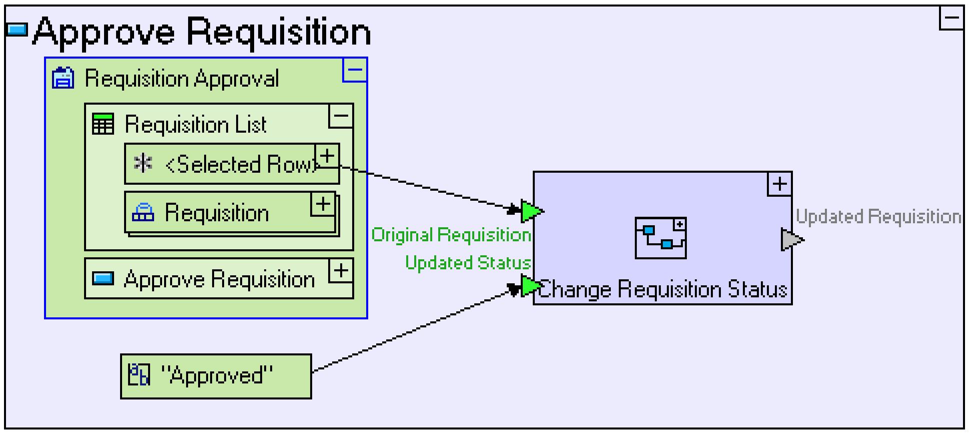
Add another **Trigger** to **Change Requisition Status**. Select it and press **[F2]** to name it ***Updated Status***.

Add **Flow** linking the "**Approved**" constant to the **Change Requisition Status/Updated Status** trigger.

Finally, the process should output the updated **Requisition:**

Add an **Exit** slot (▷) to the frame of **Change Requisition Status**. Select it and press **[F2]** to name it ***Updated Requisition***.

The **Approve Requisition** model should look similar to the following:



Now let's drill into **Change Requisition Status** and model the update of the **Status** field of **Requisition**.

Zoom into **Change Requisition Status**.

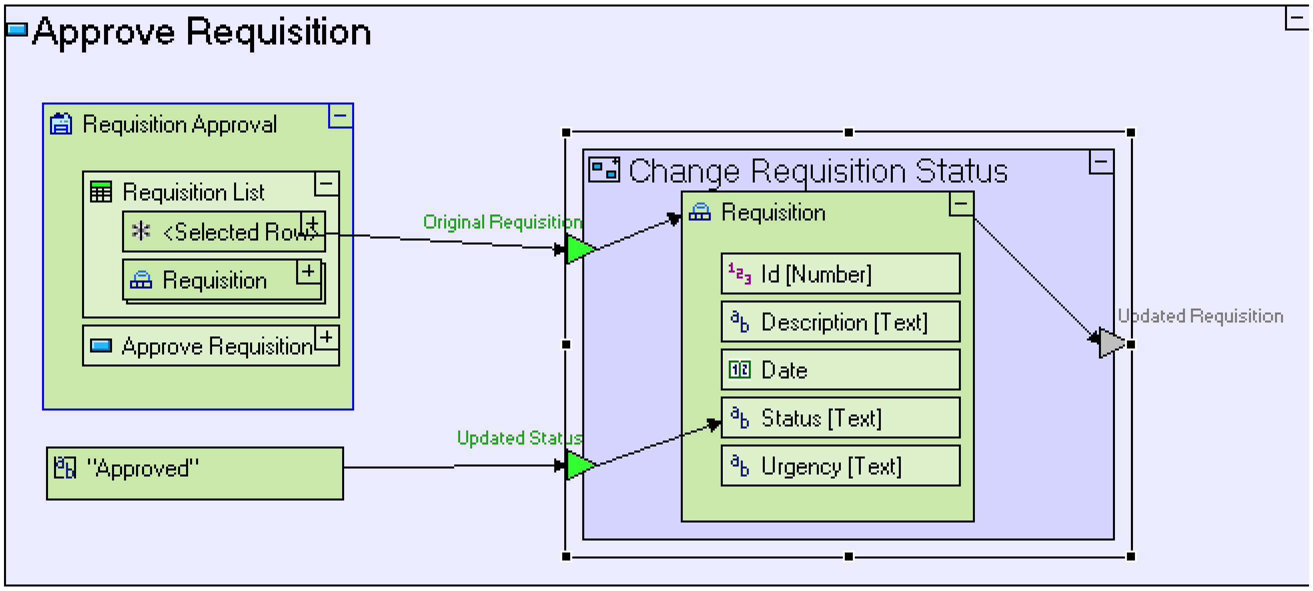
Drag the **Requisition** database record into **Change Requisition Status**.

Add **Flow** linking the **Original Requisition** trigger to the **Requisition** data structure.

Add **Flow** linking the **Updated Status** trigger to the **Requisition/Status** field.

Add **Flow** linking the **Requisition** data structure to the **Updated Requisition** exit.

The **Change Requisition Status** model should look similar to the following:



Commit the Updated Record to the Database and Refresh the Table Display

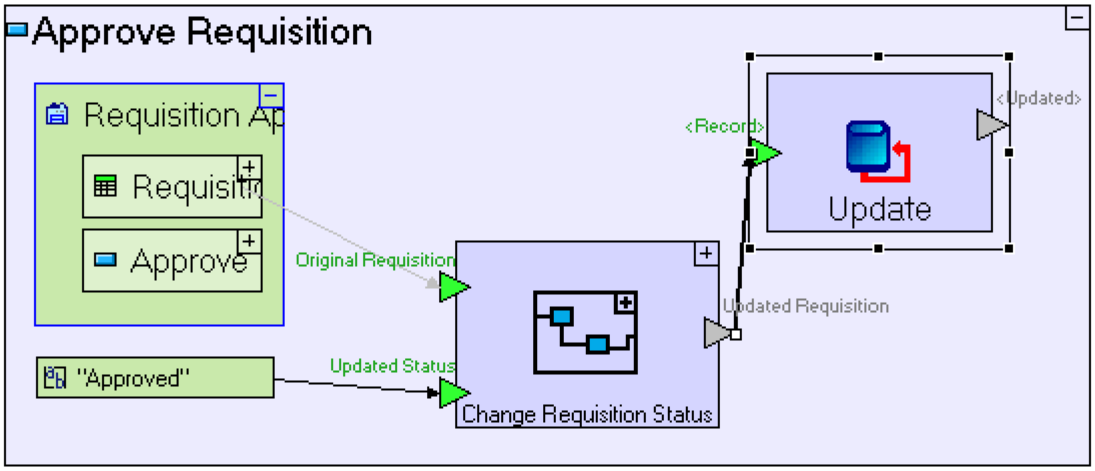
The **Updated Requisition** exit slot returns an approved requisition, which should be updated in the database. We shall use the **Update** template for this purpose.

Zoom out to the **Approve Requisition** button model.

Select the **Database/Update** template () and place it next to the **Change Requisition Status** process.

Create **Flow** linking the **Change Requisition Status/Updated Requisition** exit to **Update/<Record>** trigger.

The **Approve Requisition** model should look similar to the following:



We have to make sure the table is refreshed on screen if Update was successful, so that the display is kept synchronized with the database. On a successful update, the **Update** process will exit through its **<Updated>** exit:

Drop a **Basic/Action** template next to **Update**. Name it ***Refresh Requisition List***.

We're creating a new process model, with an identical name to an existing process (there's a similar **Refresh Requisition List** process in **Submit Requisition**). This is possible when the 2 models are in different "packages" in the repository. Giving a new model the name of an existing model in the same package is prohibited.

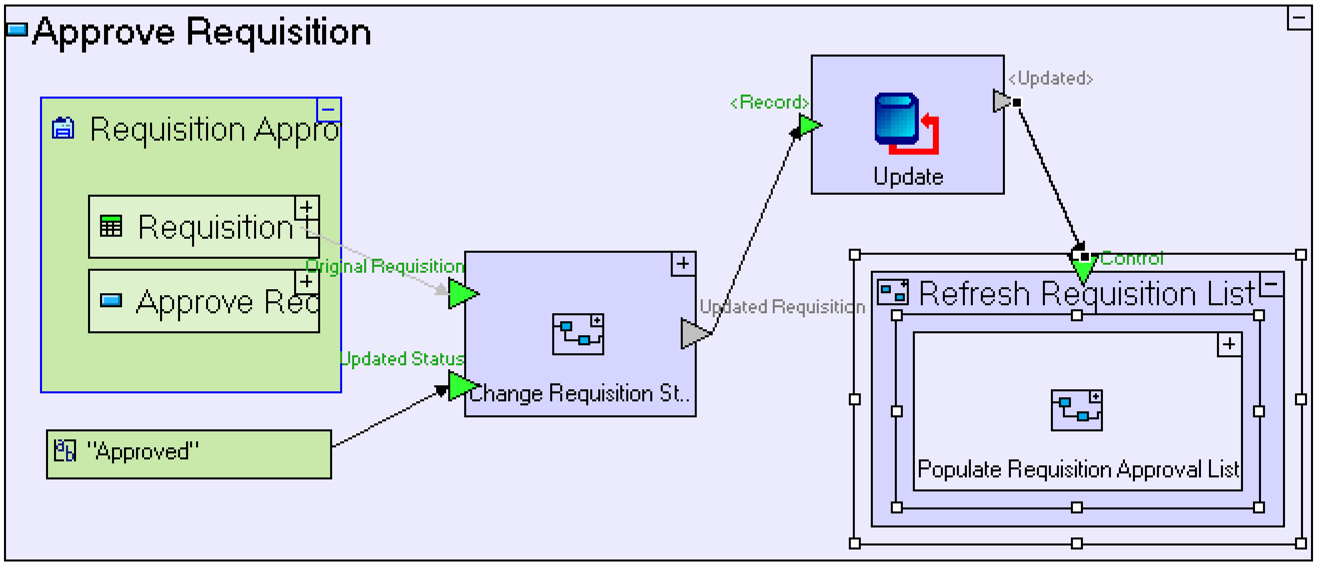
Right-click **Refresh Requisition List** and open the **Add Element** sub menu to select the **Control** trigger.

Create **Flow** linking the **Update/<Updated>** exit to the **Refresh Requisition List** trigger.

And lastly, reuse the **Populate Requisition Approval List** process we created in the beginning of this stage:

Drag the **Populate Requisition Approval List** process from the outline into the **Refresh Requisition List** process.

The **Approve Requisition** model should look similar to the following:



Save your work and view the application in the browser.

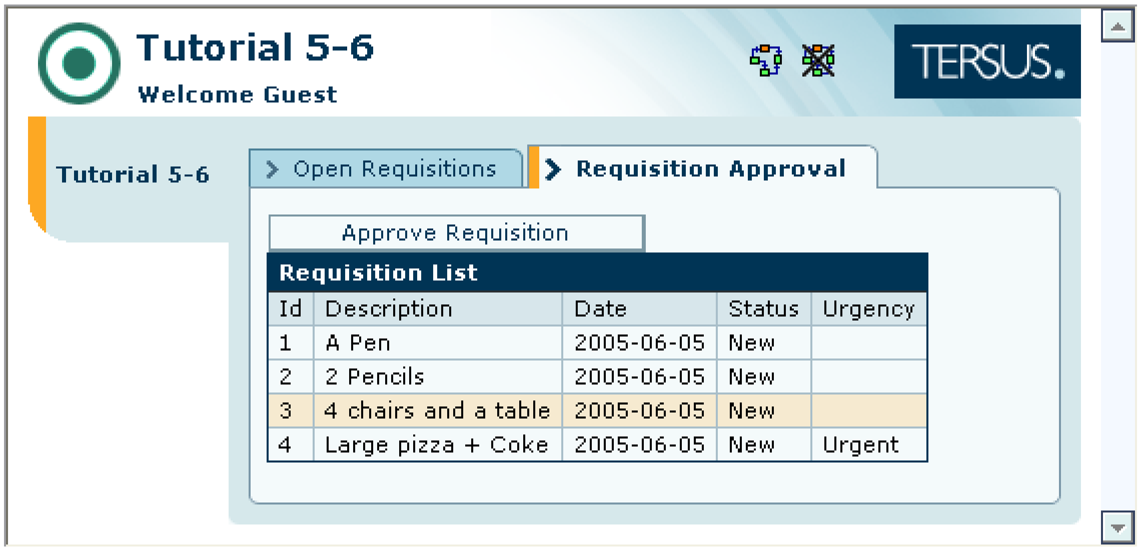
Click on the **Requisition Approval** tab.

Click on the **Approve Requisition** button before selecting a requisition.

Nothing should occur, since no requisition is selected.

Click on one of the new requisitions in the list.

The selected row is signified by a slightly darker background color, as in the following screenshot (3rd row):

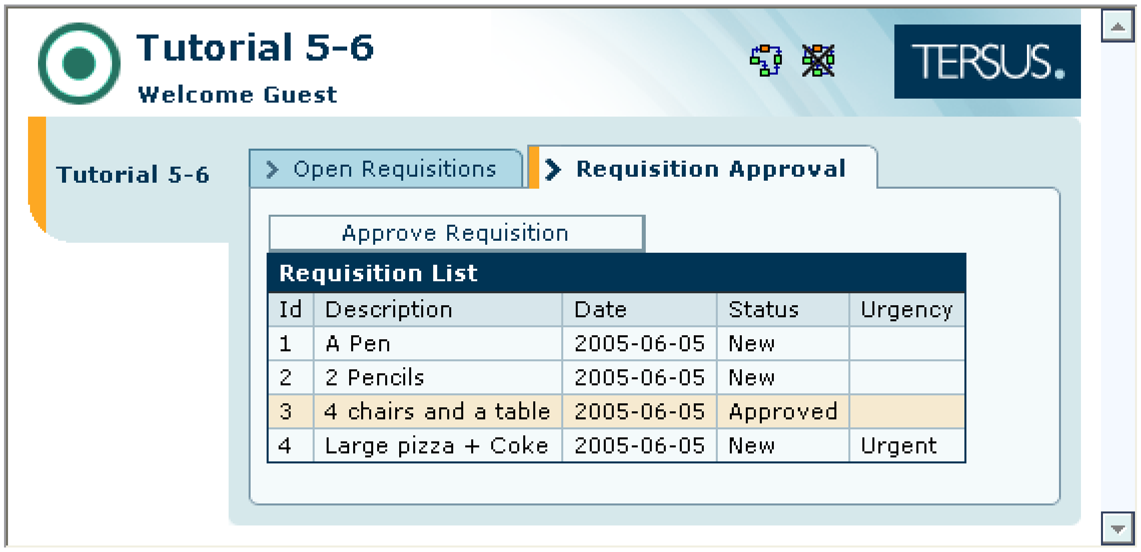


Requisition List

Id	Description	Date	Status	Urgency
1	A Pen	2005-06-05	New	
2	2 Pencils	2005-06-05	New	
3	4 chairs and a table	2005-06-05	New	
4	Large pizza + Coke	2005-06-05	New	Urgent

Click on the **Approve Requisition** button.

The table should be refreshed and the requisition selected should change status to **Approved**, as in the following screenshot:



Requisition List

Id	Description	Date	Status	Urgency
1	A Pen	2005-06-05	New	
2	2 Pencils	2005-06-05	New	
3	4 chairs and a table	2005-06-05	Approved	
4	Large pizza + Coke	2005-06-05	New	Urgent

Completing Stage 6

Import the sample project **Tutorial 6-7** and use it as the basis for the next stage of the tutorial.

For a reminder on how to import a sample project, see the **Importing a Sample Project** section at the end of **Stage 2**.

This sample project contains all the functionality modeled thus far.

You may now proceed to **Stage 7**, in which we shall model similar buttons to the Approve Requisition button modeled in this stage, by re-factoring (change) parts of the modeling and reusing it for canceling and rejecting requisitions**.**

See It Live

Click here to open the live project in a separate window.

Stage 7 – Re-factoring - Changing a Process to Enhance Reusability

Stage Goals

Tersus Concepts Covered

On completion of this stage you should be familiar with the following concepts:

Modeling notions:	Re-factoring
Modeling techniques:	Deleting model elements

Application Functionality Modeled

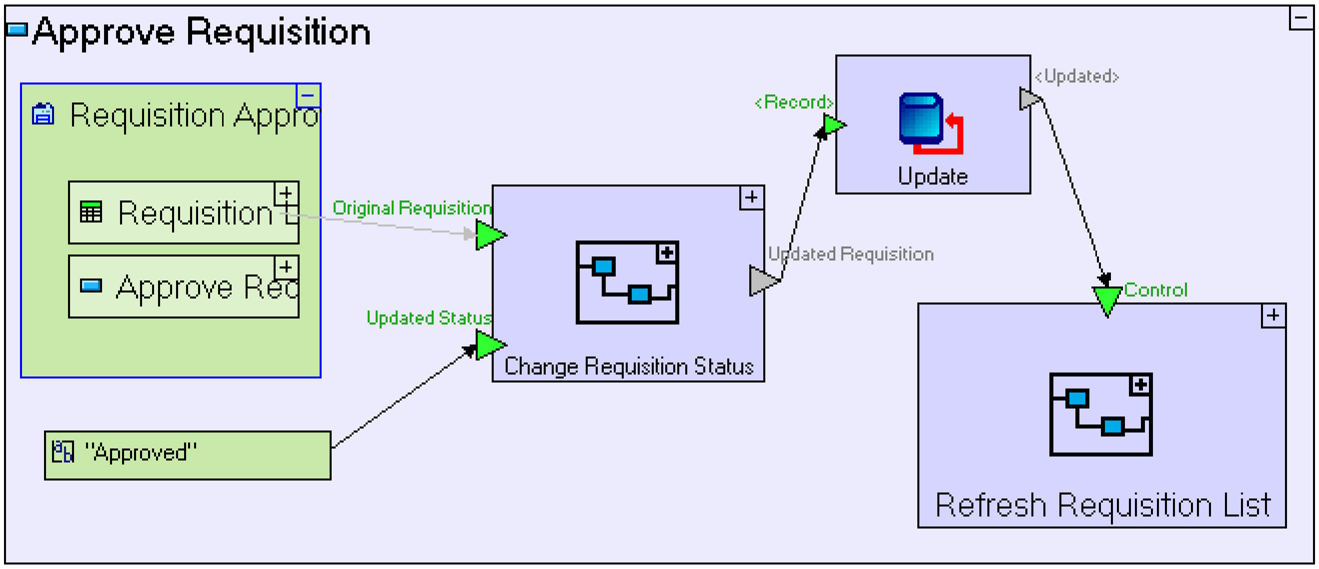
In this stage you will add a **Cancel Requisition** button, in the **Open Requisitions** view. Canceling a requisition is performed simply by setting its **Status** field to **Canceled**.

This will be followed by a **Reject Requisition** button, in the **Requisition Approval** view, setting the **Status** field to **Rejected**.

This stage's modeling should be performed in the **Tutorial 6-7** project, you imported at the end of the previous stage.

Re-factor an Existing Model

The **Cancel Requisition** and **Reject Requisition** models are very similar to the **Approve Requisition** model we created in the previous stage, so let's recall how it was modeled:



Of-course, we cannot reuse the **Approve Requisition** model as-is, because it depends on some elements which are specific to it, and may be different for **Cancel Requisition** and **Reject Requisition**, as illustrated in the following table:

Model / Element	**Approve Requisition**	**Cancel Requisition**	**Reject Requisition**
View (ancestor reference)	**Requisition Approval**	**Open Requisitions**	**Requisition Approval**
Status constant	**"Approved"**	**"Canceled"**	**"Rejected"**
The refresh process to be executed following the Update	**Refresh Requisition List** (from the **Requisition Approval** package)	**Refresh Requisition List** (from the **Open Requisitions** package)	**Refresh Requisition List** (from the **Requisition Approval** package)

Other than these elements all other elements of the model are identical, so it makes sense to change the model by grouping all the reusable functionality in a separate process. This modeling technique (re-arranging the model differently) is referred to as **Re-factoring**.

Group Reusable Elements in a Process

We shall begin the re-factoring of the **Approve Requisition** button by creating the reusable process:

Zoom to **Approve Requisitions**.

Drop a **Basic/Action** template into **Approve Requisitions**.

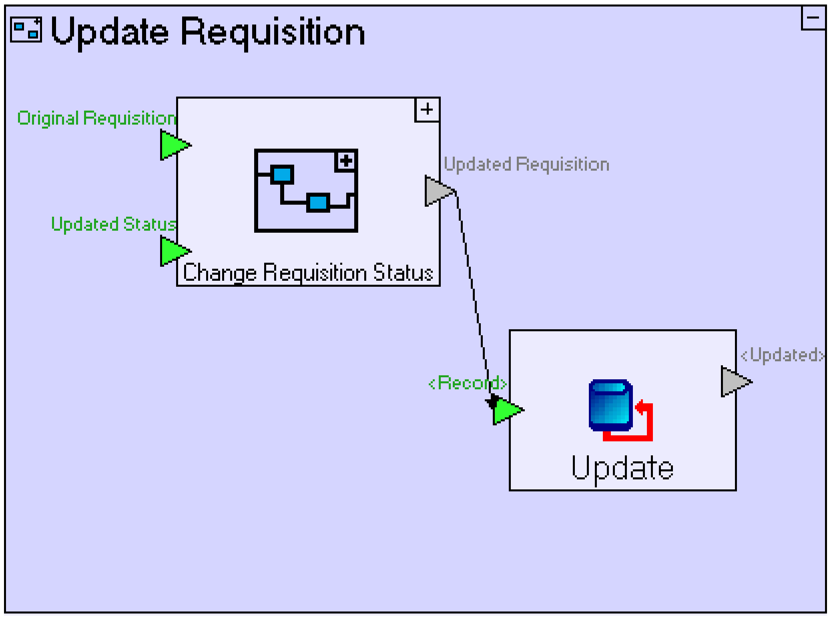
Name it ***Update Requisition***, and zoom into it.

Next, copy into the newly created process the reusable functionality we outlined above:

Drag and drop the following processes from the repository (or outline): **Change Requisition Status** and **Update**.

Create **Flow** linking the **Change Requisition Status/Updated Requisition** exit to **Update/<Record>** trigger.

The **Update Requisition** process should look similar to the following:



The slots which have no flow connected to them hint at what we should handle next. Recall, that all the slots in question (e.g. the trigger **Original Requisition**) were originally the target or source for flows from/to elements which we cannot reuse. We should add corresponding slots to the **Update Requisition** process we are now modeling, so that the non-reusable elements could be specified outside the reusable process.

Place a **Trigger** on the upper left frame of **Update Requisition**. Name it (using [F2]) ***Selected Requisition***.

Create **Flow** linking this trigger to **Change Requisition Status/Original Requisition**.

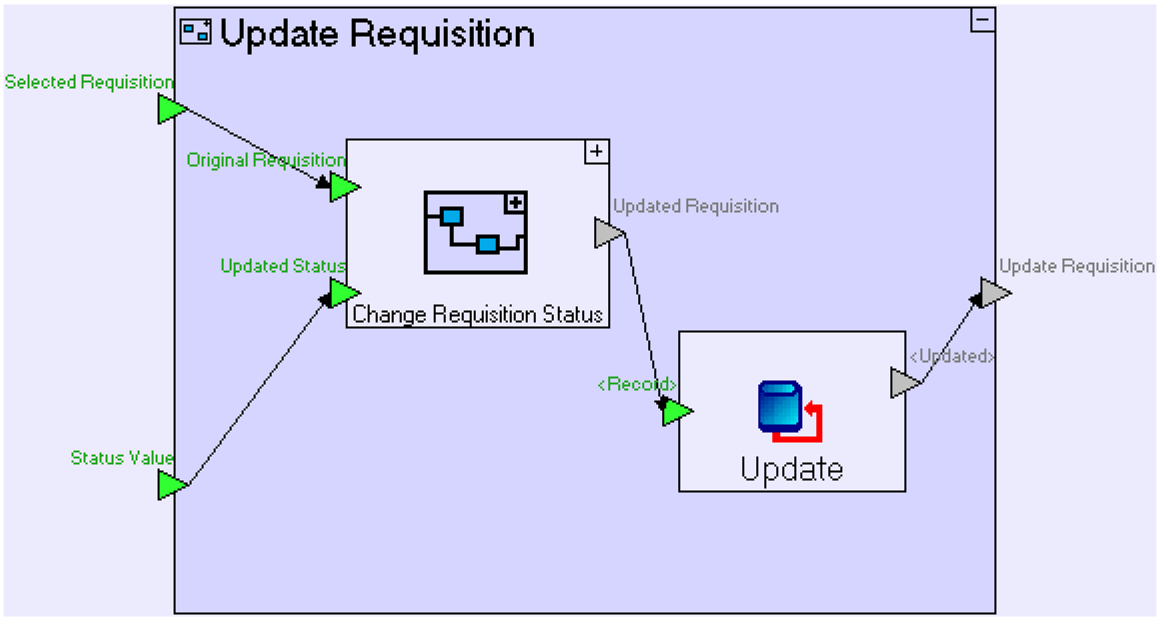
Place a second **Trigger** on the left frame of **Update Requisition**. Name it ***Status Value***.

Create **Flow** linking this trigger to **Change Requisition Status/Updated Status**.

Place an **Exit** on the right frame of **Update Requisition** frame.

Create **Flow** linking the **Update/<Updated>** exit to this exit.

The **Update Requisition** process should look similar to the following:



Finish Re-factoring (replacing elements and flow with the new process)

To finish the Re-factoring, we need to clean-up **Approve Requisition** by redefining flow to/from the **Update Requisition** process and removing redundant elements.

Collapse (clicking ⊟ in the top-right corner of) **Update Requisition** process (to minimize confusion or mistakes).

Zoom out to **Approve Requisition**.

Redefine 3 **Flows**:

Click on the **Flow** connecting the **Requisition Approval** display data element with the **Change Requisition Status/Original Requisition** trigger, to select it.

Click on the **Flow**'s target anchor ( ) and drag it onto the **Update Requisition/Selected Requisition** trigger.

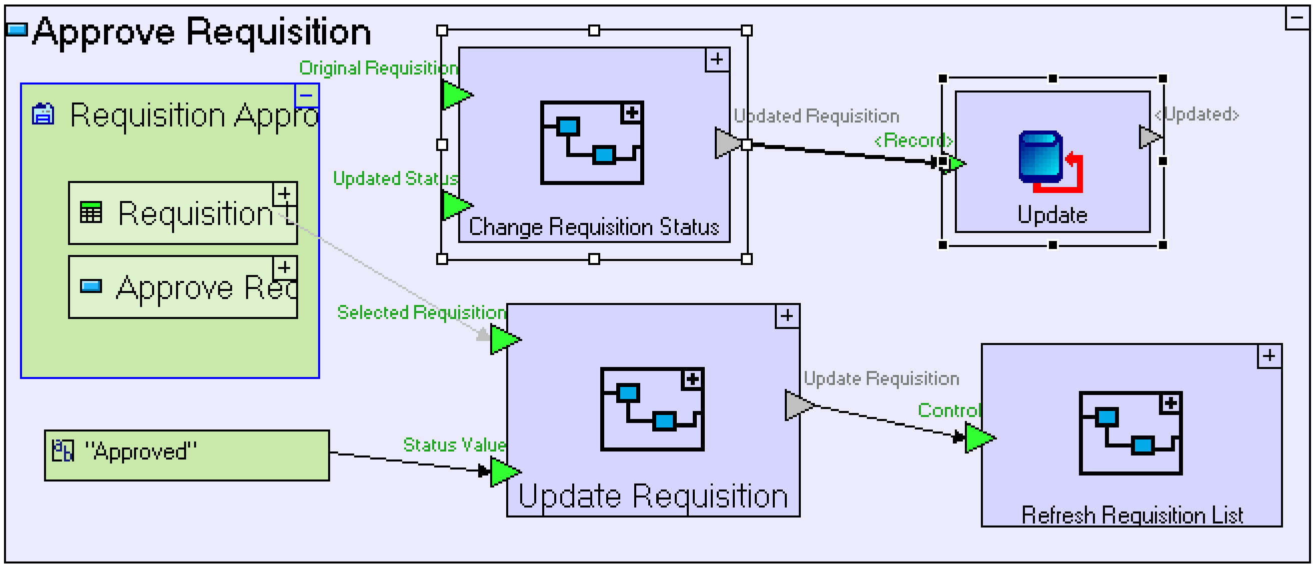
Select the **Flow** connecting the "**Approved**" constant to **Change Requisition Status/Updated Status** trigger.

Click on the **Flow**'s target anchor ( ) and drag it onto the **Update Requisition/Status Value** trigger.

Select the **Flow** connecting the **Update** process to the **Refresh Requisition List** process.

Click on the **Flow**'s source anchor ( ) and drag it onto the exit of **Update Requisition**.

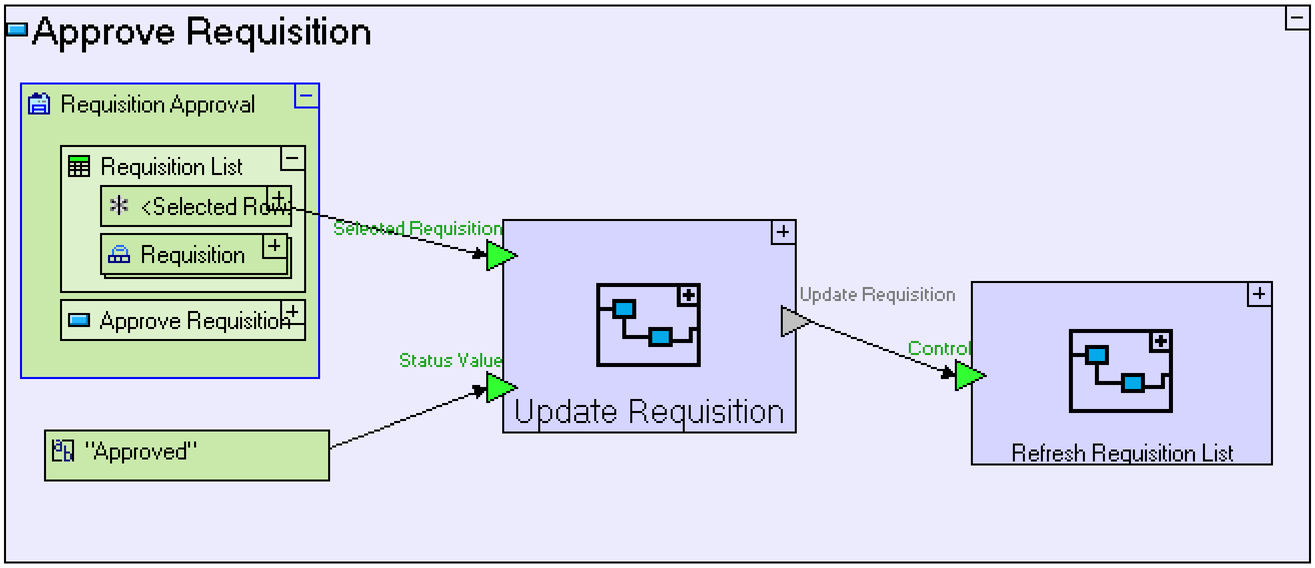
The model now looks similar to the following:



We should remove the redundant elements appearing at the top of the screenshot.

Select each of the redundant elements **Change Requisition Status**, **Update**, and the flows linking them, and delete them by pressing **[Del]**.

Now the re-factoring of **Approve Requisition** is complete and the model should look similar to the following:



Save your work and view the application in the browser. There should be no difference as to how it looks or functions.

Make sure that the **Approve Requisition** button still functions correctly.

Reuse part of the Re-factored Model (Cancel Requisition)

We shall now create the **Cancel Requisition** button in the **Open Requisitions** view which should be similar to the **Approve Requisition** button.

Zoom to the **Open Requisitions** view.

Drop a **Display/Button** template next to the **New Requisition** button.

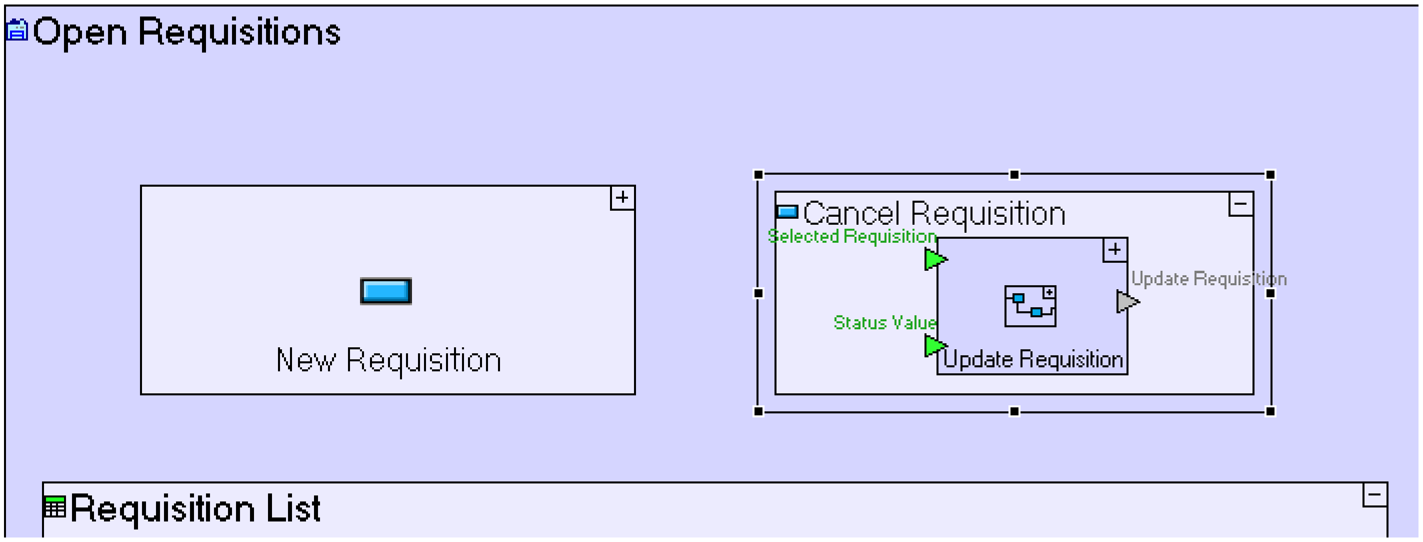
Name it ***Cancel Requisition***.

Zoom into **Cancel Requisition**.

Now create a model similar to **Approve Requisition** (use the previous screenshot as a reference), reusing the **Update Requisition** process:

Drag–and-drop the **Update Requisition** process we've just created from the outline to **Cancel Requisition**.

The **Open Requisitions** view model now looks similar to the following:

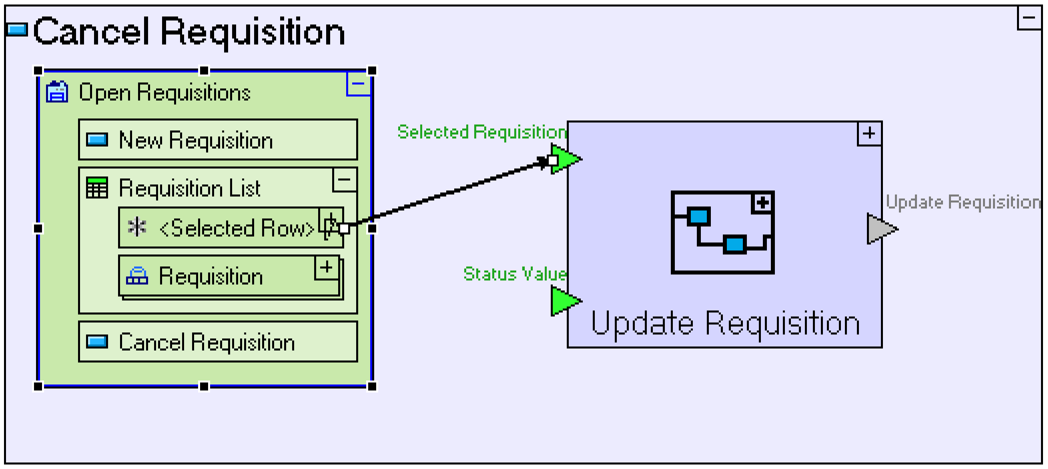


Create a reference to the view, from which the **Requisition List** is to be retrieved:

Right-click on **Cancel Requisition**, select **Add Ancestor Reference**, and choose **Open Requisitions**.

Create **Flow** linking the **Requisition List/<Selected Row>** element in the **Open Requisitions** ancestor reference to the **Update Requisition/Selected Requisition** trigger.

The **Cancel Requisition** button model now looks similar to the following:

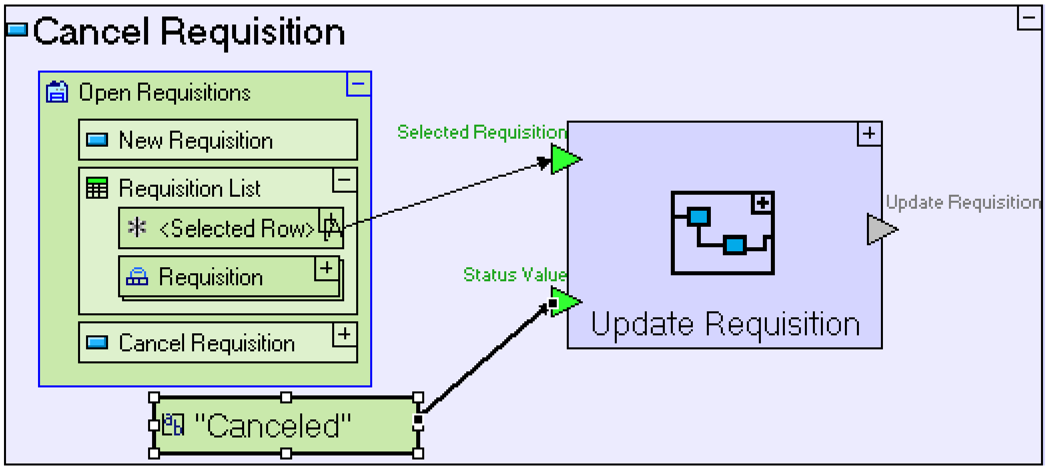


Define a new constant, **"Canceled"**:

Add a **Constants/Text** from the palette. Name it ***Canceled***.

Create Flow linking "**Canceled**" to **Update Requisition/Status Value**.

The model now looks similar to the following:

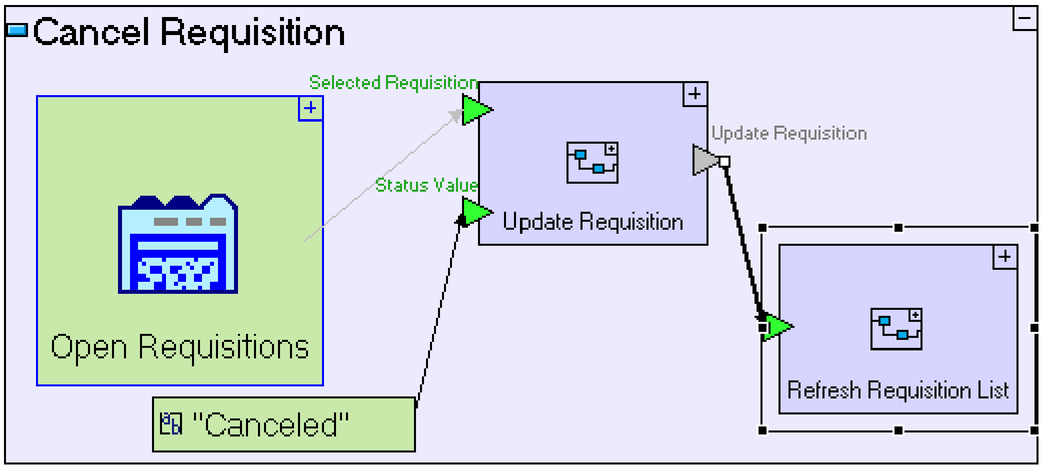


And to finish off, refresh the view following status update by reusing an existing refresh process:

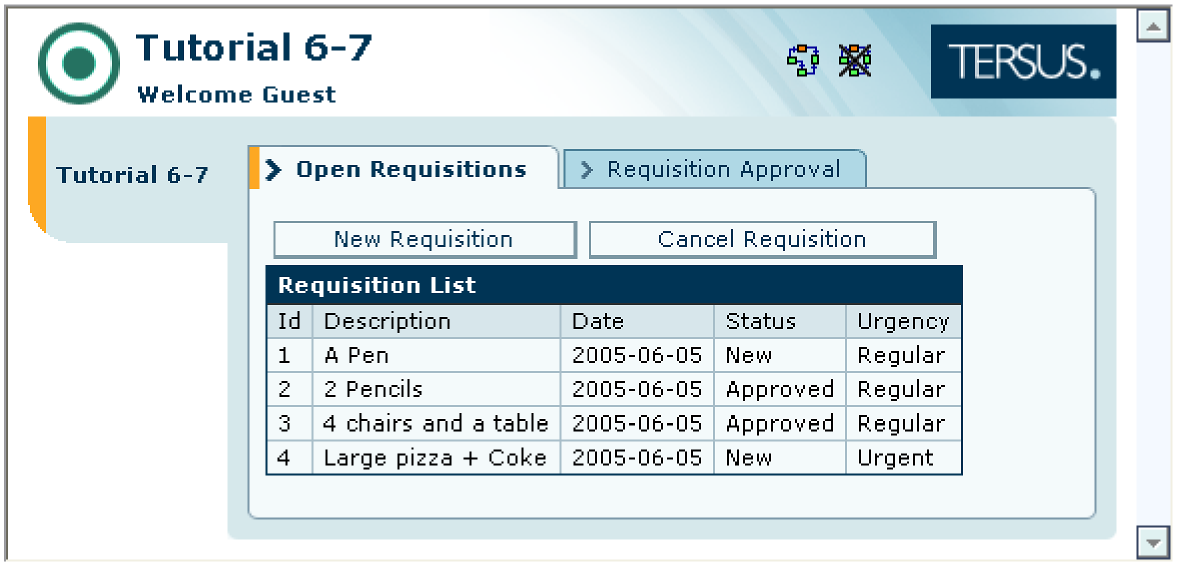
Drag-and-drop the **Refresh Requisition List** (the one used in the **Submit** button of the **Enter New Requisition** popup) from the repository (or outline) to **Cancel Requisition**.

Create **Flow** linking the **Exit** of **Update Requisition** to the trigger of **Refresh Requisition List**.

The **Cancel Requisition** button is now complete, and should look similar to the following:



Save your work and view the application in the browser, which should look similar to the following:



Requisition List

Id	Description	Date	Status	Urgency
1	A Pen	2005-06-05	New	Regular
2	2 Pencils	2005-06-05	Approved	Regular
3	4 chairs and a table	2005-06-05	Approved	Regular
4	Large pizza + Coke	2005-06-05	New	Urgent

Make sure that the **Cancel Requisition** button (in the **Open Requistion** view) functions correctly.

Completing Stage 7

Import the sample project **Tutorial 7-8** and use it as the basis for the next stage of the tutorial.

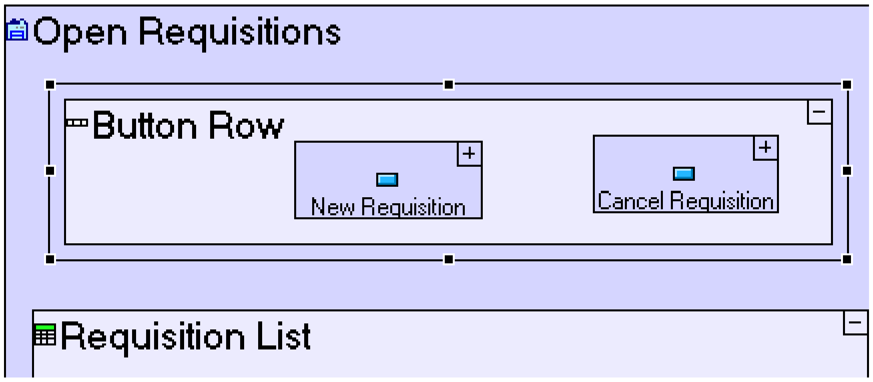
For a reminder on how to import a sample project, see the **Importing a Sample Project** section at the end of **Stage 2**.

This sample project contains all the functionality modeled thus far.

The sample project also includes additional functionality as follows:

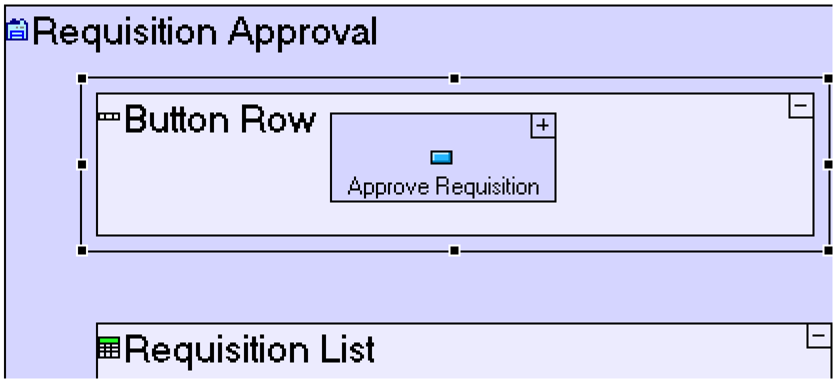
1. Add a **Button Row** to **Open Requisitions** view and place the buttons in it, for a neater

display



How to Model	Located in
Drag a **Display/Row** template. Name it ***Button Row***.	**Open Requisitions** view
Drag **New Requisition** and **Cancel Requisition** from the outline/repository	**Button Row**
Delete **New Requisition** and **Cancel Requisition** buttons	**Open Requisitions** view

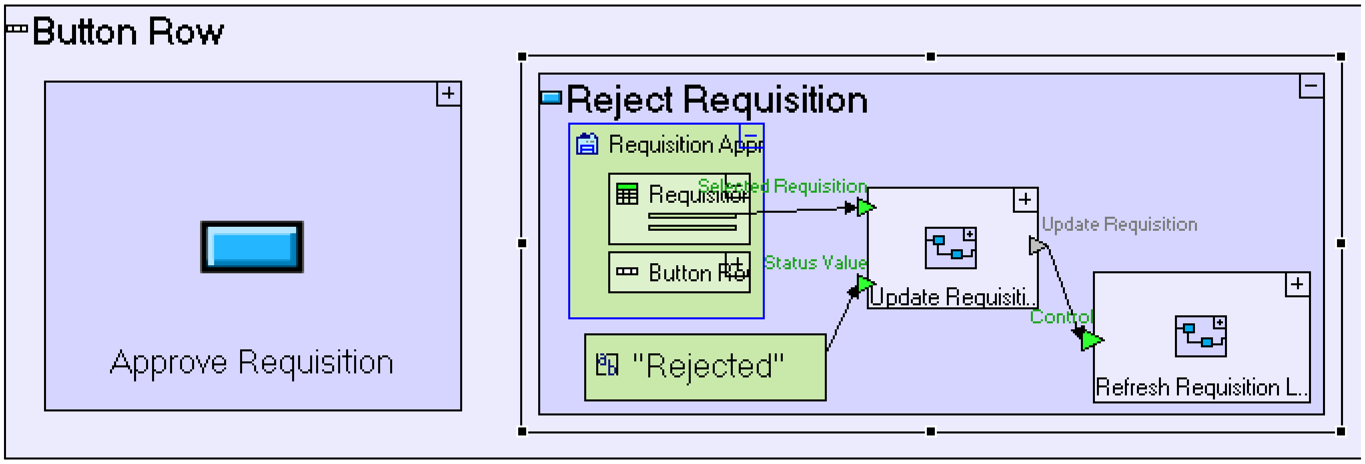
2. Add a **Button Row** to **Requisition Approval** view, for a neater display



How to Model	Located in
Drag a **Display/Row** template. Name it ***Button Row***.	**Requisition Approval** view

Drag **Approve Requisition** from the outline/repository	**Button Row**
Delete **Approve Requisition**	**Requisition Approval** view

3. Add a **Reject Requisition** button to the **Requisition Approval** view, recreating the modeling performed for **Approve Requisition** and **Cancel Requisition**



How to Model	Located in
Drag a **Display/Button** template and name it ***Reject Requisition***.	**Requisition Approval/Button Row**
Reuse the **Update Requisition** process. Add an ancestor reference to the **Requisition Approval** view. Create a new text constant "***Rejected***". Reuse the **Refresh Requisition List** process (from the **Requisition Approval** package). Create flows identical to those of **Approve Requisition**.	**Reject Requisition** button

You may now proceed to **Stage 8**, in which we shall fine-tune the retrieval of requisitions in both views to make sure that only the relevant requisitions appear in each view**.**

See It Live

Object 6

Click here to open the live project in a separate window.

Stage 8 – Filtering Retrieved Data

Stage Goals

Tersus Concepts Covered

On completion of this stage you should be familiar with the following concepts:

Modeling notions:	Flow errors, Remove Flow
Modeling techniques:	Filtering retrieved data, Clearing table display, Moving models between packages
Useful process templates:	Find (with condition), Advanced Find

Application Functionality Modeled

In this stage you will fine-tune the two views you have created (**Open Requisitions** and **Requisition Approval**).

Currently both views display all requisitions in the database, which is inconsistent with their intended user (and will make them difficult to use as more and more requisitions are entered into the system). Ideally, each view should filter requisitions and display only the requisitions which are relevant to that view:

- The **Open Requisitions** view should display all requisitions except those that have been canceled (i.e., filter requisitions based on the following condition: **Status<>Canceled**).
- The **Requisition Approval** view should only display new requisitions (**Status=New**).

This stage's modeling should be performed in the **Tutorial 7-8** project, you imported at the end of the previous stage.

User Modeling

Both views contain a process that populates the requisition list with requisitions.

We shall start with the **Populate Requisition Approval List** process.

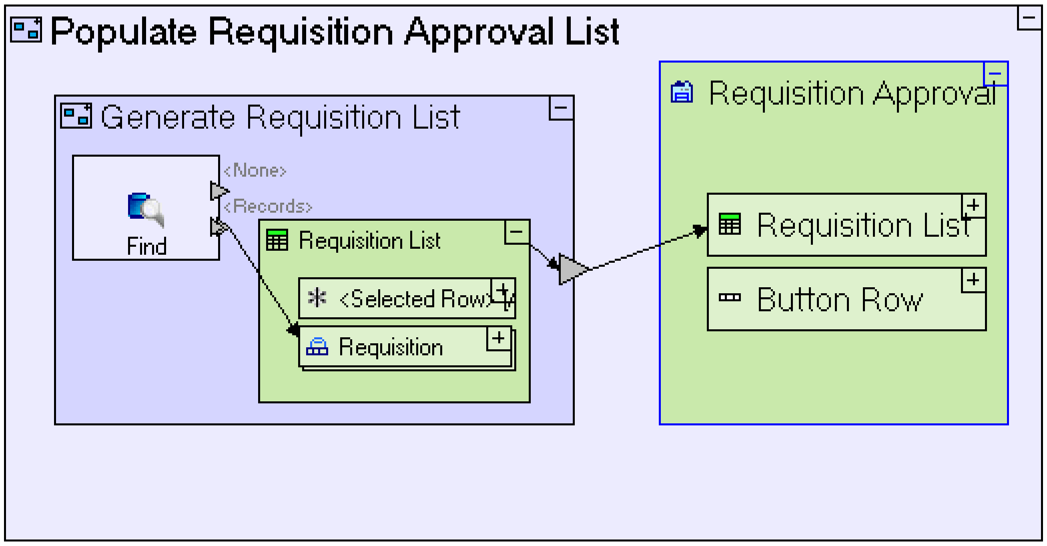
Populate Requisition Approval List is reused in the following locations:

1. **Requisition Approval** view
2. **Requisition Approval/ Approve Requisition** button
3. **Requisition Approval/ Reject Requisition** button

Because of reuse, editing and changing **Populate Requisition Approval List** in one of these locations, will be sufficient.

Add a Trigger to the Find Process (to specify a value by which to filter)

Populate Requisition Approval List currently looks as follows:



We would like the **Generate Requisition List** sub-process to return **Requisitions** whose **Status** is **New**:

Zoom to **Generate Requisition List**.

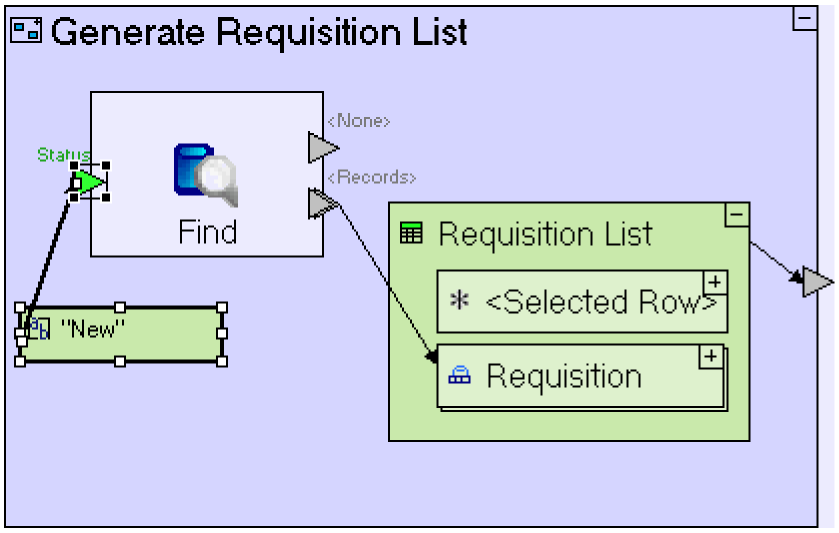
Add a **Trigger** to **Find**. Name it (pressing [F2]) ***Status***.

Drag the "**New**" constant from the repository (or outline) and drop it next to the **Status** trigger.

We want to reuse the existing **New** constant in the **Submit** button of the **Enter New Requisition** popup.

Create Flow linking the "**New**" constant to the **Find/Status** trigger.

The **Generate Requisition List** process should now look as follows:



When we add a trigger to the **Find** process and the trigger name matches a field in the table from which the records are retrieved (**Status** in our case), **Find** will only retrieve those records where the field contains the value supplied to the trigger (**"New"** in our case).

For more information regarding **Find** see the **Templates** section of the on-line documentation.

Save your work and view the application in the browser.

Switch to the **Requisition Approval** view.

Verify that the **Requisition List** is now filtered to display requisitions whose **Status** equals **New**.

Approve or reject a requisition and verify that it is removed from the **Requisition List**.

Continue approving or rejecting requisitions until one requisition is left.

Use Remove Flow (to clear the table)

If you now, approve or reject the last requisition, you will see that it is not removed from the display (although its status has changed in the database, as can be seen if you refresh the browser display – the table will appear empty).

This occurs because when **Generate Requisition List** is executed in order to refresh the requisition list, after the last requisition has been approved/rejected, the **Find** process fails to find any records whose **Status=New** and therefore exits through its **<None>** trigger (instead of the usual **<Record>** trigger). The **<None>** trigger does not have flow defined from it, and so the process terminates, which means the display is not refreshed.

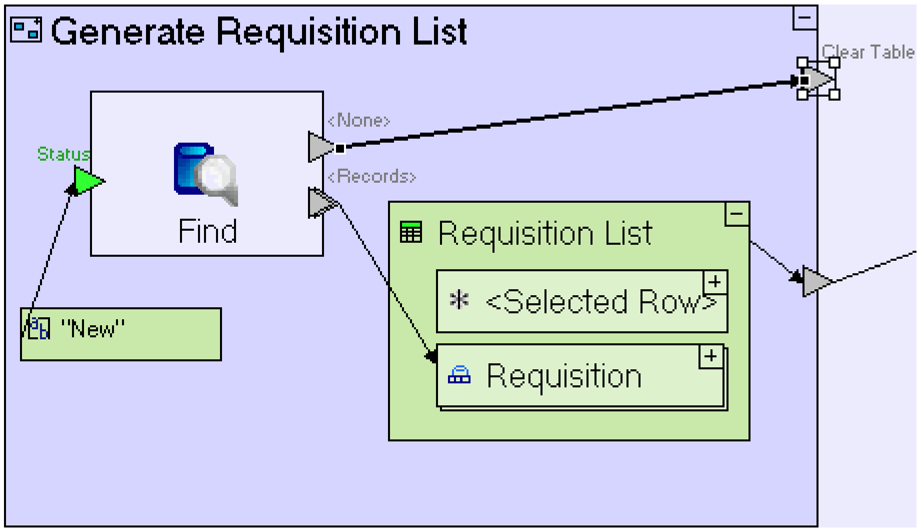
To refresh the display in this case, we need to define flow exiting **<None>** and use a special type of flow, **Remove**, which will clear the table display:

Zoom to **Generate Requisition List**.

Add an exit slot to **Generate Requisition List**. Name it ***Clear Table***.

Create flow linking the **Find/<None>** exit to the **Generate Requisition List/Clear Table** exit.

The **Generate Requisition List** model should look similar to the following:

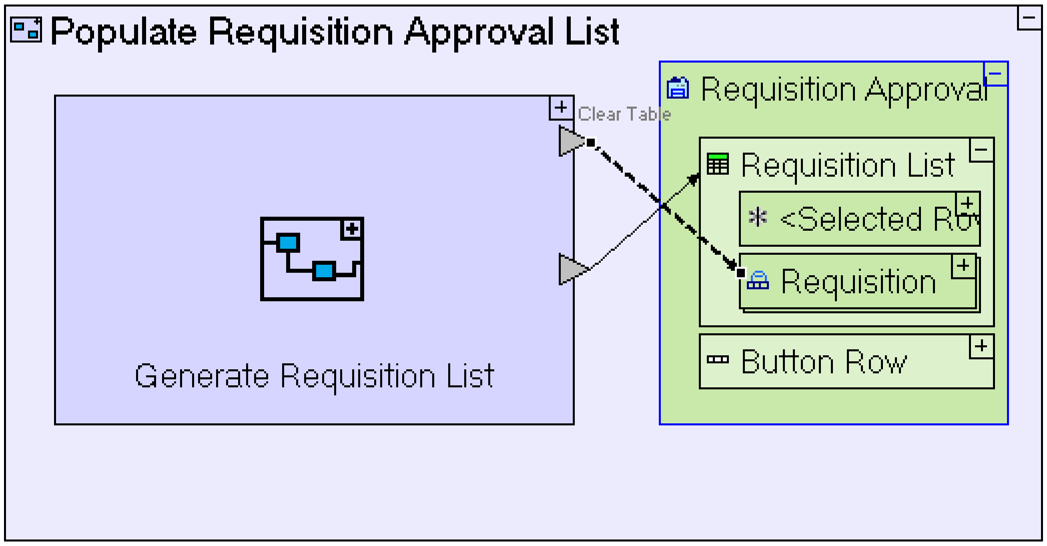


What is left to do in order to clear the display is to add the **Remove** flow tool, which is used to specify that the flow's target should be cleared of existing elements (in our case at runtime this will be the single requisition that we have just approved or rejected).

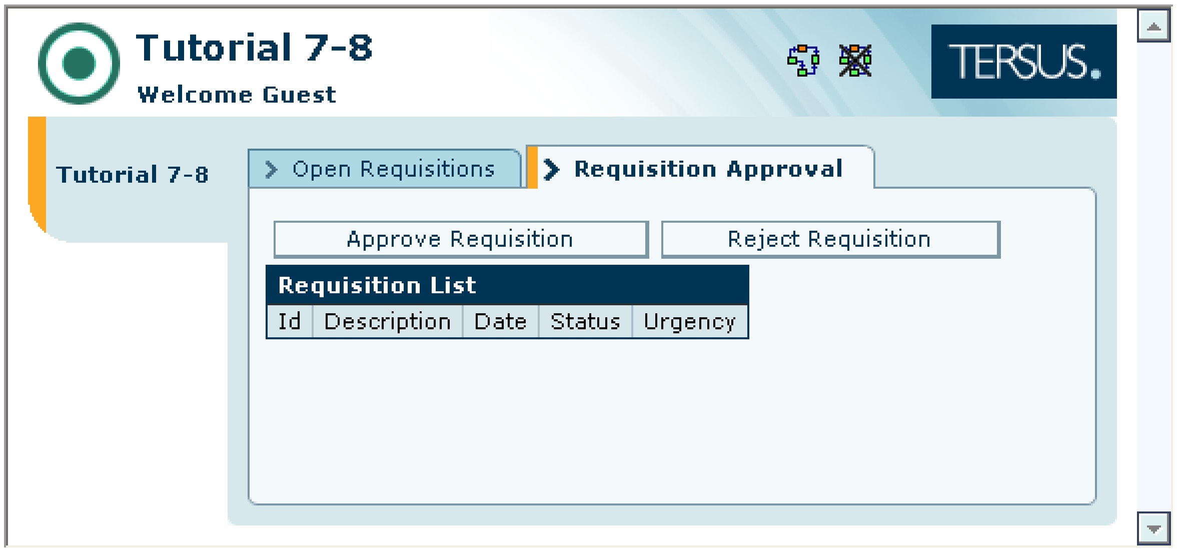
Zoom to **Populate Requisition Approval List**.

Select the **Remove** flow tool (). Click on the **Generate Requisition List/Clear Table** exit to define the source. Click on **Requisition Approval/Requisition List/Requisition** database record to define the target.

The **Populate Requisition Approval List** model should look similar to the following:



Save your work and view the application in the browser. Verify that in the case of a single requisition appearing in the table, approving/rejecting it clears the table, as in the following screenshot:



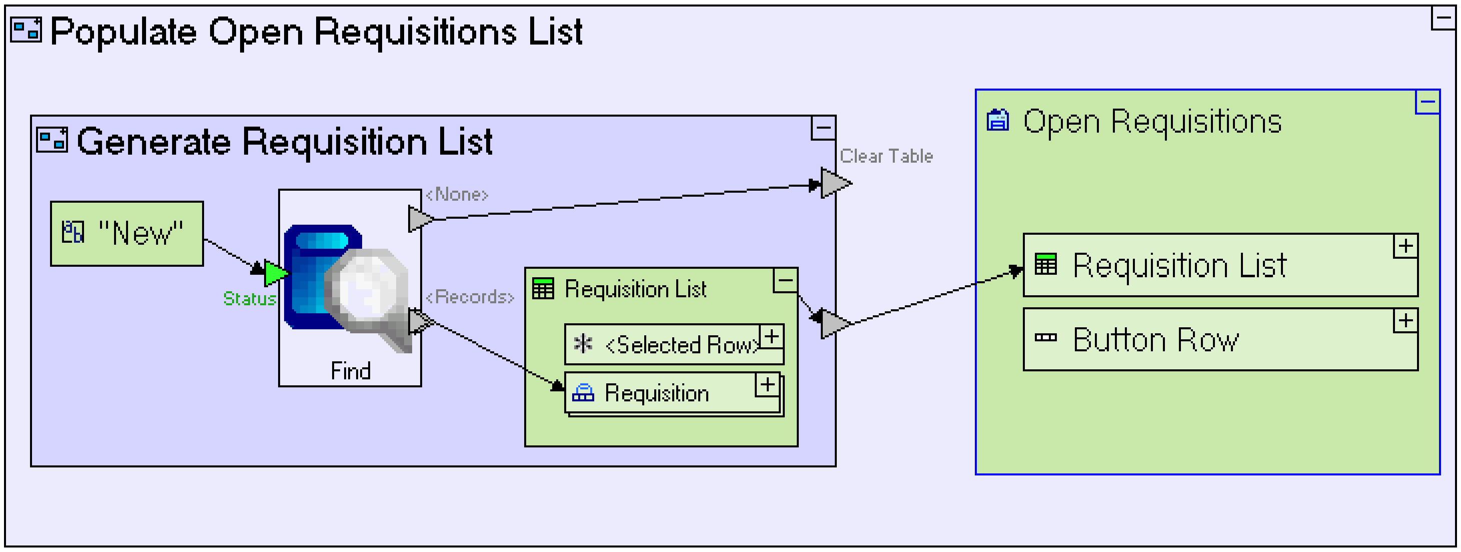
Requisition List				
Id	Description	Date	Status	Urgency

Notice that because **Generate Requisition List** is reused in the **Open Requisitions** as well it will display the same list of requisitions whose status equals **New**. We will handle this next.

Opening Models in a Separate Editor Window

The **Open Requisitions** view requires that we implement a different filter: **Status<>Canceled**.

Populate Open Requisitions List currently looks as follows:



We need to update the **Populate Open Requisitions List** process to implement the different filter, and since the actual filtering is performed in the **Generate Requisition List** sub-process, which we must not change (as it is used by **Requisition Approval**), we shall replace the existing process with a new (separate) **Generate Requisition List**.

Since we are going to recreate (in part) functionality that is already modeled in **Populate Requisition Approval List**, we can open it in a separate model editor window. This will allow us to use it as a reference for further modeling, switching back and forth between the two windows.

To open **Populate Requisition Approval List** in a separate window, you can either:

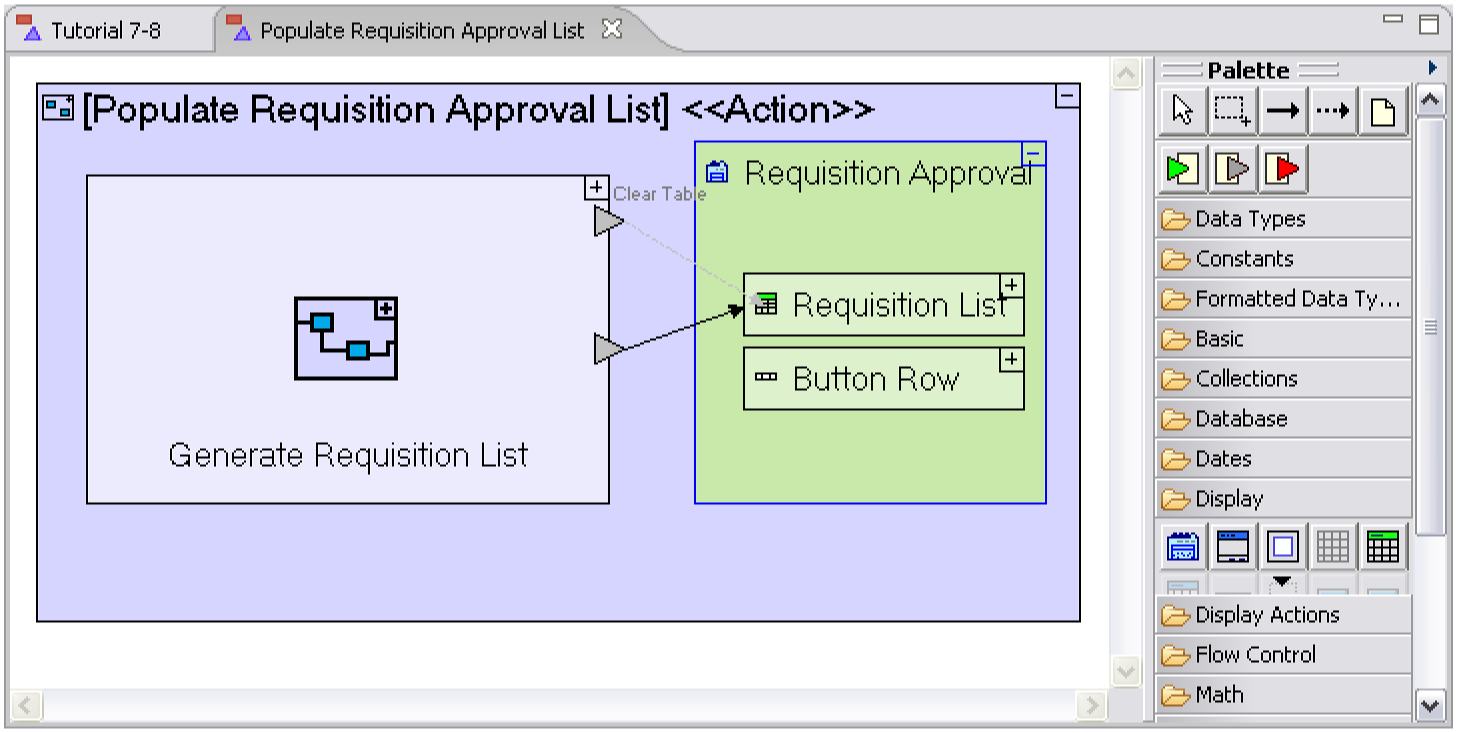
> Locate **Populate Requisition Approval List** in the **Repository Explorer**, and double-click.

Or:

> Locate **Populate Requisition Approval List** in the **Model Editor**, right-click it, and select **Open in a New Tab**

> The quickest way to locate a model in the repository is by right-clicking the model in the editor and selecting

Your model editor should now display 2 editor windows; one for the root model **Tutorial 7-8** and another for **Populate Requisition Approval List**, as in the following screenshot:



Remove an Element from the Model

Go back to editing the complete model:

> Click on the **Tutorial 7-8** editor tab.

Locate **Populate Open Requisitions List** (in **Open Requisitions**) to begin modeling the changes:

> Zoom to **Populate Open Requisitions List**.

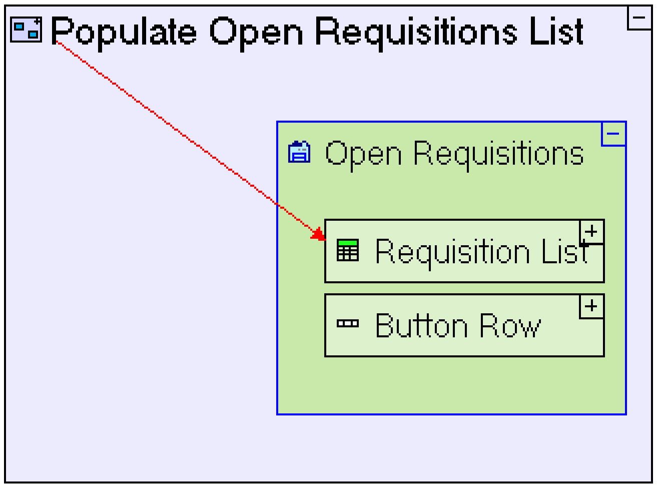
As explained before, any of the instances of the model will do (as long as they are the

Requisition Approval’s version).

Now remove the obsolete process:

> Select **Generate Requisition List** and delete it.

The model should look as follows:



The flow arrow which linked **Generate Requisition List** to the **Open Requisitions** display data element will turn red to indicate a modeling error. We shall deal with it in a moment.

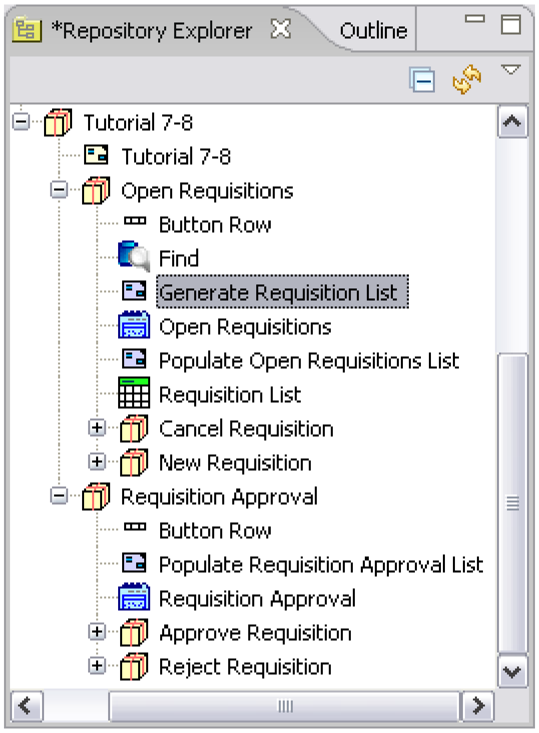
Understanding Model Packaging and Naming

Despite the fact that we deleted the **Generate Requisition List** process from the **Open Requisitions** view, meaning it remains in use only in the **Requisition Approval** view, the **Generate Requisition List** model remains in the package in which it was originally created, i.e. the **Open Requisitions** package (verify this by looking in the **Model Repository**). This has no effect on application behavior in runtime, since packaging is implemented solely for developer convenience, and a model from any package may contain elements from any other package.

There is however one interesting side effect on our subsequent modeling: We next plan to create a new **Generate Requisition List** model, similar to the existing one, in **Populate Open Requisitions List** - which means it will be created by default in the **Open Requisitions** package (the same package as the existing **Generate Requisition List** model), resulting in a new model called **Generate Requisition List 2**, which will work just the same, but in order to keep things neat and tidy, we would like to avoid this.

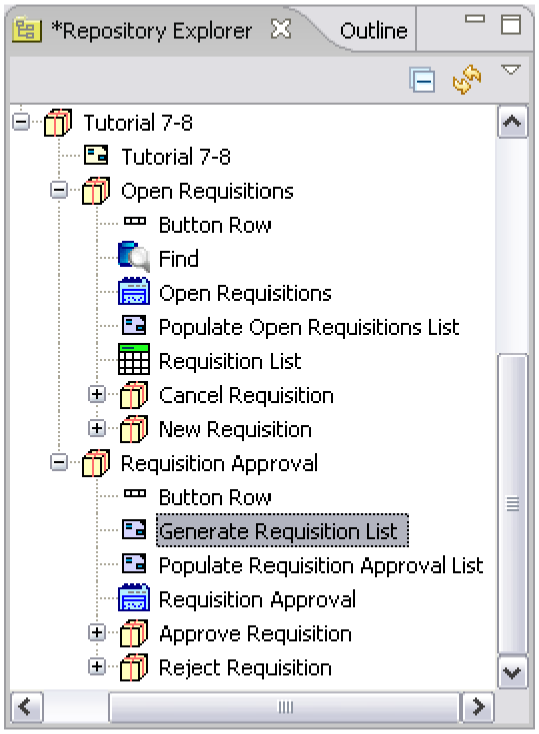
A solution to this is to move the existing **Generate Requisition List** model to the **Requisition Approval** package:

> Locate **Generate Requisition List** in the **Open Requisitions** package (see the screenshot below).



Tutorial 7-8
Tutorial 7-8
Open Requisitions
Button Row
Find
Generate Requisition List
Open Requisitions
Populate Open Requisitions List
Requisition List
Cancel Requisition
New Requisition
Requisition Approval
Button Row
Populate Requisition Approval List
Requisition Approval
Approve Requisition
Reject Requisition

Drag-and-drop **Generate Requisition List** on the **Requisition Approval** package.



Tutorial 7-8
Tutorial 7-8
Open Requisitions
Button Row
Find
Open Requisitions
Populate Open Requisitions List
Requisition List
Cancel Requisition
New Requisition
Requisition Approval
Button Row
Generate Requisition List
Populate Requisition Approval List
Requisition Approval
Approve Requisition
Reject Requisition

Now create the alternative **Generate Requisition List** process:

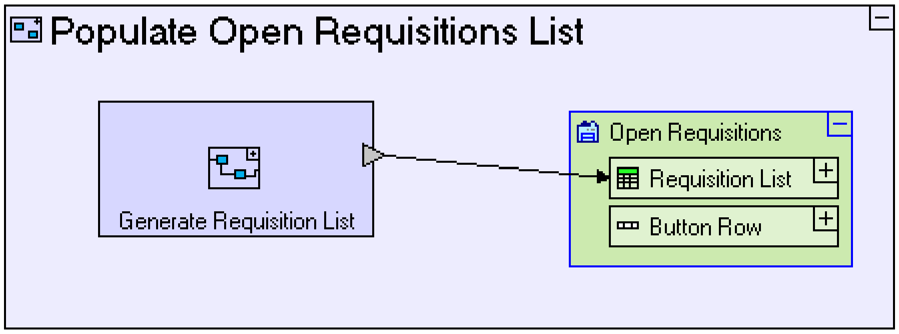
Drop a **Basic/Action** template into **Populate Open Requisitions List**. Name it ***Generate Requisition List***.

Add an **Exit** slot to the new process.

Select the red flow arrow and drag its source anchor to the new exit.

If you entered all names correctly, the red arrow will reconnect automatically, and the last step above will not be needed.

The **Populate Open Requisitions List** model should look as follows:



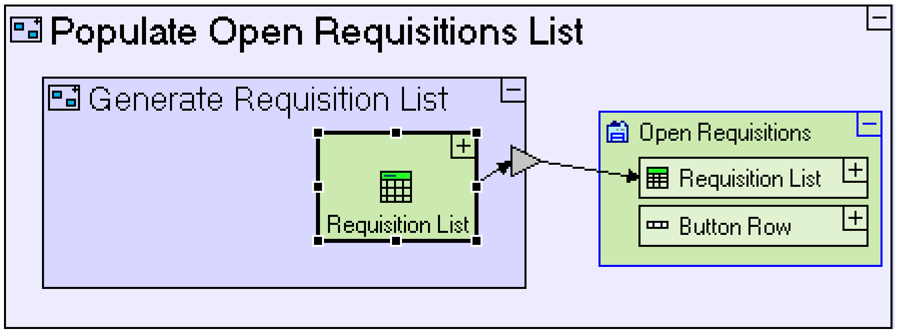
Now, let's model the new **Generate Requisition List**, by looking again at the existing **Generate Requisition List** (in the **Populate Requisition Approval List** process which we opened in a separate model editor window).

We need to add a **Requisition List** display data element:

Drag-and-drop **Requisition List** from the repository (or outline) onto **Generate Requisition List**.

Add flow linking **Requisition List** to the **Generate Requisition List** exit.

The **Populate Open Requisitions List** model should look as follows:



Use the Advanced Find template (to filter records using a complex criteria)

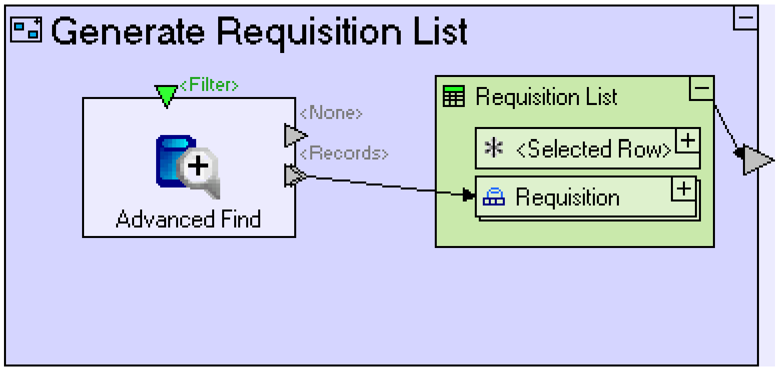
Looking once more at the **Populate Requisition Approval List** window, we can see that we're missing the **Find** process which retrieves data, but as explained earlier, we need to implement a "does not equal" criteria, which is not supported by the **Find** template. Instead we shall use the **Advanced Find** template:

Select the **Database/Advanced Find** template ( ) and drop it in **Generate Requisition List**.

Advanced Find is similar to **Find** in that it has 2 exits, **<None>** and **<Records>**, so:

Create Flow linking **Advanced Find/<Records>** exit to the **Requisitions** data structure in **Requisition List**.

Generate Requisition List should look as follows:



We want to filter out records, whose **Status** is **Canceled**.

The filter is provided through the **<Filter>** trigger of **Advanced Find**, as a text string defining constraints on one or more fields in the table. The constraint in our case should be **"Status<>'Canceled'"**.

In its simple form, **Advanced Find** relies exclusively on the default **<Filter>** trigger, but to provide additional flexibility, triggers can be added which will provide parameters for the filter itself:

Add a **Trigger** to **Advanced Find**. Name it (pressing [F2]) ***Status***.

Now the filter can be defined as "**Status<>${Status}**", where in runtime, **${Status}** will be replaced with the value passed to the **Status** trigger.

Note the use of the **$** sign and **{curly}** brackets to signify a parameter.

Define the filtering condition itself:

Add a **Constants/Text** from the palette.

Name it ***Status<>${Status}***.

Link it to the **Advanced Find/<Filter>** trigger.

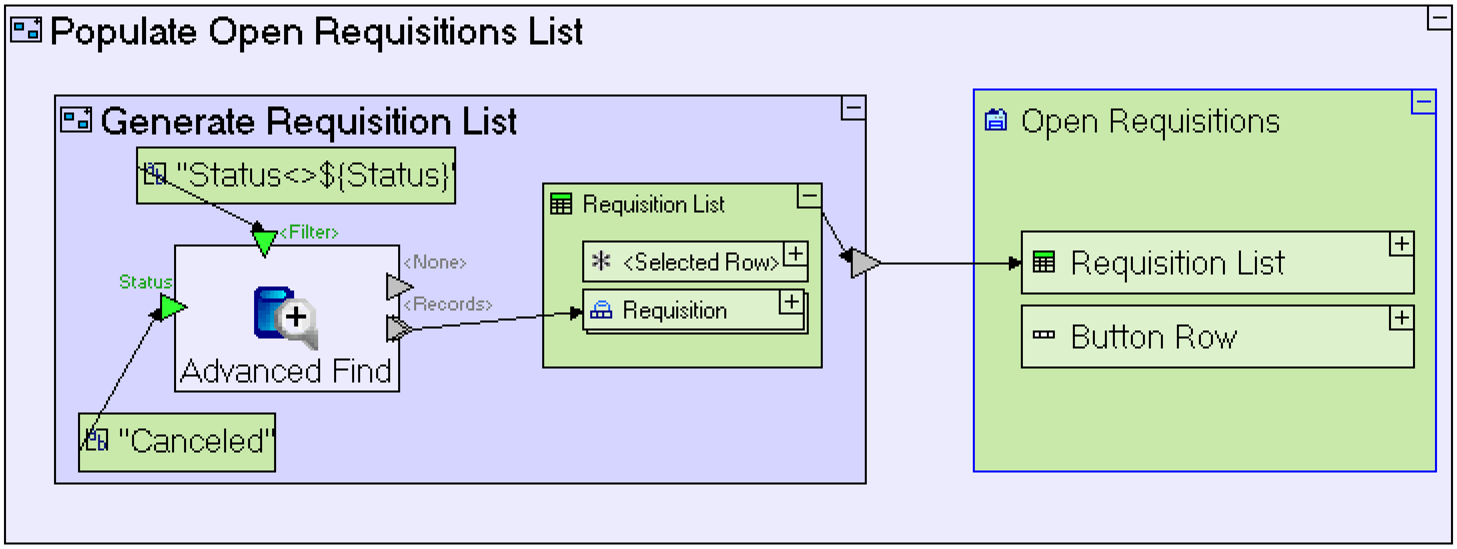
And set the parameter of the filter, **Status**:

Locate and drag the "**Canceled**" constant (from the **Cancel Requisition** button) in the repository (or outline).

Link it to the **Advanced Find/Status** trigger.

For more information regarding **Advanced Find** see the **Templates** section of the on-line documentation.

The **Populate Open Requisitions List** model should look similar to the following:



Save your work and view the application in the browser.

Verify that **Requisition List** in **Open Requisitions** does not display any requisitions with status **Canceled**.

Select a requisition from the list and press **Cancel Requisition**. The **Requisition** should disappear from the list.

This demonstrates, again, how reuse works: the changes which have been performed on the **Populate Open Requisitions List** model in one location (**Open Requisitions**), also apply to other locations where it is reused (in this case: **Open Requisitions/Cancel Requisition**).

Completing Stage 8

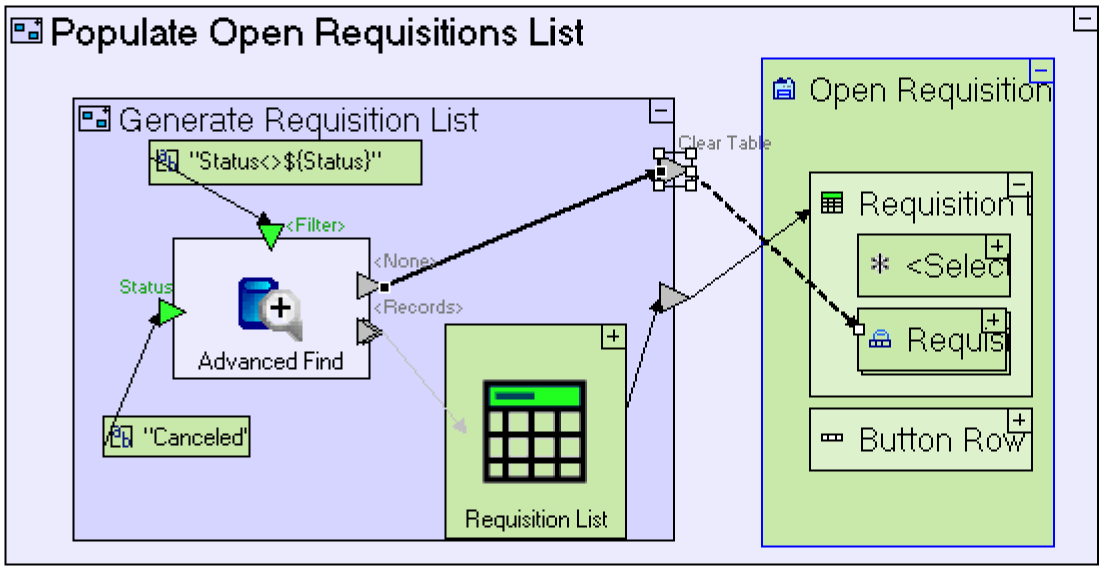
Import the sample project **Tutorial 8-9** and use it as the basis for the next stage of the tutorial.

For a reminder on how to import a sample project, see the **Importing a Sample Project** section at the end of **Stage 2**.

This sample project contains all the functionality modeled thus far.

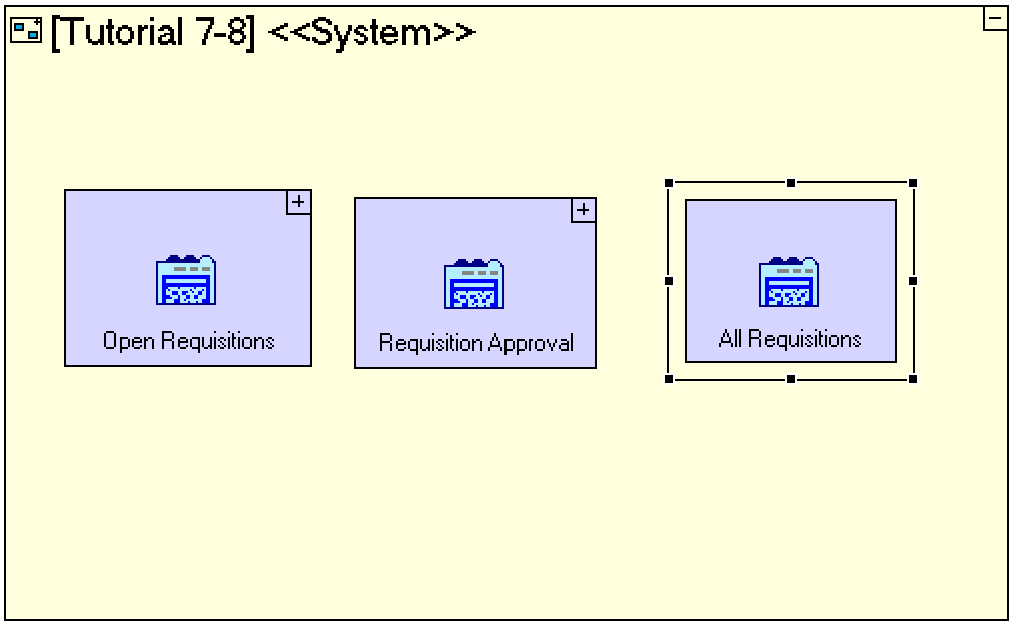
The sample project also includes additional functionality, as follows:

1. Emptying **Requisition List** display when **Populate Open Requisitions List/Generate Requisition List** does not find requisitions.



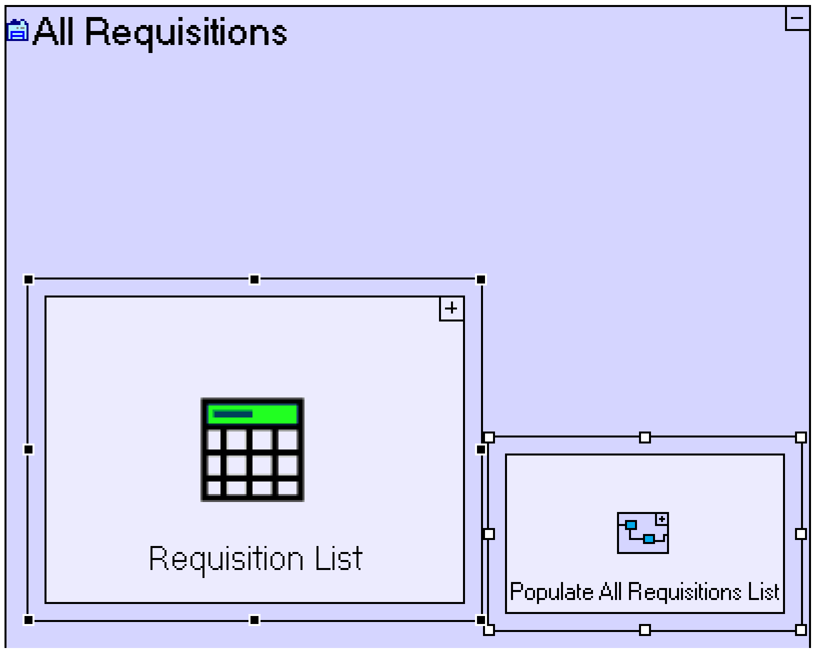
How to Model	Located in
Add **exit** slot to **Generate Requisition List**. Name it ***Clear Table***. Link **Advanced Find/<None>** to **Generate Requisition List/Clear Table**.	**Populate Open Requisitions List/Generate Requisition List** process
Create **Remove** flow linking **Generate Requisition List/Clear Table** to **Open Requisitions/Requisition List/Requisition**	**Populate Open Requisitions List** process

2. Add another view, **All Requisitions**, which displays all requisitions in the system, regardless of their Status.



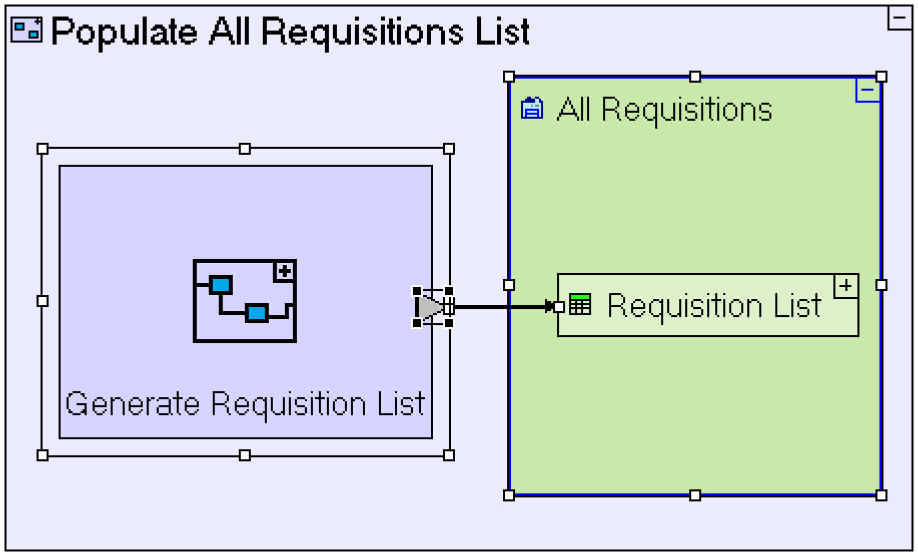
How to Model	Located in
Add **Display/View**. Name it ***All Requisitions***.	**Tutorial 7-8** root

3.**All Requisitions** view modeling (continued)



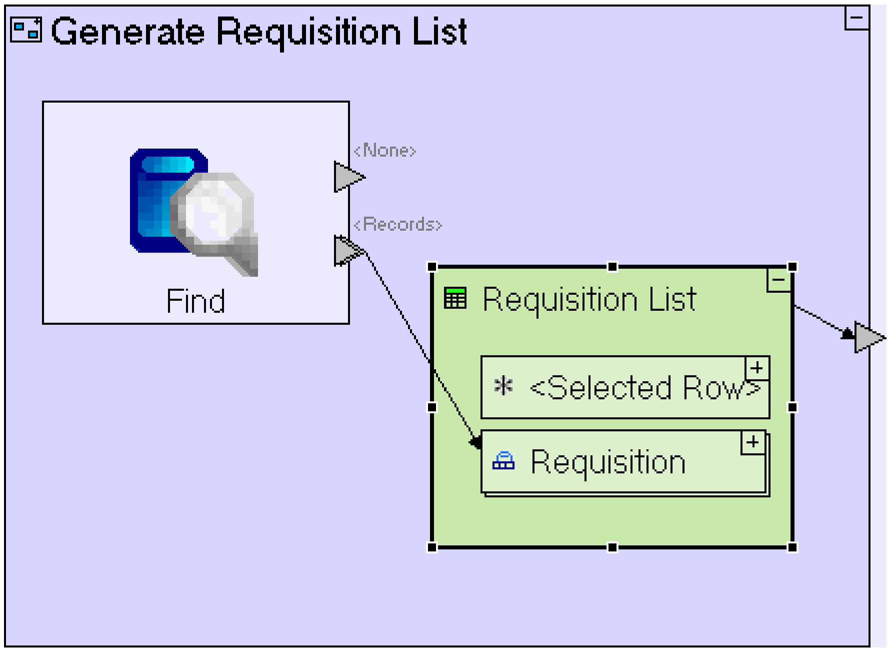
How to Model	Located in
Drag **Requisition List** simple table from the outline/repository. Add **Basic/Action**. Name it ***Populate All Requisitions List***.	**All Requisitions** view

4. **All Requisitions/Populate All Requisitions List** process modeling.



How to Model	Located in
Add an ancestor reference of the **All Requisitions** view. Add **Basic/Action**. Name it ***Generate Requisition List***. Add an **Exit** to the **Generate Requisition List** process. Link the exit to the **All Requisition/Requisition List** data structure.	**Populate All Requisitions List** process

5. **All Requisitions/Populate All Requisitions List/Generate Requisition List** process modeling.



How to Model	Located in
Add **Database/Find**. Drag **Requisition List** from the outline/repository. Link **Find/<Records>** to **Requisition List/Requisition**. Link **Requisition List** to the **Generate Requisition List** exit.	**Generate Requisition List** process

You may now proceed to **Stage 9**, in which we are going to visually re-arrange the views we have created so far into groups targeted at the different users of the system**.**

See It Live

Click here to open the live project in a separate window.

Stage 9 – Arranging Views into Perspectives

Stage Goals

Tersus Concepts Covered

On completion of this stage you should be familiar with the following concepts:

Modeling notions:	Perspectives
Modeling techniques:	Grouping views together
Useful display templates:	System (used to define perspectives)

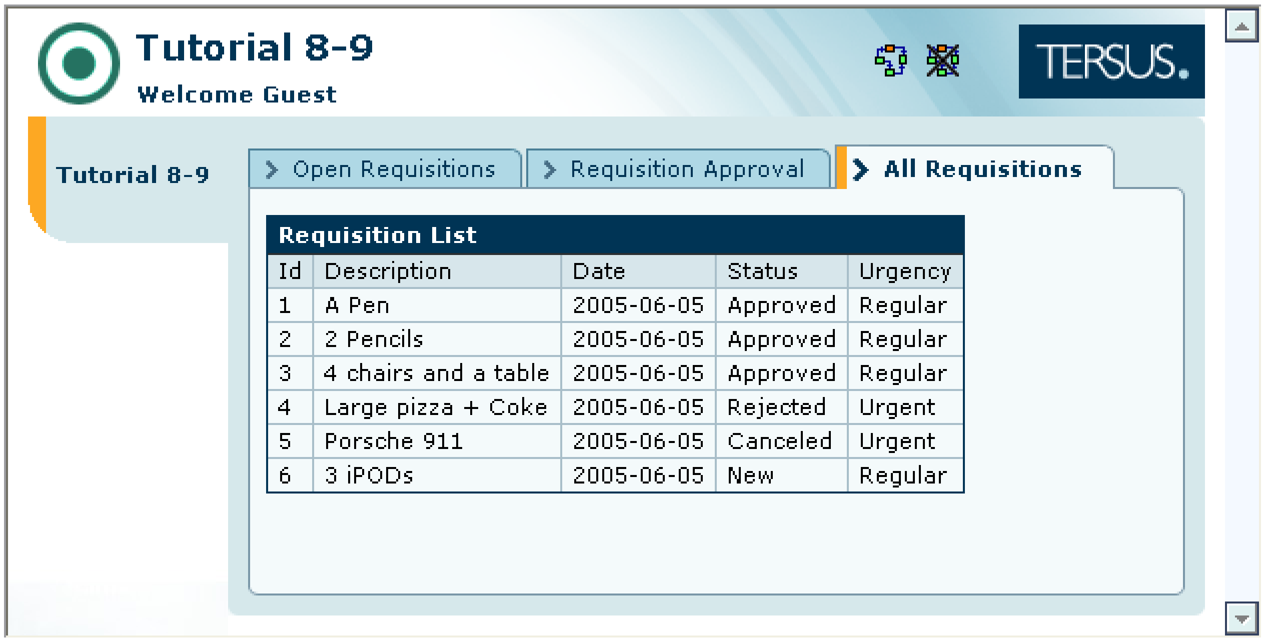
Application Functionality Modeled

In the following stages we are going to model more views, in addition to the three already modeled. The additional views will be used by the purchaser and the shipping clerk. It would be a good idea to organize and group the views according to the different roles of users: Employee, Manager, and Purchaser.

This stage's modeling should be performed in the **Tutorial 8-9** project, you imported at the end of the previous stage.

User Modeling

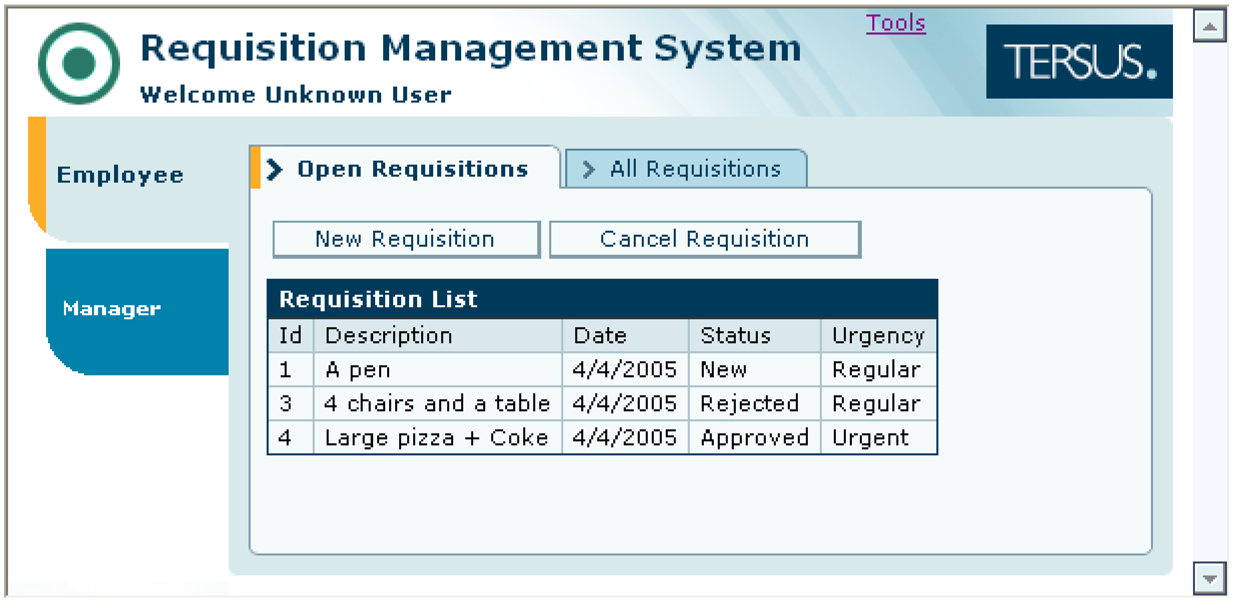
The application currently looks as follows:



Requisition List

Id	Description	Date	Status	Urgency
1	A Pen	2005-06-05	Approved	Regular
2	2 Pencils	2005-06-05	Approved	Regular
3	4 chairs and a table	2005-06-05	Approved	Regular
4	Large pizza + Coke	2005-06-05	Rejected	Urgent
5	Porsche 911	2005-06-05	Canceled	Urgent
6	3 iPODs	2005-06-05	New	Regular

The three views, **Open Requisitions**, **Requisition Approval**, and **All Requisitions**, are grouped together. We would like to group them so that **Open Requisitions** and **All Requisitions** (used by all employees) are separated from **Requisition Approval** (used by managers), as in the following screenshot:



Requisition List

Id	Description	Date	Status	Urgency
1	A pen	4/4/2005	New	Regular
3	4 chairs and a table	4/4/2005	Rejected	Regular
4	Large pizza + Coke	4/4/2005	Approved	Urgent

The main visual difference between the two screen shots is the use of multiple **Perspectives**, appearing as tabs vertically on the left side of the screen (two in this case: **Employee** and **Manager**).

Perspectives are used to group together multiple views. The default behavior, for views not defined in a perspective, is to group all views in a single perspective whose name is the root model name (in our case, **Tutorial 8-9** - as demonstrated in the first screen shot).

Add a Perspective (Employee)

Do the following to define the **Employee** perspective:

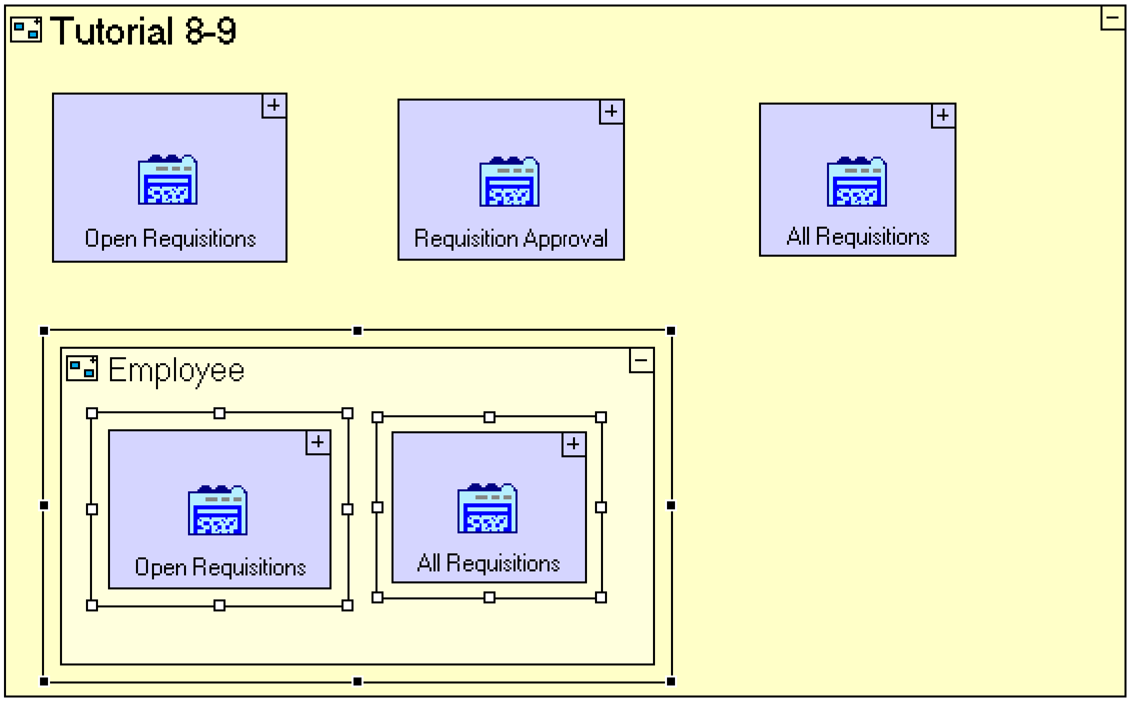
Zoom to the **Tutorial 8-9** root model.

Add a **Basic/System** template (). Name it ***Employee***.

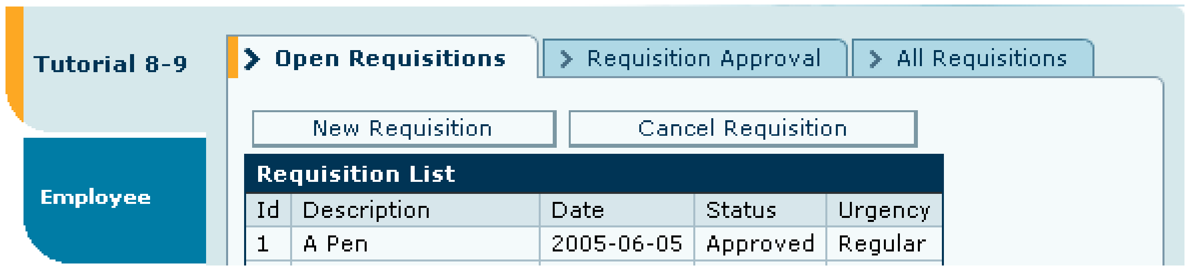
Zoom into the **Employee** system.

Drag the **Open Requisitions** view and the **All Requisitions** view from the repository (or outline) into the **Employee** system.

The model should look as follows.



Save your work and view the application in the browser. The application should look as follows:



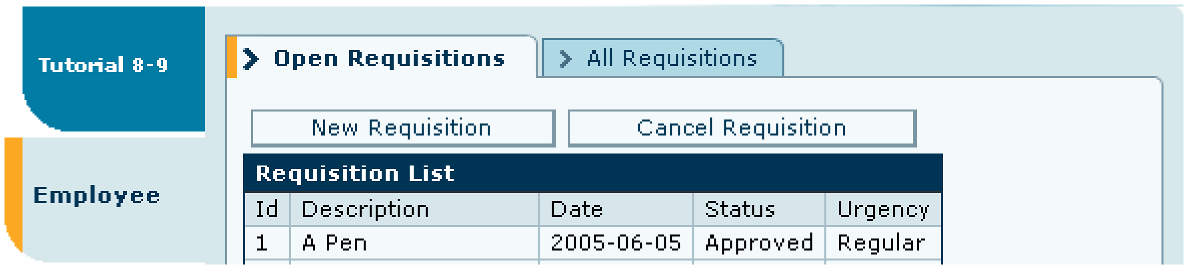
Requisition List

Id	Description	Date	Status	Urgency
1	A Pen	2005-06-05	Approved	Regular

We have not removed the three views from the root model; therefore they still appear in a default perspective named after the root model.

Now, click on the new **Employee** perspective.

You should see the following:



Requisition List

Id	Description	Date	Status	Urgency
1	A Pen	2005-06-05	Approved	Regular

The **Employee** perspective is defined correctly.

The application model as it is now defined demonstrates the fact that views, as with other model types may be reused in an application (in the case of views, reusing across perspectives).

Remove the Default Perspective

To remove the default perspective, **Tutorial 8-9**, do the following:

Zoom to the **Tutorial 8-9** root model.

Select each of the three views **Open Requisitions**, **Requisition Approval**, and **All Requisitions** and delete them from the view.

We have deleted the **Requisition Approval** view from its parent model. This is not a problem, since the **Requisition Approval** model remains in the repository.

Add an additional Perspective (Manager)

Last, let’s create the **Manager** perspective.

Do the following to define the **Manager** perspective:

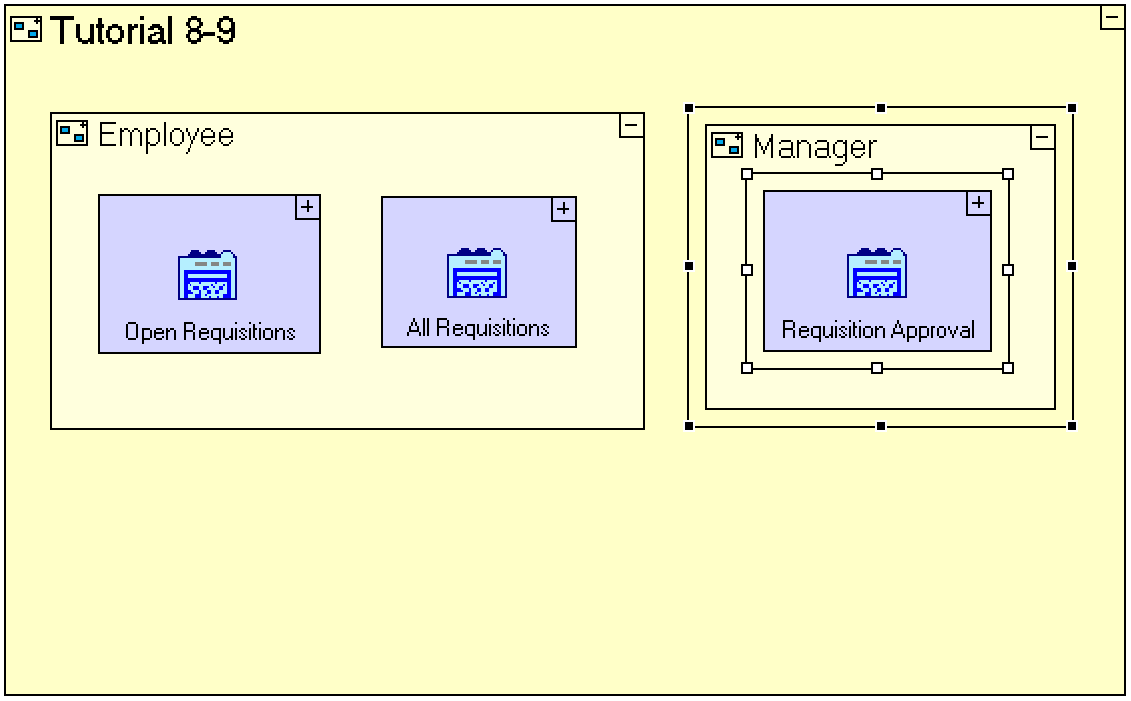
Zoom to the **Tutorial 8-9** root model.

Add a **Basic/System** template. Name it ***Manager***.

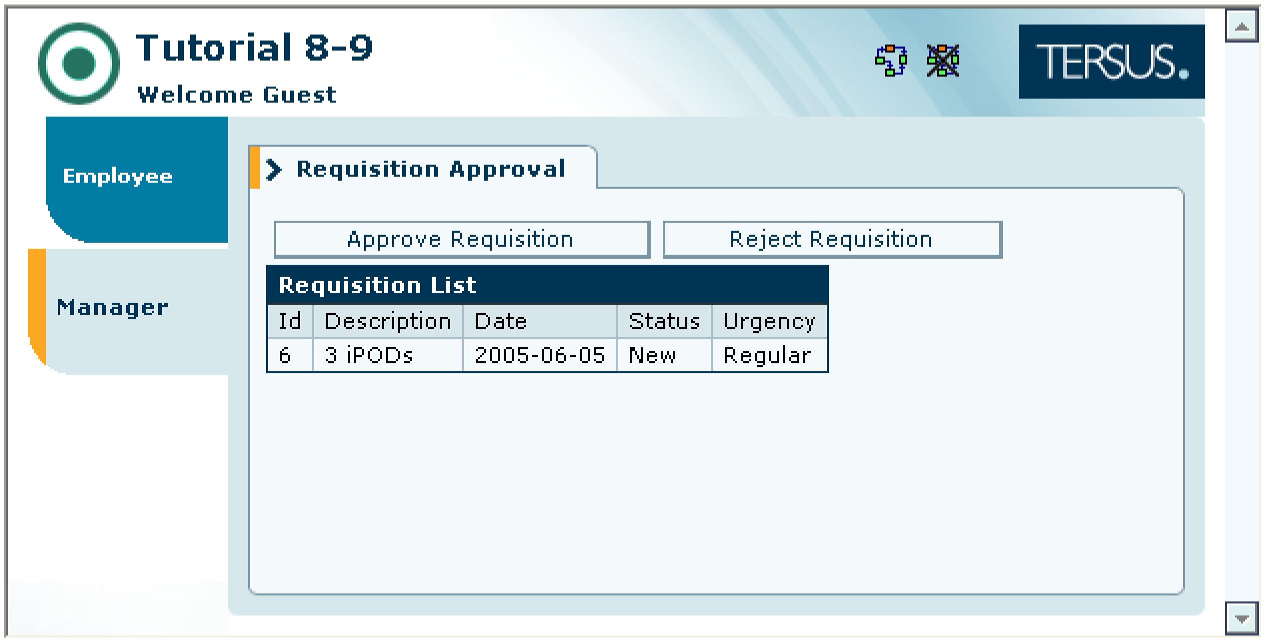
Zoom into the **Manager** system.

Drag the **Requisition Approval** view from the repository into the **Manager** system.

The model should look as follows.



Save your work and view the application in the browser. The application should look as follows:



Requisition List

Id	Description	Date	Status	Urgency
6	3 iPODs	2005-06-05	New	Regular

The grouping of views into perspectives may later be used in conjunction with the built-in support for a User/Permissions system to specify which perspectives are displayed after a user logs in to the system.

Completing Stage 9

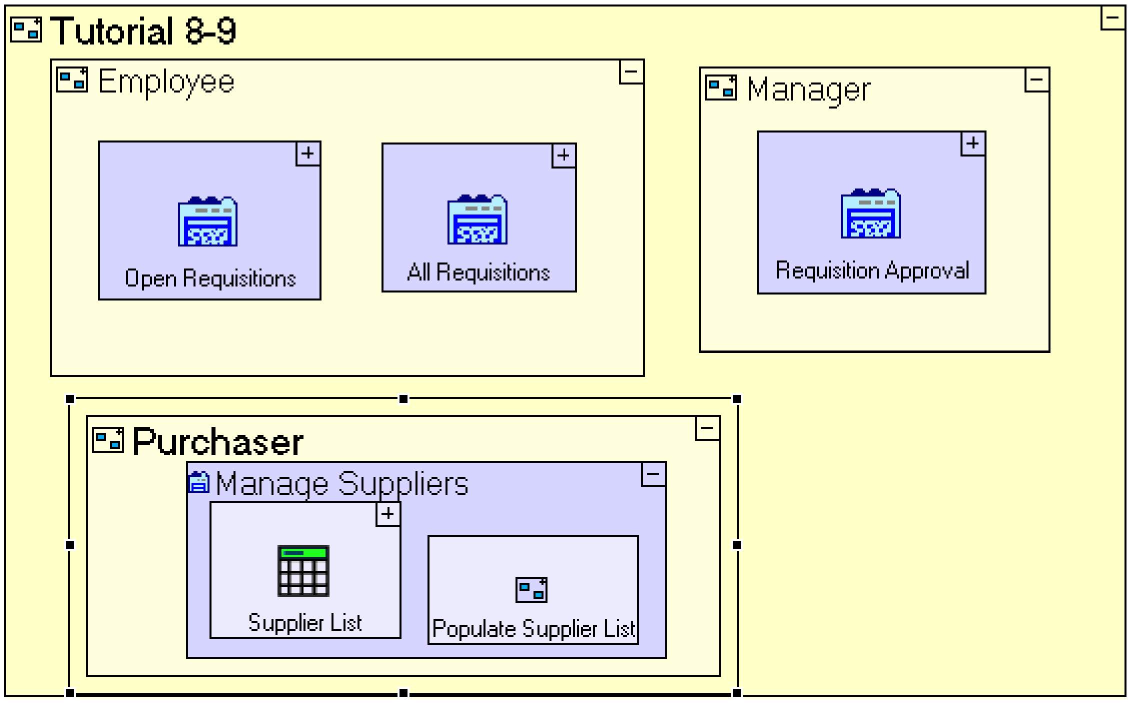
Import the sample project **Tutorial 9-10** and use it as the basis for the next stage of the Tutorial 8-9.

For a reminder on how to import a sample project, see the **Importing a Sample Project** section at the end of **Stage 2**.

This sample project contains all the functionality modeled thus far.

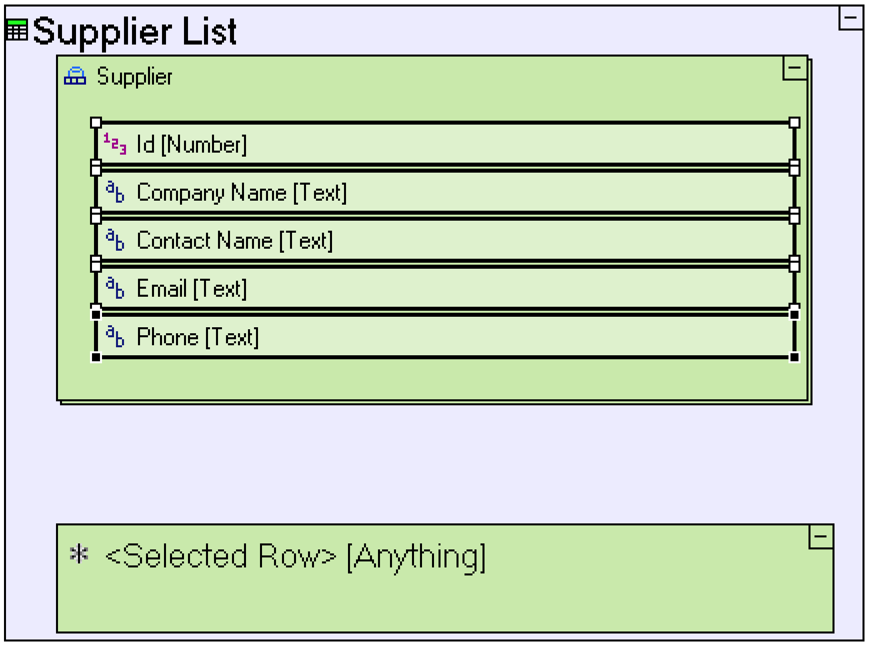
The sample project also includes additional functionality as follows:

1. Add a new **Purchaser** perspective and a **Manage Suppliers** view, containing a table and a "Populate" process.



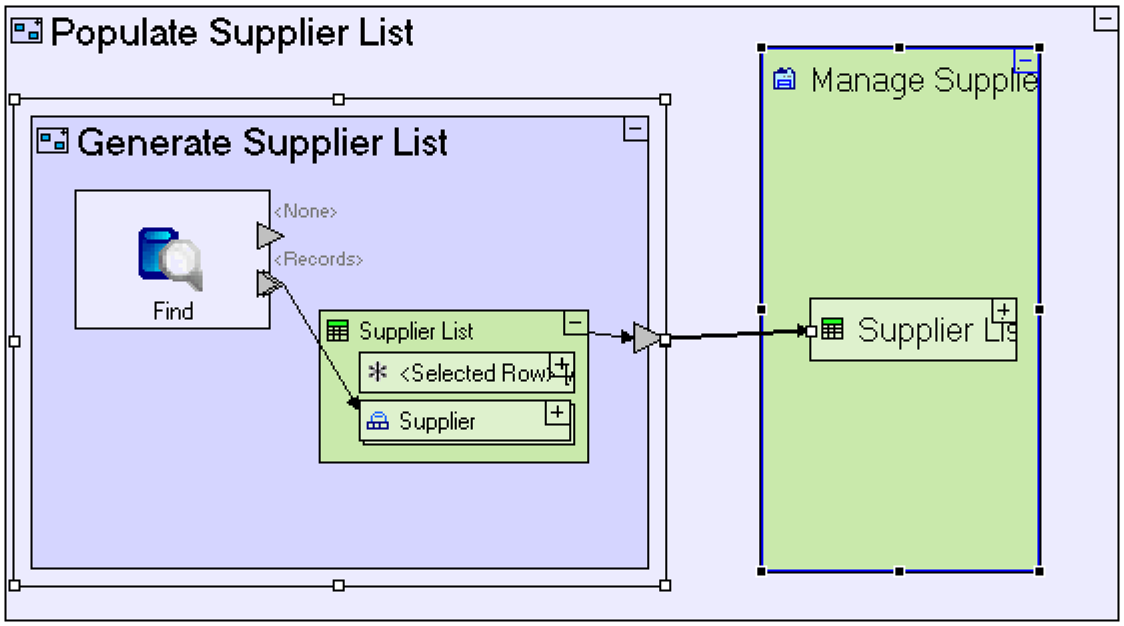
How to Model	Located in
Add a **Basic/System**. Name it ***Purchaser***.	**Tutorial 8-9** root
Add a **Display/View**. Name it ***Manage Suppliers***.	**Purchaser** system
Add a **Display/Simple Table**. Name is ***Supplier List***. Add a **Basic/Action**. Name it ***Populate Supplier List***.	**Manage Suppliers** view

2. Define a new database table, **Supplier**.



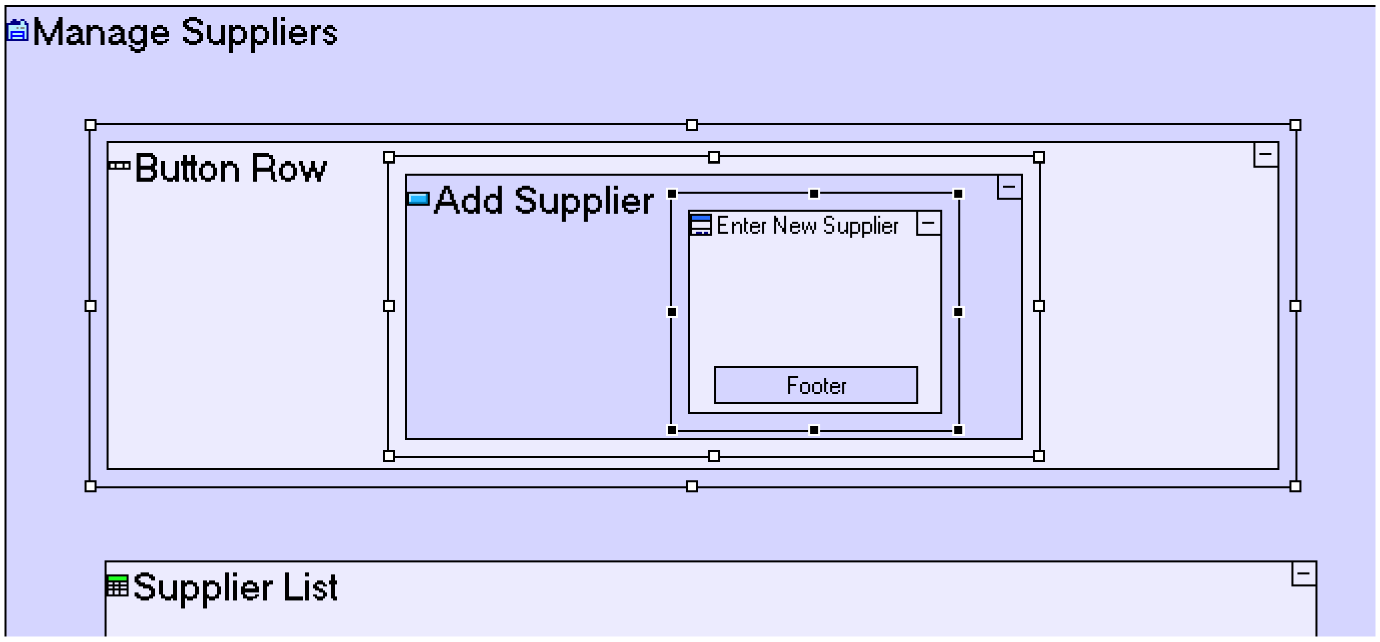
How to Model	Located in
Add a **Data Types/Database Record**. Name it ***Supplier***. Set it to **repetitive**.	**Supplier List** table
Add the following fields (data type in parentheses): ***Id* (Number)** ***Company Name* (Text)** ***Contact Name* (Text)** ***Email* (Text)** ***Phone* (Text)**	**Supplier** data structure

3. Model the **Populate Supplier List** process to initiate the display of the suppliers list



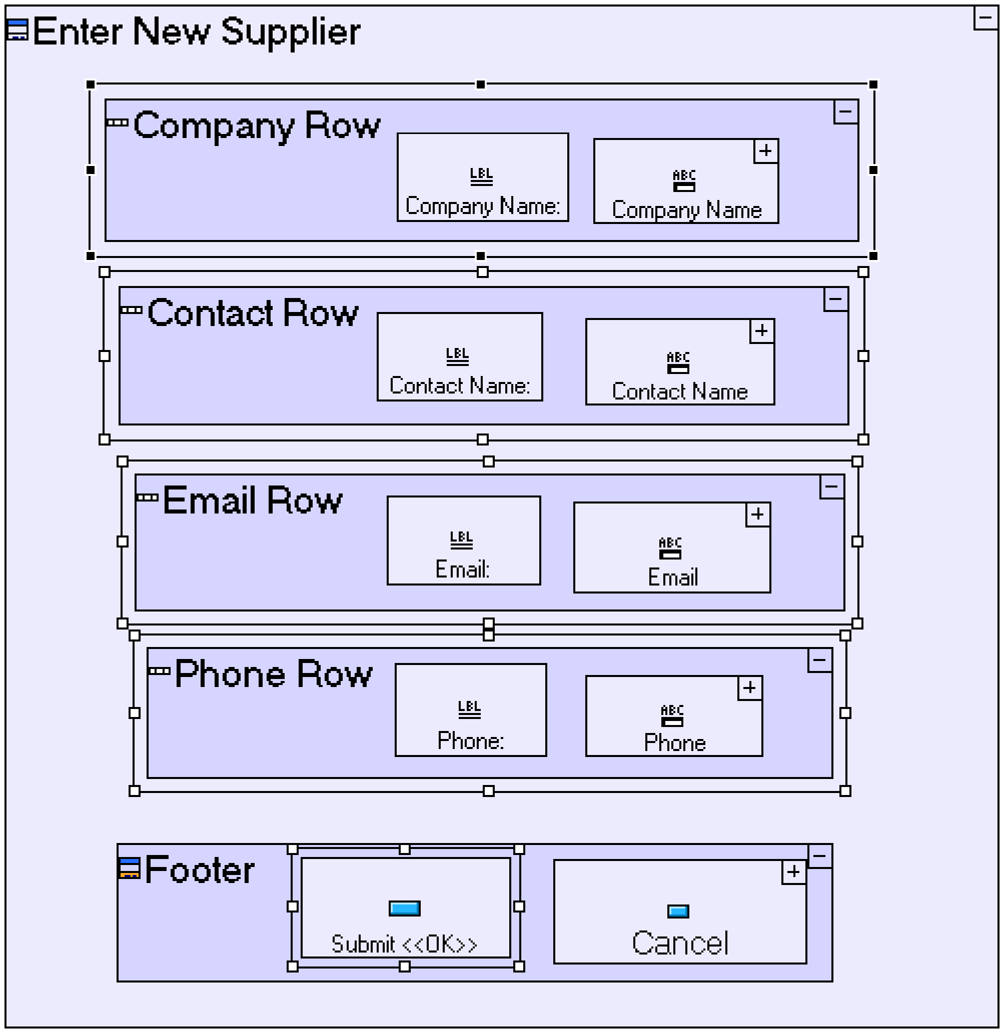
How to Model	Located in
Add **Basic/Action**. Name it ***Generate Supplier List***. Add an exit to it. Add an ancestor reference of **Manage Suppliers** view. Link the **Generate Supplier List** exit to **Manage Suppliers/Supplier List**.	**Populate Supplier List** process
Add **Database/Find**. Reuse the **Supplier List** display as a data structure by dragging it from repository/outline. Link **Find/<Records>** to **Supplier List/Supplier**. Link **Supplier List** to the **Generate Supplier List** exit	**Generate Supplier List** process

4. To **Manage Suppliers** view, add a **Button Row** with an **Add Supplier** button containing a **Enter New Supplier** popup.



How to Model	Located in
Add **Display/Row**. Name it ***Button Row***.	**Manage Suppliers** view
Add **Display/Button**. Name it ***Add Supplier***.	**Button Row**
Add **Display/Popup**. Name it ***Enter New Supplier***.	**Add Supplier** button

5. Add display elements to the **Enter New Supplier** popup.

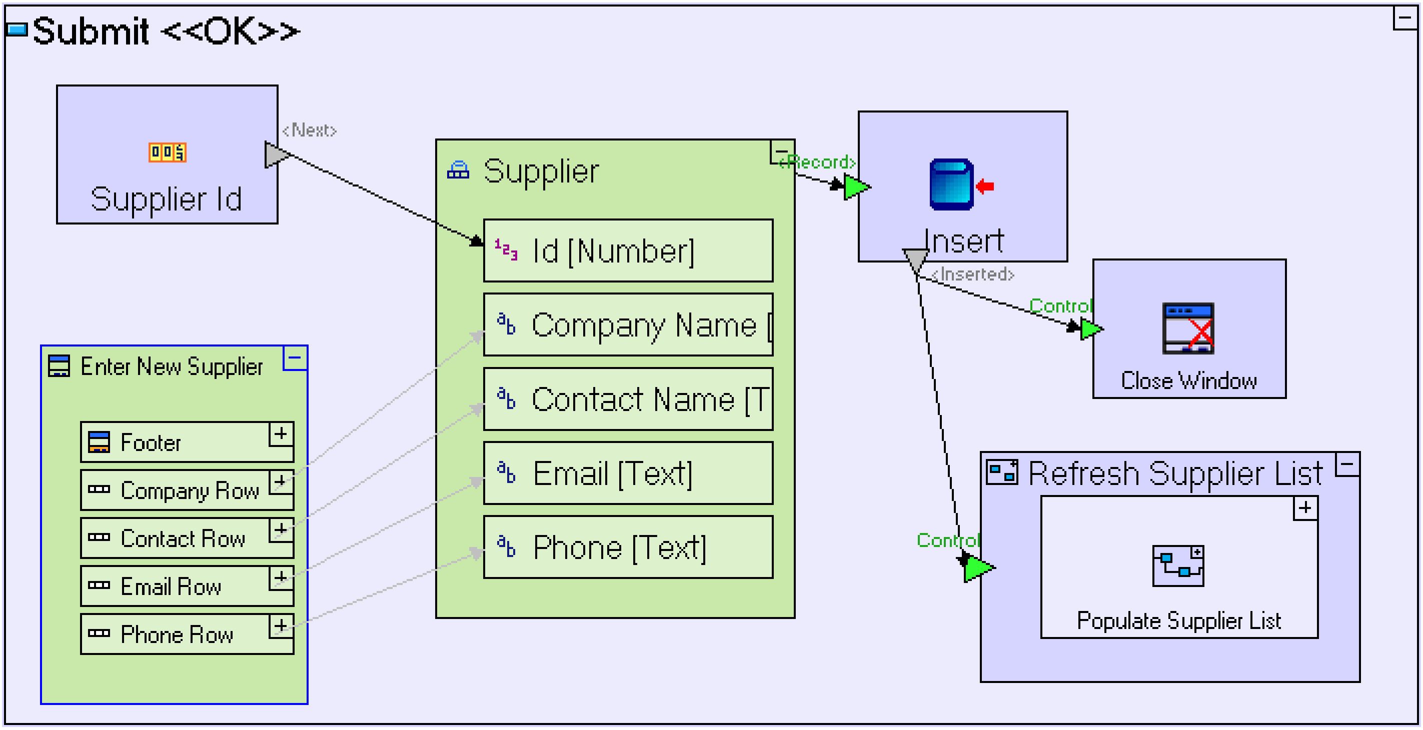


How to Model	Located in

Add the following display elements (template in parentheses):
Company Row **(Row)**
    Company Name: **(Label)**
    Company Name **(Text Input Field)**
Contact Row **(Row)**
    Contact Name: **(Label)**
    Contact Name **(Text Input Field)**
Email Row **(Row)**
    Email: **(Label)**
	Enter New Supplier popup

Email **(Text Input Field)** ***Phone Row*** **(Row)** ***Phone:*** **(Label)** ***Phone*** **(Text Input Field)**	
Rename **OK** button to **Submit**	**Footer** footer

6. Implement the **Enter New Supplier/Submit** button (similar to the **Enter New Requisition/Submit** button we created in the beginning of the tutorial).



How to Model	Located in
Add an ancestor reference of **Enter New Supplier** popup. Add a **Database/Sequence Number**. Name it ***Supplier Id***. Reuse the **Supplier** database record from the repository/outline. Link **Supplier Id/<Next>** to **Supplier/Id** Add the following flows (source in **Enter New**	**Submit** button

Supplier/Supplier List ancestor reference, target in **Supplier** data structure): • **Company Name/.value** to **Company Name** • **Contact Name/.value** to **Contact Name** • **Phone Number/.value** to **Phone** • **Email Address/.value** to **Email** Add **Database/Insert**. Link **Supplier** data structure to **Insert/<Record>** Add **Basic/Action**. Name it ***Refresh Supplier List***. Add a **Control** trigger to it. Link **Insert/<Inserted>** to the **Refresh Supplier List/Control**. Add **Display Actions/Close Window**. Add a **Control** trigger to it. Link **Insert/<Inserted>** to the **Close Window/Control**.	
Reuse **Populate Supplier List** from the outline/repository.	**Refresh Supplier List** process.

You may now proceed to **Stage 10**, in which we are going to model a process which imports a list of suppliers from an excel worksheet into the database.

See It Live

Object 8

Click here to open the live project in a separate window.

Stage 10 – Importing Data from Excel

Stage Goals

Tersus Concepts Covered

On completion of this stage you should be familiar with the following concepts:

Modeling techniques:	Importing data, Concatenating text values, Validation
Useful process templates:	Service, Load Excel Table, Concatenate
Useful display templates:	File Input Field

Application Functionality Modeled

In this stage we shall move on to handle the Purchaser's view of the application.

The purchaser's job is to review requisitions, to get quotes for prices and to issue Purchase Orders (POs) for the items specified in the requisitions.

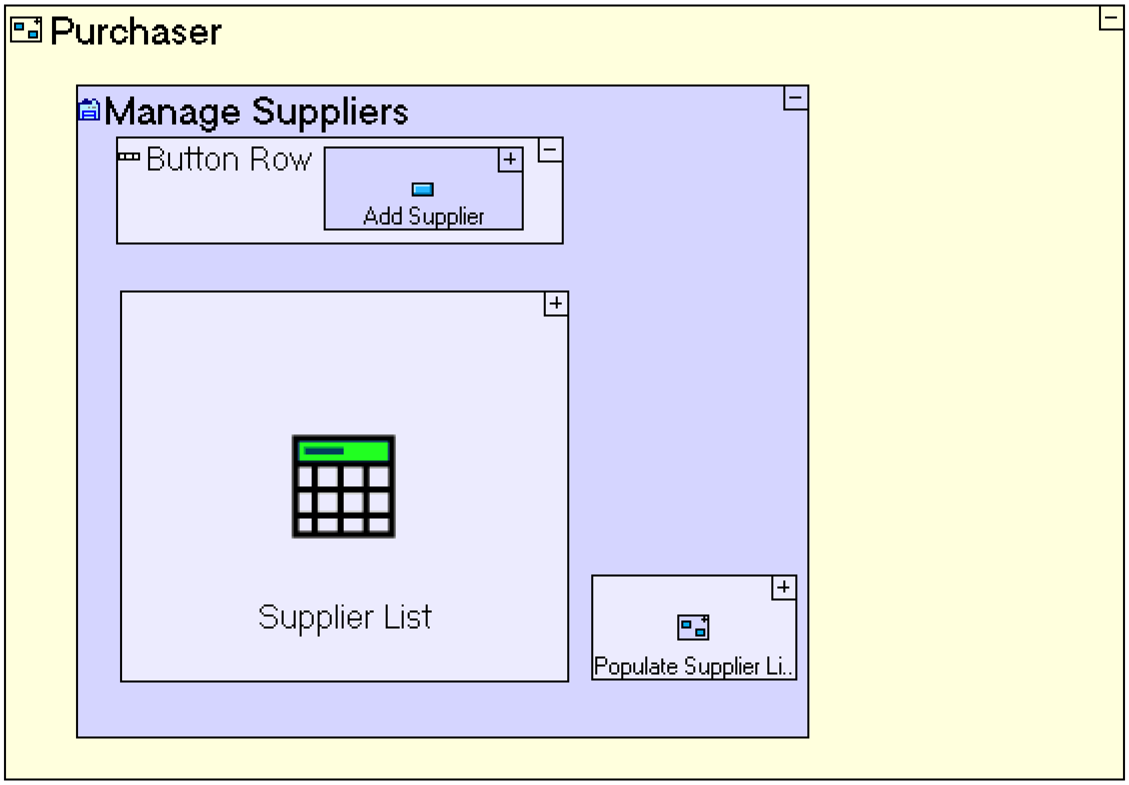
The purchaser manages a list of suppliers to whom the relevant POs are issued.

To support this, we shall model the following:

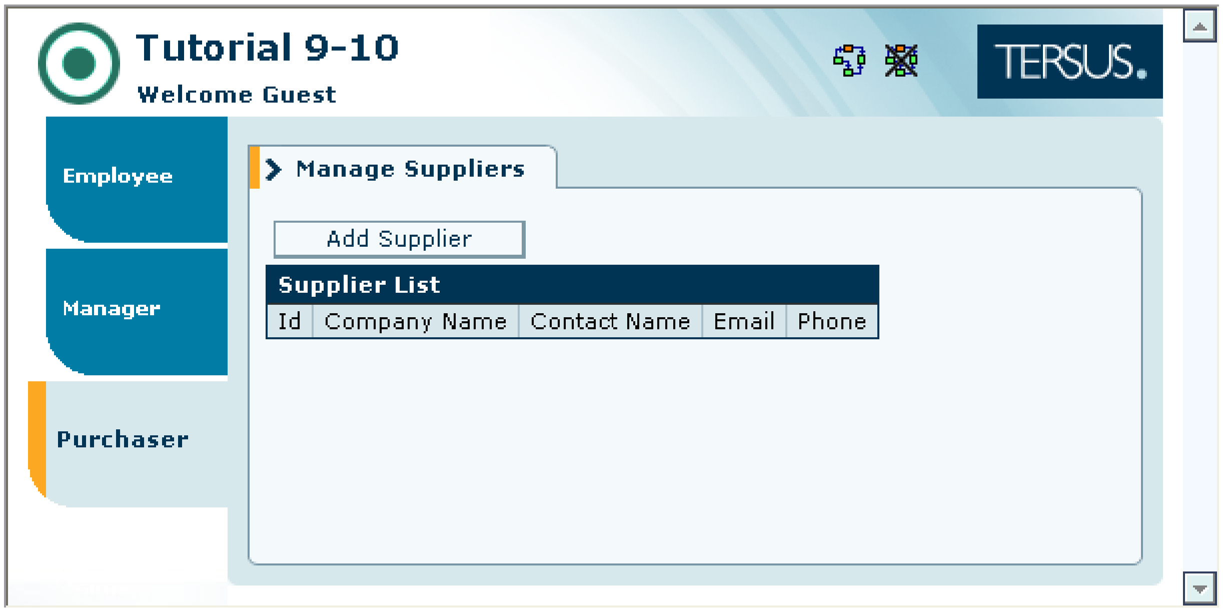
- Managing the list of approved suppliers – Including the import of a list of suppliers from an Excel spreadsheet.
- Managing the list of POs. Each requisition may result in the purchaser issuing multiple POs to one or more suppliers.

On completing the previous stage, we included in the sample project a new **Purchaser** perspective with a basic **Manage Suppliers** view, which manages a **Suppliers** database table and already implements the display in a table and the addition of new suppliers.

The model provided looks as follows:



When viewed in the browser, it looks as follows:



This stage's modeling should be performed in the **Tutorial 9-10** project, you imported at the end of the previous stage.

User Modeling

In most organizations, some data is maintained by certain officers on their PCs, using a spreadsheet program such as Excel (or in an application that can output/export Excel files).

For the sake of this tutorial, we shall assume that the purchaser in our organization has been maintaining the list of approved suppliers in a spreadsheet. A sample spreadsheet, **Suppliers.xls** is provided in **[tersus root]/workspace/Tutorial 9-10**.

The data in the spreadsheet should be imported into the **Suppliers** database table which is already modeled.

Use a File Input Field (to select the spreadsheet file)

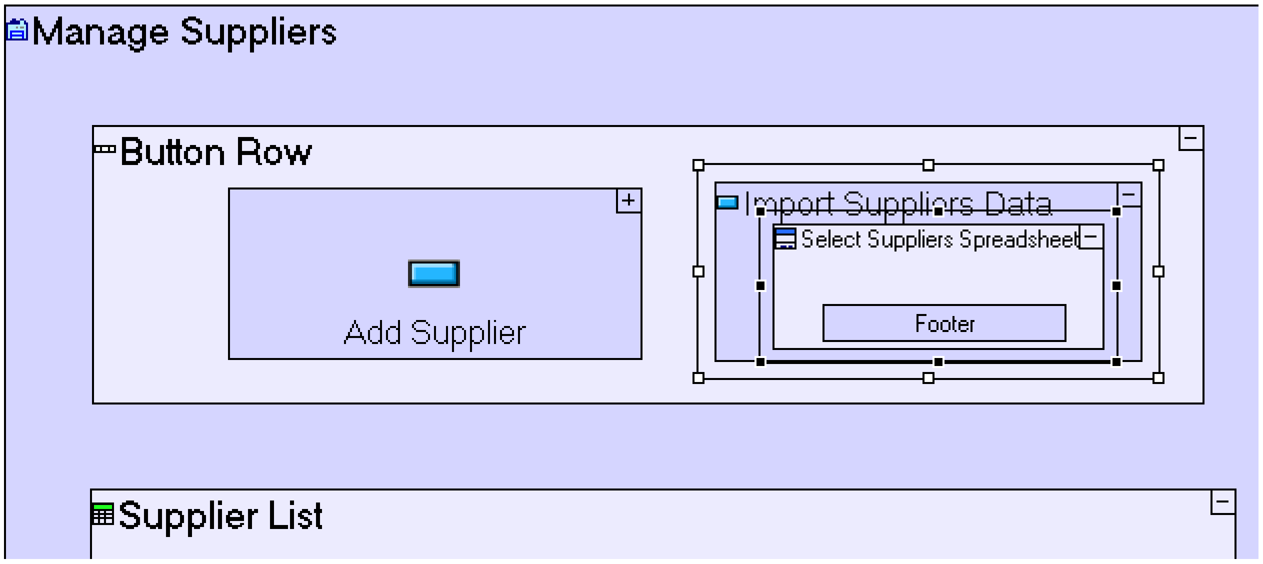
We shall add a button and popup which allow the user to select, and ultimately import, the data:

Zoom to the **Manage Suppliers** view.

Add **Display/Button** next to **Add Supplier**. Name it ***Import Suppliers Data***. Zoom into it.

Add **Display/Popup**. Name it ***Select Suppliers Spreadsheet***.

The model should look as follows:



Zoom into ***Select Suppliers Spreadsheet***.

Add **Display/Row**. Name it **File Row**. Zoom into it.

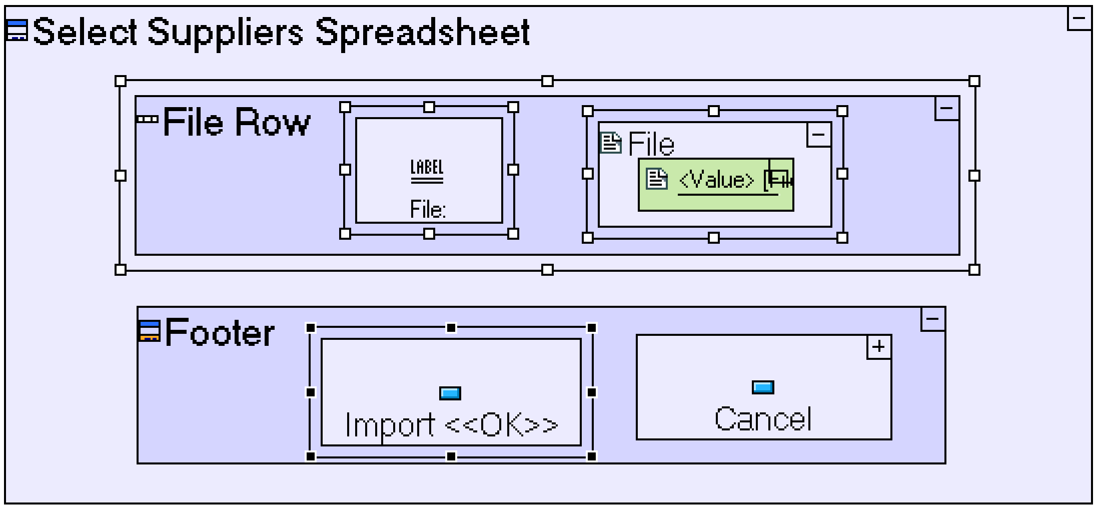
Add **Display/Label**. Name it ***File:***.

Add **Display/File Input Field**. Name it ***File***.

Zoom to **Select Suppliers Spreadsheet/Footer**.

Rename the **OK** button to **Import**.

The **Select Suppliers Spreadsheet** popup model should look as follows:



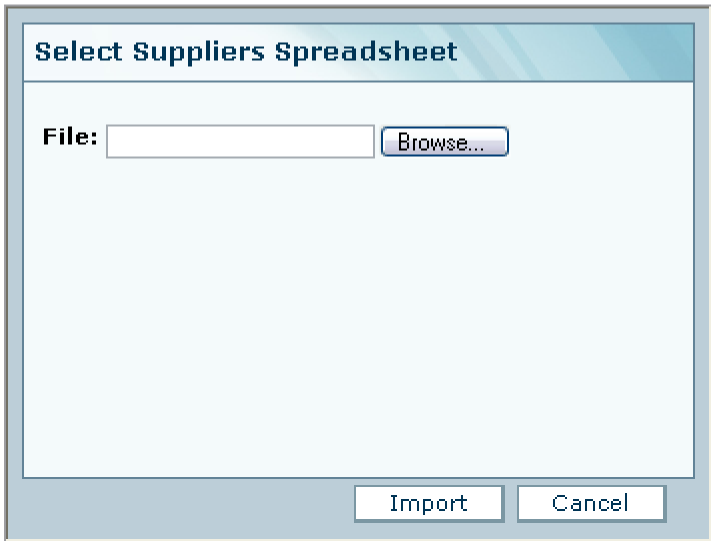
The **File Input Field** (**File**) contains a default **<Value>** data element (of type **File**) which is needed in order to manipulate and extract the contents of the file.

Save your work, and view the application in the browser.

Click on the **Purchaser** perspective tab.

Press the **Import Suppliers Data** button.

The popup we have created should look as follows:



Notice that a **File Input Field** is actually implemented in the browser in two parts: a **text edit** which contains the file name which may be manually entered, and a **Browse…** button, which allows the user to navigate his local file system, and select the file to import.

Use a Load Excel Table Template (to extract data rows)

Now that we can select a spreadsheet file to import, let’s model the actual import process.

Start by retrieving the file selected by the user in the **Select Suppliers Spreadsheet** popup:

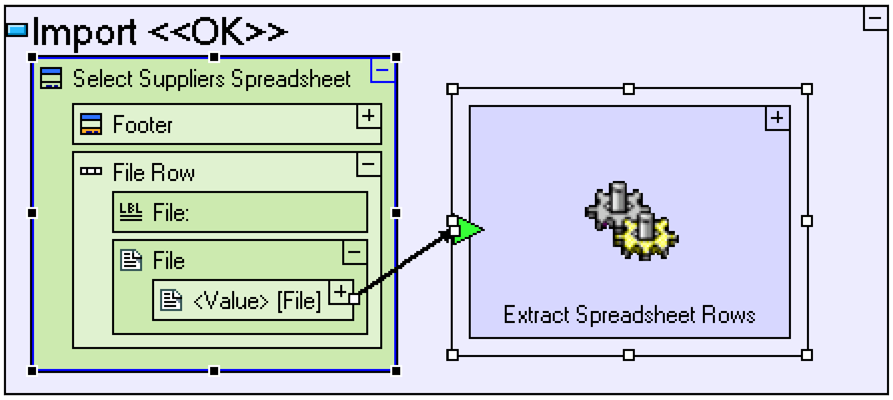
Zoom to the **Import** button.

Add an ancestor reference of the **Select Suppliers Spreadsheet** popup.

Add a **Basic/Service** (). Name it ***Extract Spreadsheet Rows***. Add a Trigger to it.

Add flow linking **Select Suppliers Spreadsheet/Row/File/<Value>** to the **Extract Spreadsheet Rows** trigger.

The **Import** button model should look as follows:



Note that we are using a new type of process template, **Basic/Service** (), for **Extract Spreadsheet Rows**, instead of the usual **Basic/Action**.
The fundamental difference between a **Service** and an **Action** is that the modeling inside a **Service**, will always execute on the server-side, whereas **Action** models will switch dynamically (and transparently) between **Client** and **Server** as required by the different templates used in the model.
There are however certain modeling scenarios, such as the modeling we are about to perform, which must explicitly be defined as executing on the server-side. For more information, see **Stage 13**.

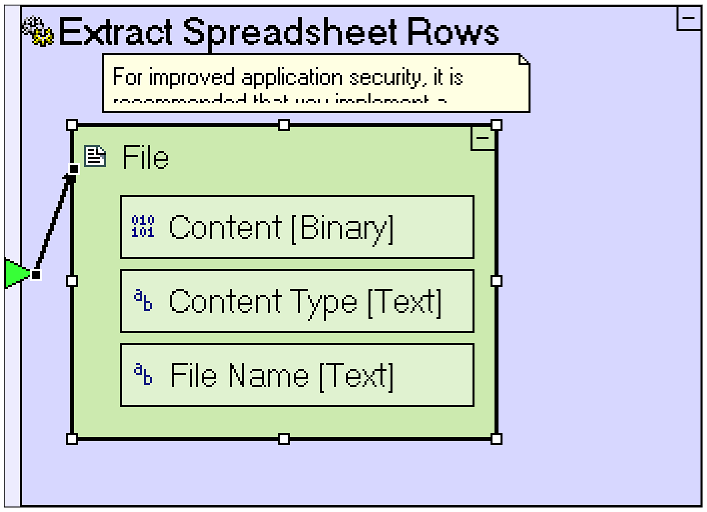
The **<Value>** data element of the **File** input field is passed to the **Extract Spreadsheet Rows** process, and into another **File** data element:

Zoom into **Extract Spreadsheet Rows**.

Add a **Data Types/File** data structure ().

Add flow linking the **Extract Spreadsheet Rows** trigger to the **File** data structure.

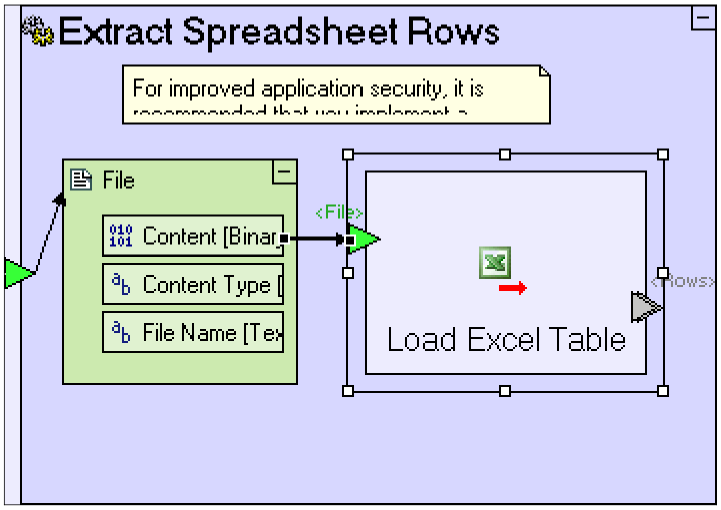
The **Extract Spreadsheet Rows** service model should now look as follows:



The **Content** element of the **File** data structure contains the actual Excel data:

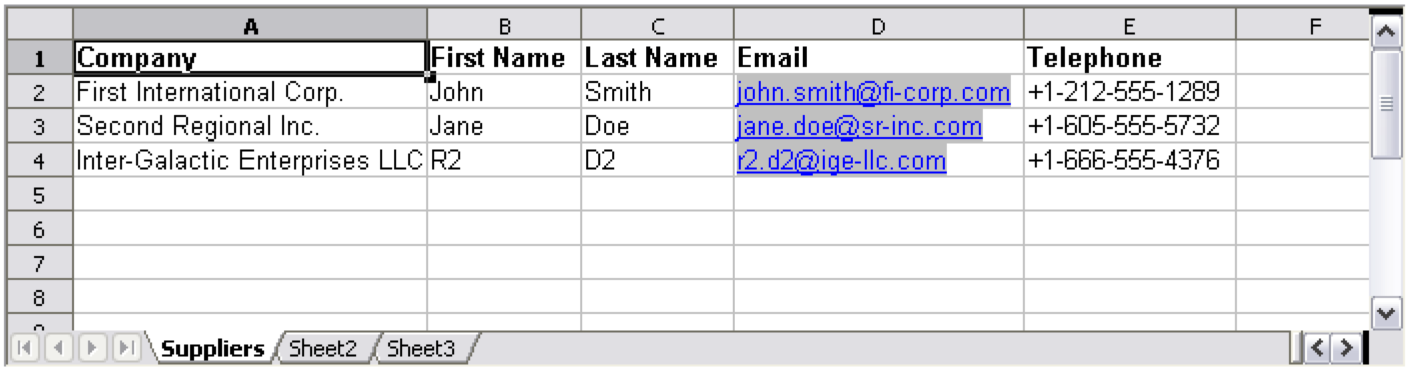
> Add a **Miscellaneous /Load Excel Table** template ().
>
> Add flow linking **File/Content** to **Load Excel Table/<File>**.

The **Extract Spreadsheet Rows** service model should now look as follows:



You may be thinking that the modeling we have so far performed in the **Import** button is unnecessarily complicated, and can be simplified by directly passing **Select Suppliers Spreadsheet/Row/File/<Value>/Content** to **Load Excel Table/<File>**.
We have modeled as we did, due to limitations imposed by the browser, which does not allow access to **Content** directly. The **File** data structure must be passed to the server as-is, in order for Tersus to be able to extract the binary content from it.

Let's pause for a minute and take a look at the sample spreadsheet file, **Suppliers.xls**, from which we plan to import the data:



	A	B	C	D	E	F
1	Company	First Name	Last Name	Email	Telephone	
2	First International Corp.	John	Smith	john.smith@fi-corp.com	+1-212-555-1289	
3	Second Regional Inc.	Jane	Doe	jane.doe@sr-inc.com	+1-605-555-5732	
4	Inter-Galactic Enterprises LLC	R2	D2	r2.d2@ige-llc.com	+1-666-555-4376	
5						
6						
7						
8						

The data appears in the **Suppliers** sheet, and is formatted in a tabular format, where the first row defines column names (**Company, First Name**, **Last Name**, **Email**, and **Telephone**) and subsequent rows contain supplier information (one per row).

> **Load Excel Table** will search for data in the first sheet by default. If the required sheet is not the first one in the file, you may specify the sheet using the optional **<Sheet Name>** trigger (available through right-click -> **Add Element**).

Define the Data Structure of Rows extracted from the Spreadsheet

The **Load Excel Table** process still needs a definition of the way the relevant data in the sheet is structured. This definition is provided in a similar fashion to the **Find** template (discussed in a previous stage) – by deducing the data structure from the target of its exit, **<Rows>**, as we will see in a few minutes.

Once rows are extracted, the data should be copied into the **Supplier** database record which is used to store suppliers in the database:

Zoom to **Extract Spreadsheet Rows**.

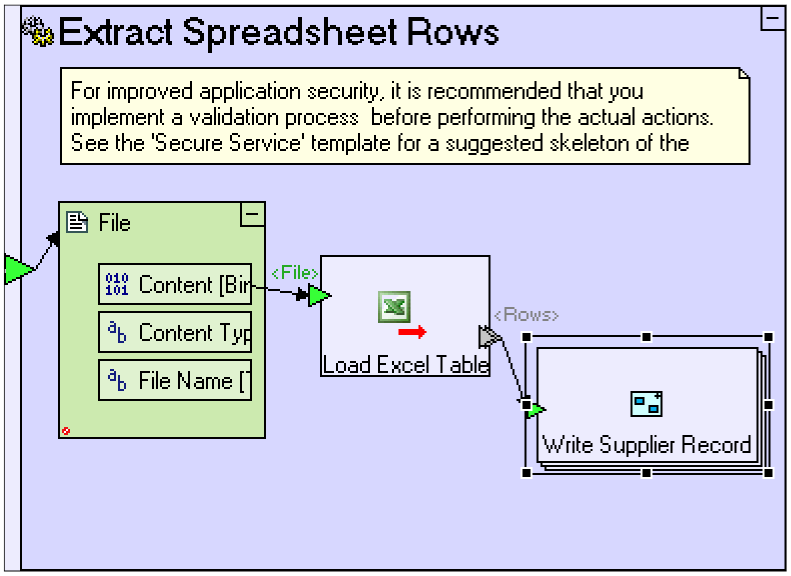
Add a **Basic/Action**. Name it ***Write Supplier Record***. Add a Trigger to it.

Add a flow linking **Load Excel Table/<Rows>** to the **Write Supplier Record** trigger.

As there are usually multiple rows extracted from the spreadsheet (which is the reason the **<Rows>** exit is **repetitive**), the **Write Supplier Record** process should be marked as repetitive – meaning that it is executed once for each row which is extracted by **Load Excel Table**:

Right-Click on the **Write Supplier Record** process, and check the repetitive option.

The **Extract Spreadsheet Rows** process should look similar to the following:


Extract Spreadsheet Rows
For improved application security, it is recommended that you implement a validation process before performing the actual actions. See the 'Secure Service' template for a suggested skeleton of the
File
Content [Bir
Content Typ
File Name [
<File>
Load Excel Table
<Rows>
Write Supplier Record

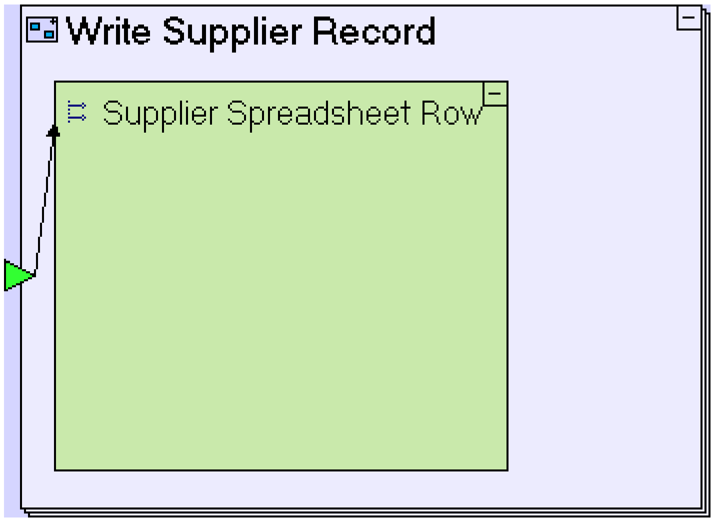
The input to the **Write Supplier Record** trigger is a single row from the spreadsheet, and so it is time to define its type:

Zoom into **Write Supplier Record**.

Add a **Data Types/Data Structure** (). Name it ***Supplier Spreadsheet Row***.

Add flow linking the **Write Supplier Record** trigger to **Supplier Spreadsheet Row**.

The **Write Supplier Record** action process should look similar to the following:



Recall that in a previous stage we mentioned that a **Data Structure** and a **Database Record** are practically identical, apart from the fact that the latter is automatically mapped to a table in the database.

The fields in this data structure should exactly match the columns names (the first row) of the spreadsheet (see the screenshot above):

Zoom into **Supplier Spreadsheet Row**.

Add a **Data Types/Text**. Name it ***Company***.

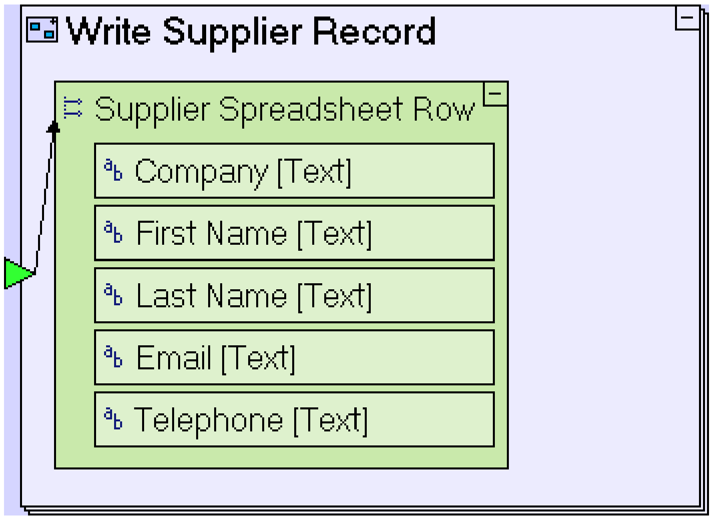
Add a **Data Types/Text**. Name it ***First Name***.

Add a **Data Types/Text**. Name it ***Last Name***.

Add a **Data Types/Text**. Name it ***Email***.

Add a **Data Types/Text**. Name it ***Telephone***.

The **Write Supplier Record** process should now look as follows:



The data in each **Supplier Spreadsheet Row** should be copied to a corresponding **Supplier** record:

Zoom to **Write Supplier Record**.

Reuse the **Supplier** data structure from the repository/outline (find it in **Manage Suppliers** view/**Supplier List** table).

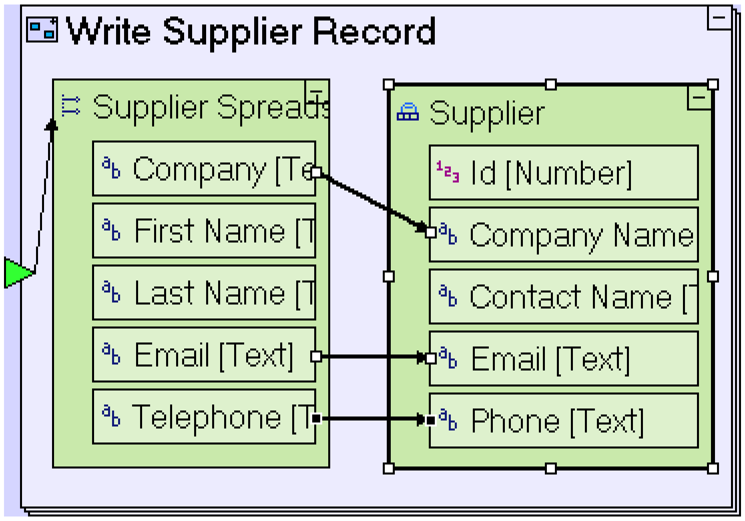
Create flow linking the following:

Supplier Spreadsheet Row/Company to **Supplier/Company Name**

Supplier Spreadsheet Row/Email to **Supplier/Email**

Supplier Spreadsheet Row/Telephone to **Supplier/Phone**

The **Write Supplier Record** process should now look as follows:



Two additional fields should be populated in **Supplier**:

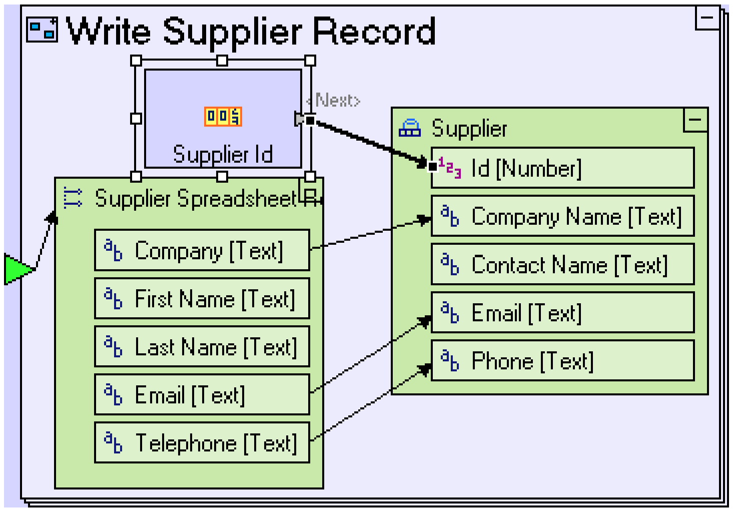
4. **Id –** A unique identifier for each supplier.
5. **Contact Name –** The **Supplier** record has a single field for storing the name of the contact person as opposed to the spreadsheet where there are two fields, **First Name** and **Last Name**. Therefore **Contact Name** should store the two fields joined together.

Id will be populated using the **Sequence Number** action. The same one used in **Add Supplier** button/**Enter New Supplier** popup/**Submit** button:

Reuse **Supplier Id** by dragging it from the repository/outline.

Add flow linking **Supplier Id/<Next>** to **Supplier/Id**.

The **Write Supplier Record** process should now look as follows:



Using a Text Manipulation Template (to concatenate text values)

To join **First Name** and **Last Name** into a single text value (remember that there should also be a space separating them), do the following:

Add a **Text/Concatenate** template (ab+cd).

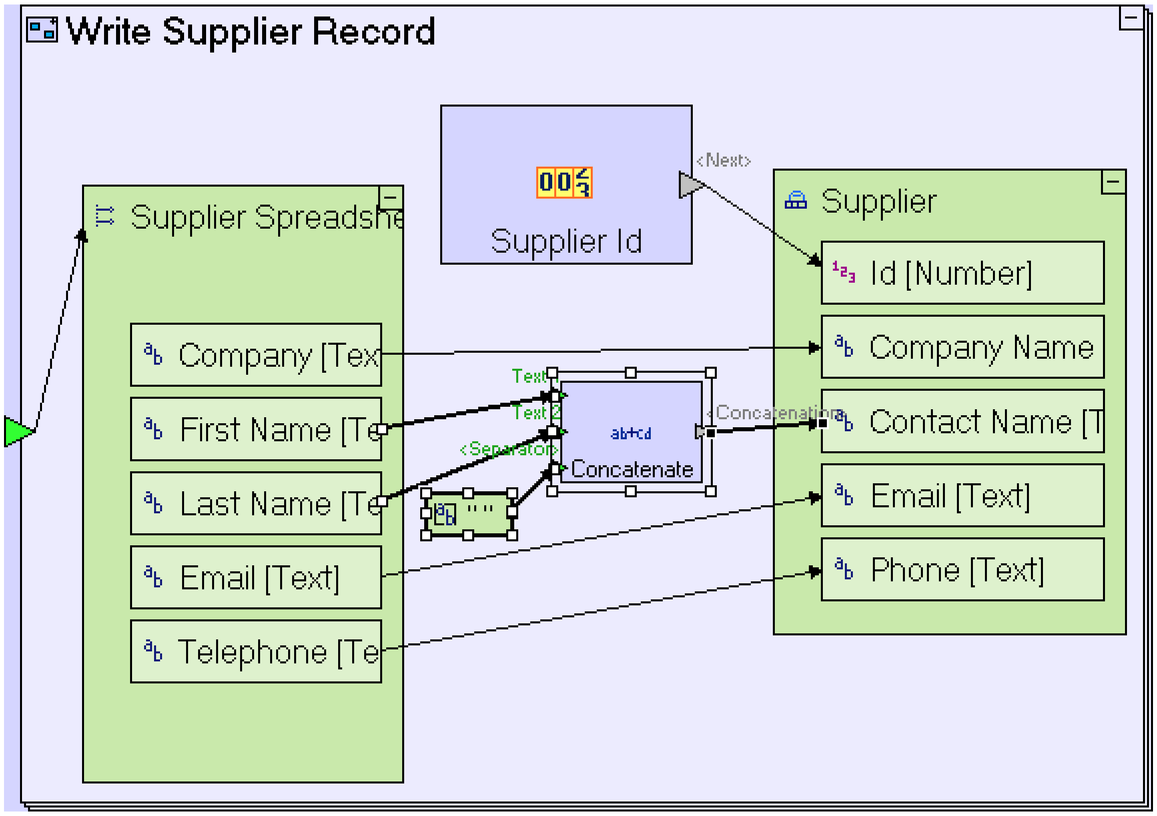
Create a flow linking **Supplier Spreadsheet Row/First Name** to **Concatenate/Text 1**.

Create a flow linking **Supplier Spreadsheet Row/Last Name** to **Concatenate/Text 2**.

Add a **Constants/Text**. Press **[Space]** once to create the " " constant, and create a flow linking " " to **Concatenate/<Separator>**.

Create a flow linking **Concatenate/<Concatenation>** to **Supplier/Contact Name**.

The **Write Supplier Record** process should now look as follows:



Completing the Import process

To complete the import process, we need to take care of the following:

1. Insert **Supplier** records to the database.
2. Close the **Select Suppliers Spreadsheet** popup
3. Refresh the **Supplier List** table in the **Manage Suppliers** view.

To add **Supplier** records to the database:

Zoom to **Write Supplier Record**.

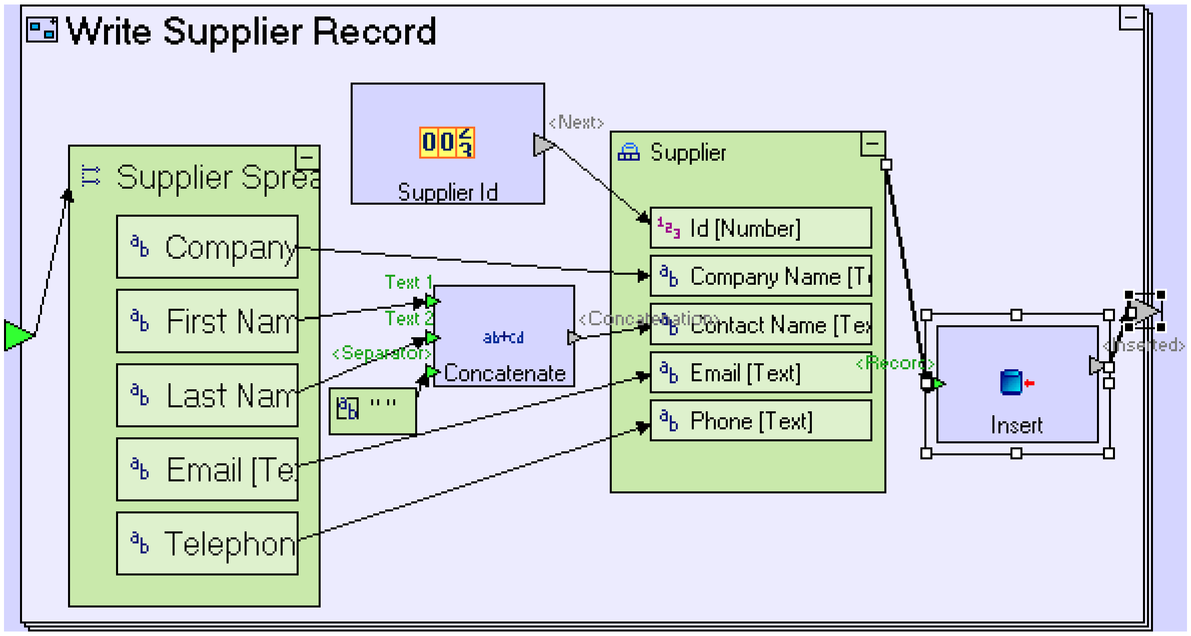
Add **Database/Insert**.

Create a flow linking **Supplier** to **Insert/<Record>**.

Add an **Exit** to **Write Supplier Record**.

Create flow linking the **Insert/<Inserted>** exit to the **Write Supplier Record** exit.

The **Write Supplier Record** process should now look as follows:



To wrap up the import process, closing the popup and refreshing the **Supplier List**, additional flow must be defined:

Zoom out to **Extract Spreadsheet Rows** service. Add an exit to the process.

Create a flow linking the **Write Supplier Record** exit to the **Extract Spreadsheet Rows** exit.

Zoom out to **Import** button.

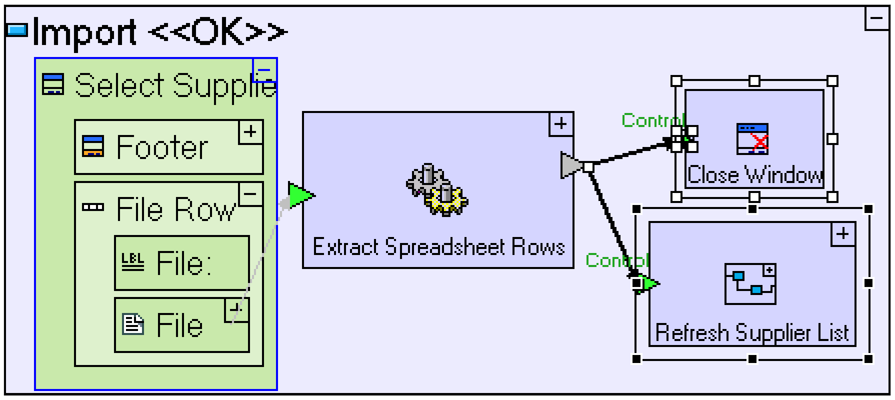
Add a **Display Actions/Close Window** template. Add a **Control** trigger (through right-click->**Add Element** or by simply adding a trigger).

Link the **Extract Spreadsheet Rows** exit to the trigger of **Close Window**.

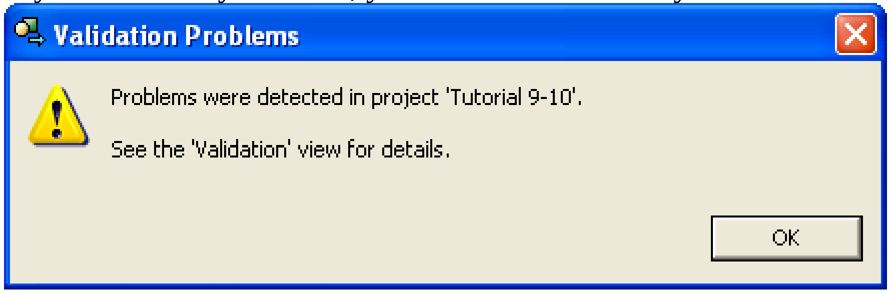
Reuse the **Refresh Supplier List** process (used in **Add Supplier** button/**Enter New Supplier** popup/**Submit** button) by dragging it from the repository/outline.

Link the **Extract Spreadsheet Rows** exit to the trigger of **Refresh Supplier List**.

The **Import** button model should now look as follows:



If you now save your work, you should automatically receive the following warning:

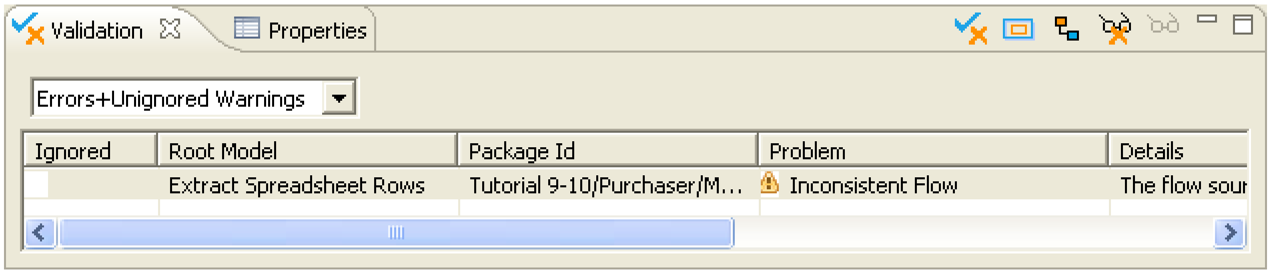


Validation Problems

Problems were detected in project 'Tutorial 9-10'.

See the 'Validation' view for details.

Clicking **OK**, the focus will switch to the **Validation** view:

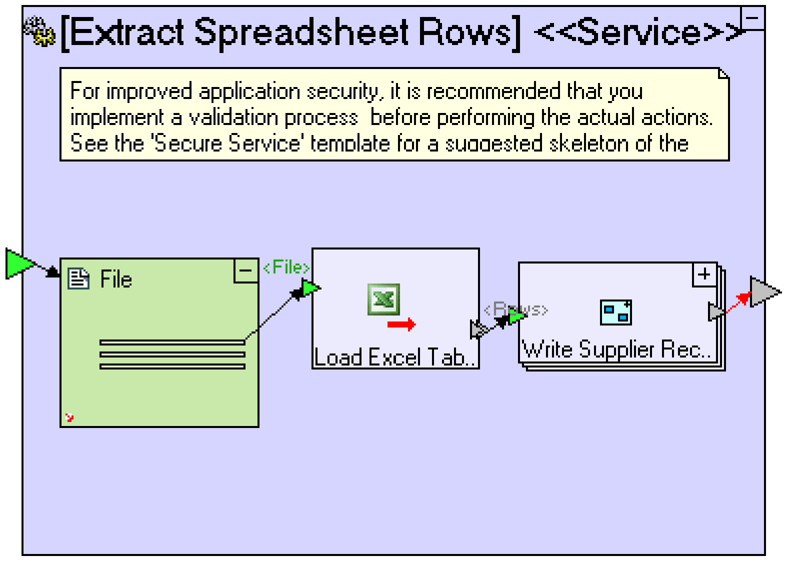


Ignored	Root Model	Package Id	Problem	Details
	Extract Spreadsheet Rows	Tutorial 9-10/Purchaser/M...	Inconsistent Flow	The flow sour

The **Validation** view is the interface to a built-in utility, which checks your models and notifies you of potential problems).
By default validation is run on every save, but you can run it independently, without saving, by switching to the **Validation** view and clicking the **Validate** (✓✗) toolbar button.

Click on the cell containing **Extract Spreadsheet Rows** (in fact you can click on any of the cells except the one in the Ignored column).

The model editor will zoom to the **Extract Spreadsheet Rows** with the last flow arrow highlighted in red to pinpoint the validation warning, as in the following screenshot:


[Extract Spreadsheet Rows] <<Service>>
For improved application security, it is recommended that you implement a validation process before performing the actual actions. See the 'Secure Service' template for a suggested skeleton of the
File
<File>
Load Excel Tab..
<Rows>
Write Supplier Rec..

The validation warning is caused by the fact that the flow arrow starts at a repetitive element (the exit slot of the repetitive **Write Supplier Record** process) but terminates at a non-repetitive element (the exit slot of the **Extract Spreadsheet Rows** process), as descibed in the **Details** column of the **Validation** view.

This issue is designated a warning rather than an error, because in some modelling scenarios this is actually the intended modelling. Indeed **Extract Spreadsheet Rows** is one such case, therefore it should simply be ignored.

In order to avoid being reminded of this non-issue in future validations, you can request to hide it in the future, as follows:

Click (again) on the row in the **Validation** view, to make sure it is selected.

Click the **Ignore selected warnings** ( ) toolbar button.

To wrap up this stage, switch to the browser and review the results:

Click on the **Purchaser** perspective tab.

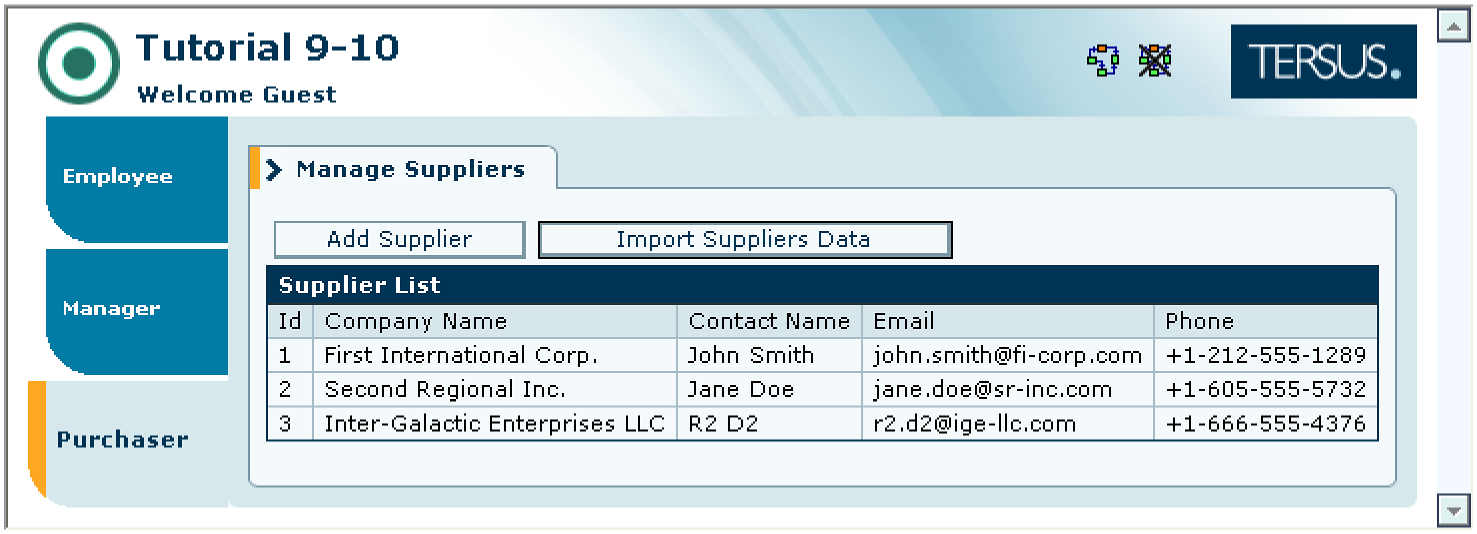
Press the **Import Suppliers Data** button.

In the **Select Suppliers Spreadsheet** popup press the **Browse…** button.

Navigate to **[tersus root]/workspace/Tutorial 9-10** and select **Suppliers.xls**.

Press the Import button.

The resulting **Supplier List** should look similar to the following:



Supplier List

Id	Company Name	Contact Name	Email	Phone
1	First International Corp.	John Smith	john.smith@fi-corp.com	+1-212-555-1289
2	Second Regional Inc.	Jane Doe	jane.doe@sr-inc.com	+1-605-555-5732
3	Inter-Galactic Enterprises LLC	R2 D2	r2.d2@ige-llc.com	+1-666-555-4376

Completing Stage 10

Import the sample project **Tutorial 10-11** and use it as the basis for the next stage of the tutorial.

For a reminder on how to import a sample project, see the **Importing a Sample Project** section at the end of **Stage 2**.

This sample project contains all the functionality modeled thus far.

You may now proceed to **Stage 11**, in which we shall model a table display using a different technique which provides better control of its content.

See It Live

Object 9

Click here to open the live project in a separate window.

Stage 11 – Controlling Table Display

Stage Goals

Tersus Concepts Covered

On completion of this stage you should be familiar with the following concepts:

Modeling techniques: Controlling the columns appearing in a display

Useful display templates: Table, Number Display, Text Display

Application Functionality Modeled

We shall now begin modeling the issuing of **Purchase Orders (POs)** by purchasers. We shall start with the display of a tabular list of requisitions approved by managers. This table will differ from previously modeled tables – we will specify which columns to display and in which order, instead of relying on defaults.

In the subsequent stages we shall add:

- An alert, displayed if any of the requisitions are marked as urgent, to prompt the purchaser to process them.
- A list of issued POs for the currently selected requisition.
- An **Order** button to issue a purchase order to a supplier for the currently selected requisition.
- After issuing a purchase order, the requisition is marked as (partially or fully) ordered.

This stage's modeling should be performed in the **Tutorial 10-11** project, you imported at the end of the previous stage.

User Modeling

We start by adding a new view:

Zoom to **Purchaser**

Add **Display/View**. Name it ***Issue Purchase Orders***.

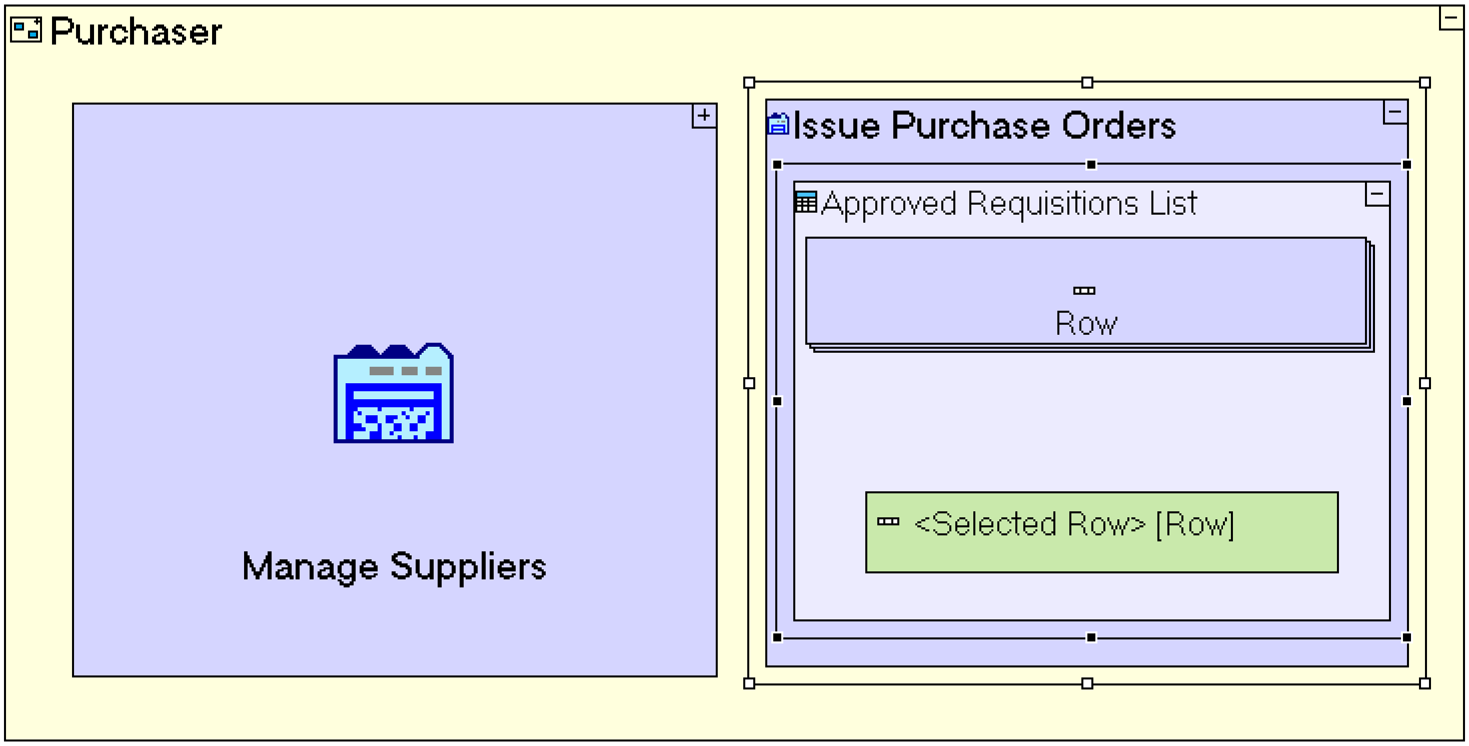
Use a Table Template

Up to this point in the tutorial, whenever we wanted to display a tabular list of data, we used the **Simple Table** template. The advantage of using this template is that it is very quick to model. The disadvantage is that you have no control over which columns are displayed, and at what order. There is also additional functionality we will require in upcoming stages which cannot be implemented using a **Simple Table**, but rather using the **Table** template:

Zoom to **Issue Purchase Orders**.

Add a **Display/Table** template (). Name it ***Approved Requisitions List***. Zoom into it.

The **Purchaser** model should look similar to the following:



The **Table** template comes pre-configured with a **Row** display element and a **<Selected Row>** display data element (of the **Row**) display, which will be used in subsequent steps to design and control the table display.

Use Number/Text/Date Display Templates

Define the columns, by inserting **Text**/**Number**/**Date Display** templates into the row, each defining a cell in the row.

The name given to each element in the row will be used as the column heading, when the table is displayed:

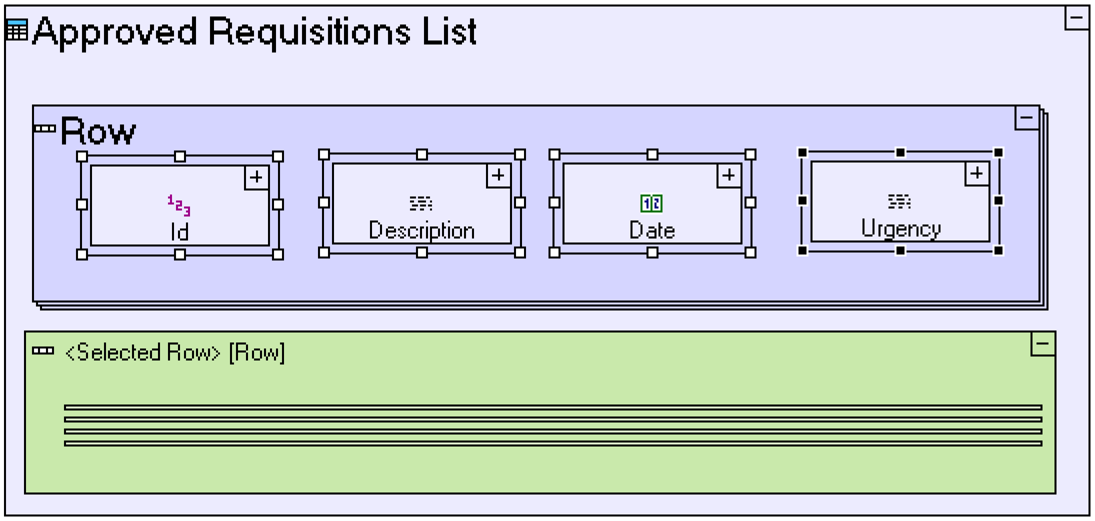
Add a **Display/Number Display** (). Name it ***Id.***

Add a **Display/Text Display** (). Name it ***Description***.

Add a **Display/Date Display** (). Name it ***Date***.

Add a **Display/Text Display** (). Name it ***Urgency***.

The **Approved Requisitions List** table model should look similar to the following:



The **Number/Text/Date Display** templates are similar to the **Text Area** input field we've previously used: each contains a **<Value>** element, which holds its value and serves as the source/target of flows which involve this value.
The difference is in the user interface aspects of the two: An input field provides the user with the ability to change the value, whereas a display does not.

The flexibility of using the **Table** template now comes into play. We control which fields from the database are displayed (**Status** in this case is not displayed), Also, we can explicitly define the order by which columns are displayed.

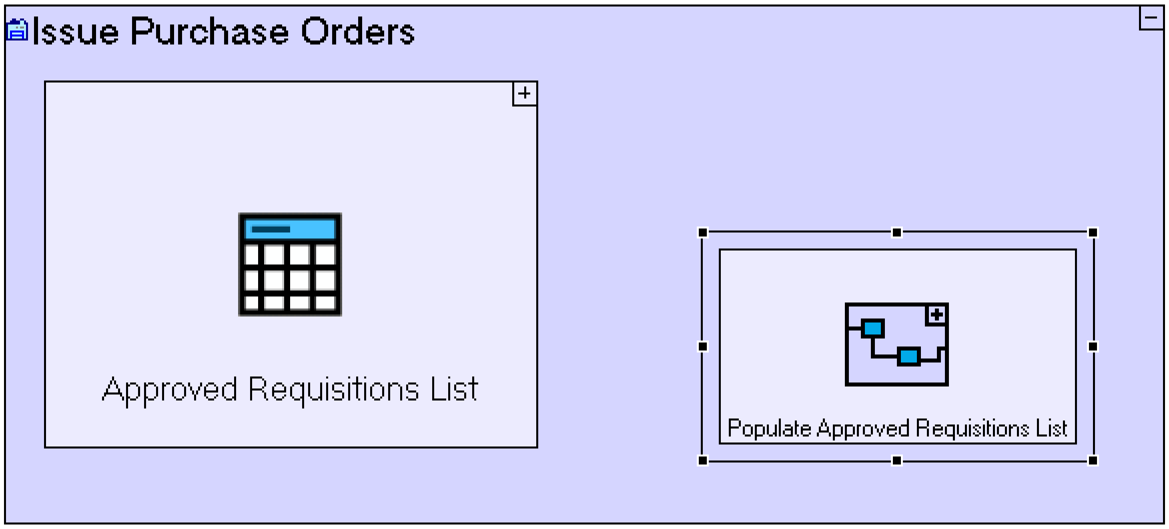
Populate the Table (with requisitions)

As with the **Simple Table**, we should create a process that populates the table with values from the database:

Zoom out to **Issue Purchase Orders** view.

Add a **Basic/Action**. Name it ***Populate Approved Requisitions List***.

The **Issue Purchase Orders** view model should look similar to the following:


Issue Purchase Orders
Approved Requisitions List
Populate Approved Requisitions List

As we have done in previous cases, the **Populate…** process will have a **Generate…** process which creates a display data element representing the table and outputs it to an ancestor reference of the display:

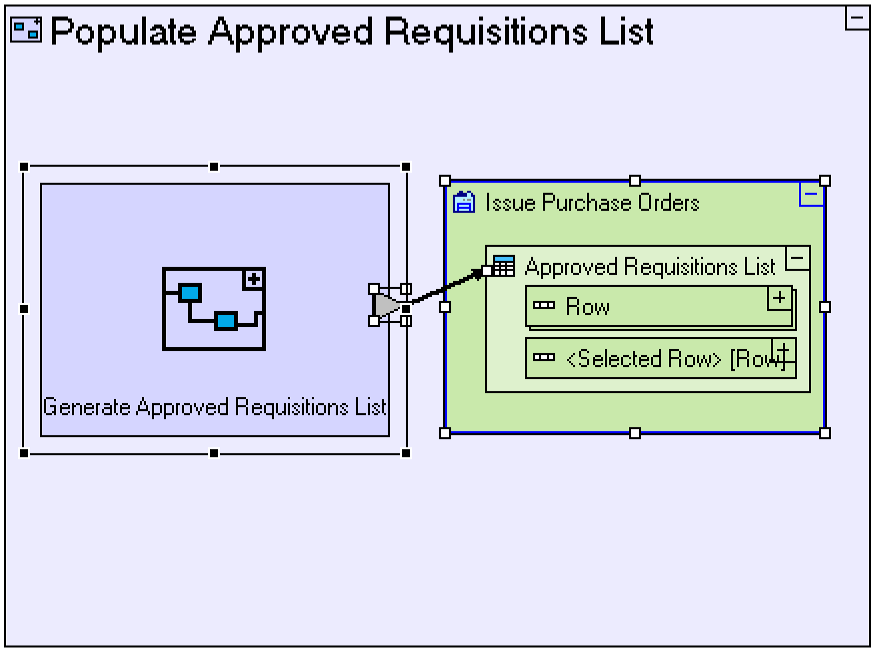
Zoom into ***Populate Approved Requisitions List***.

Add **Basic/Action**. Name it ***Generate Approved Requisitions List***. Add an exit to it.

Add an ancestor reference of the **Issue Purchase Orders** view.

Create a flow linking the exit of **Generate Approved Requisitions List** to the **Issue Purchase Orders/Approved Requisitions List** table.

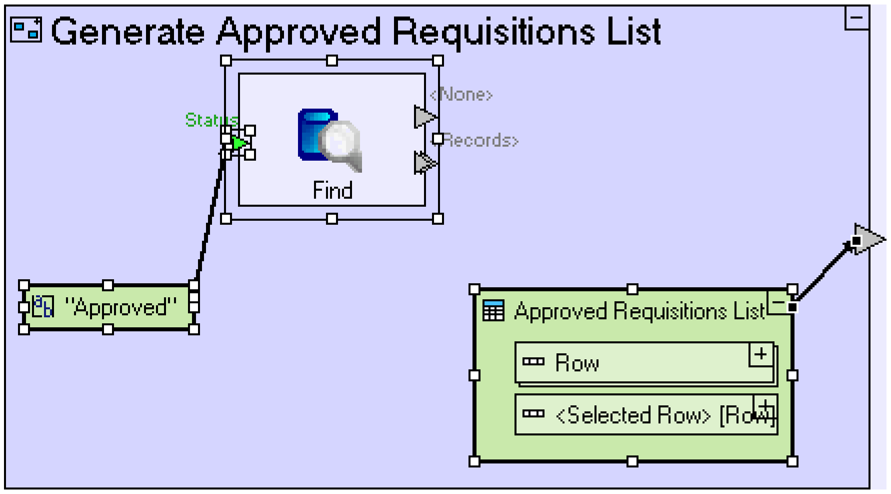
The **Populate Approved Requisitions List** view model should look similar to the following:


Populate Approved Requisitions List
Generate Approved Requisitions List
Issue Purchase Orders
Approved Requisitions List
Row
<Selected Row> [Row]

The **Generate Approved Requisitions List** process retrieves all requisitions which have been approved:

> Zoom into **Generate Approved Requisitions List**.
>
> Add a **Database/Find**. Add a trigger to it. Name the trigger ***Status***.
>
> Reuse the **"Approved"** constant (used in **Manager** system/**Requisition Approval** view/**Approve Requisition** button). Create a flow linking it to **Find/Status**.
>
> Drag **Approved Requisitions List** table from the repository/outline. Create flow from it to the **Generate Approved Requisitions List** exit.

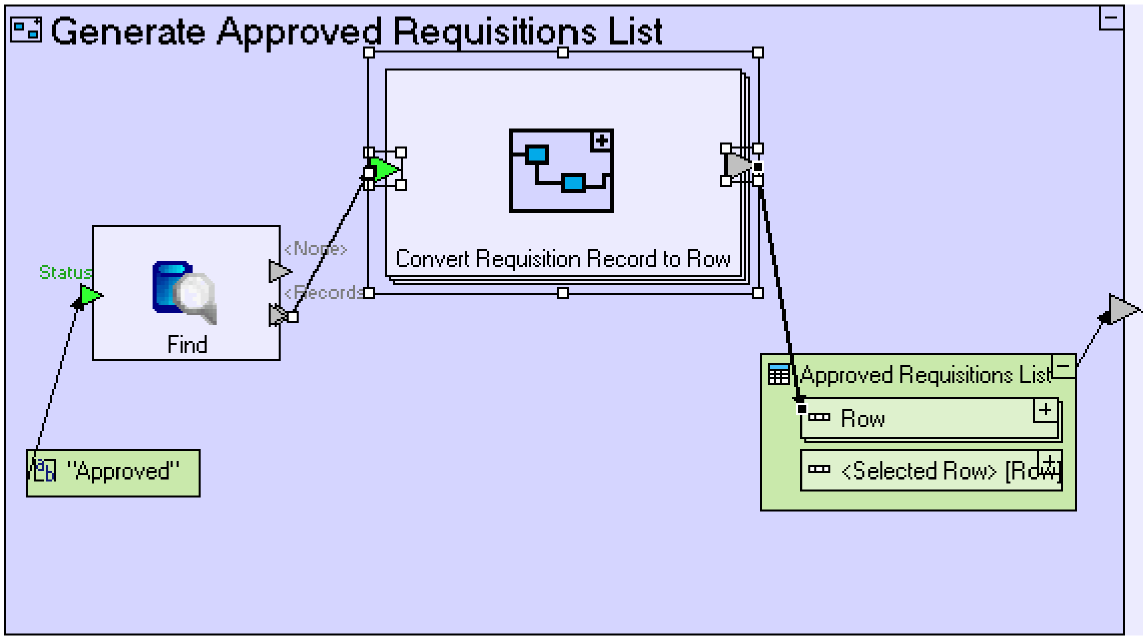
The **Generate Approved Requisitions List** view model should look similar to the following:


Generate Approved Requisitions List
Status
<None>
<Records>
Find
"Approved"
Approved Requisitions List
Row
<Selected Row> [Row]

Up to this point, the **Generate Approved Requisitions List** process, is similar to other **Generate...** processes we modeled. From this point on, the modeling changes. As the table will not accept the records found directly, we must create a process that converts (or maps) the fields of the **Requisition** database record to a **Row** display data element in the table:

> Add a **Basic/Action**. Name it ***Convert Requisition Record to Row***. Set it repetitive. Add a trigger and exit.
>
> Create a flow linking **Find/<Records>** to **Convert Requisition Record to Row/Record**.
>
> Create a flow linking the **Convert Requisition Record to Row/Row** exit to **Approved Requisitions List/Row**.

The **Generate Approved Requisitions List** view model should look similar to the following:


Generate Approved Requisitions List
Convert Requisition Record to Row
Status
<None>
<Records>
Find
"Approved"
Approved Requisitions List
Row
<Selected Row> [Row]

And model the mapping from **Requisition** to **Row**:

Zoom into **Convert Requisition Record to Row**.

Drag a **Requisition** databased record from the repository/outline (it's reused in various locations such as **Open Requisitions** view/**Requisition List** table). Create a flow linking the process trigger to it.

Drag a **Row** from the repository/outline (this is the row in **Approved Requisitions List** table). Create a flow linking it to the process exit.

Create the following flows:

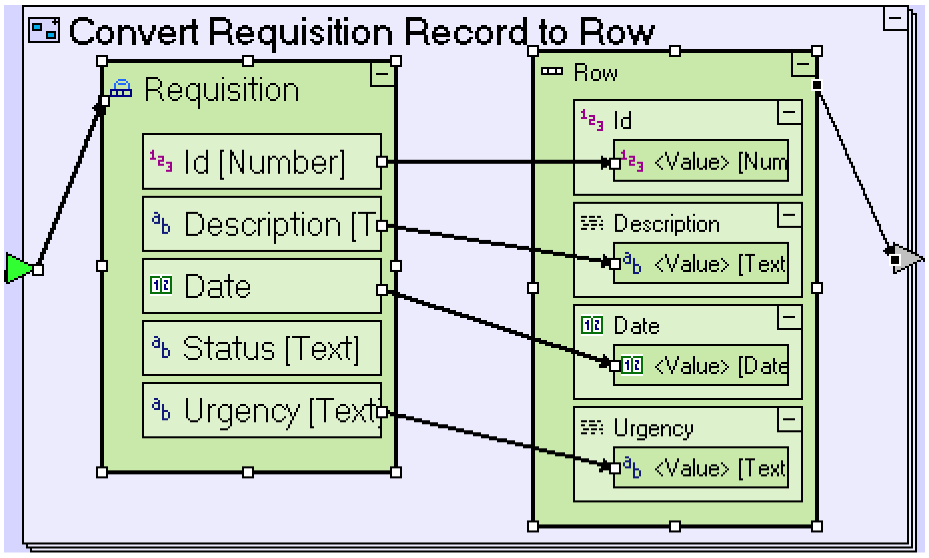
Requisition/Id to **Requisition Row/Id/<Value>**

Requisition/Description to **Requisition Row/Description/<Value>**

Requisition/Date to **Requisition Row/Date/<Value>**

Requisition/Urgency to **Requisition Row/Urgency/<Value>**

The **Convert Requisition Record to Row** model should look similar to the following:

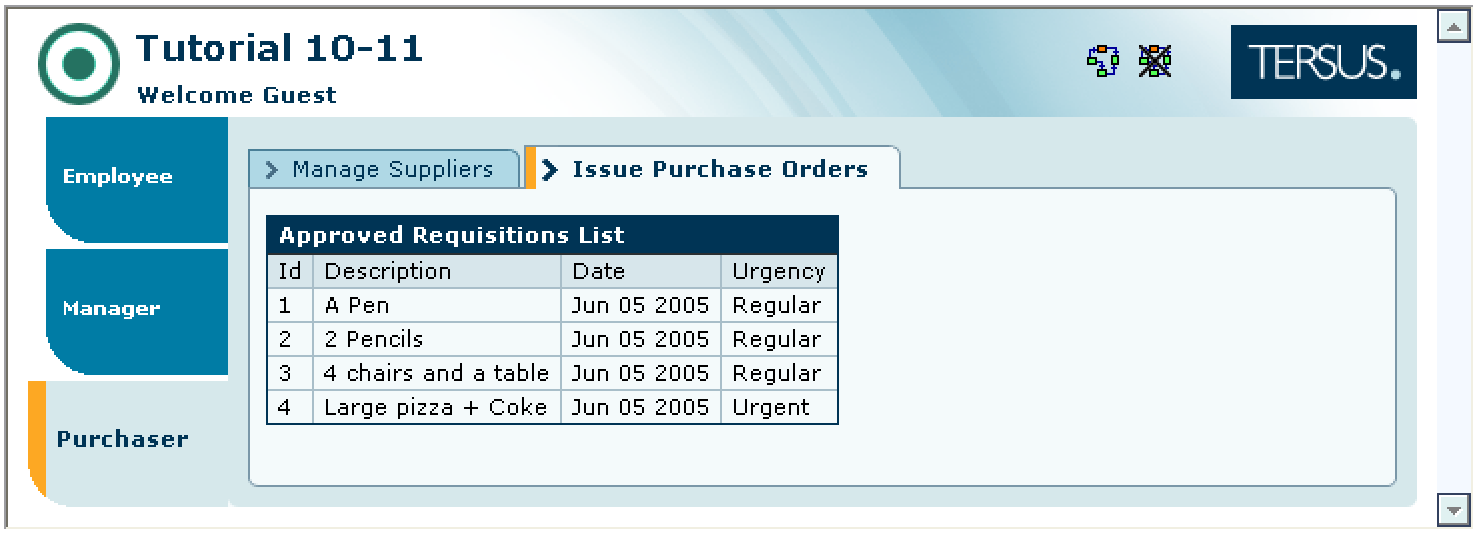


Save your work, and view the application in the browser.

Click on the **Purchaser** perspective tab.

Switch to the **Issue Purchase Orders** view.

You should see a list of (approved) requisitions, similar to the following:



Approved Requisitions List

Id	Description	Date	Urgency
1	A Pen	Jun 05 2005	Regular
2	2 Pencils	Jun 05 2005	Regular
3	4 chairs and a table	Jun 05 2005	Regular
4	Large pizza + Coke	Jun 05 2005	Urgent

Note that the requisition list displayed is different from previously modeled requisition lists, specifically the absence of the **Status** column. In the subsequent stages we will utilize additional features of the **Table** display, such as the ability to display computed values, and the ability to perform an action when a row is selected.

Completing Stage 11

Import the sample project **Tutorial 11-12** and use it as the basis for the next stage of the tutorial.

For a reminder on how to import a sample project, see the **Importing a Sample Project** section at the end of **Stage 2**.

This sample project contains all the functionality modeled thus far.

You may now proceed to **Stage 12**, in which we will learn how to perform actions conditionally.

See It Live

Object 10

Click here to open the live project in a separate window.

Stage 12 – Controlling Application Flow

Stage Goals

Tersus Concepts Covered

On completion of this stage you should be familiar with the following concepts:

Modeling notions:	Activated slots, Validation
Modeling techniques:	Controlling process flow, Controlling popup appearance
Useful process templates:	Branch
Useful display templates:	Alert

Application Functionality Modeled

We shall continue modeling the issuing of **Purchase Orders (POs)** by the purchaser, by adding the following features:

- An alert displayed when at least one of the requisitions is marked as urgent, prompting the purchaser to process them.
- An **Order** button used to issue a purchase order to a supplier.

This stage's modeling should be performed in the **Tutorial 11-12** project, you imported at the end of the previous stage.

User Modeling

Use a Branch Template

When the list of approved requisitions is displayed, we would like to check whether it includes requisitions whose **Urgency** is **Urgent**.

This check should be performed immediately after the approved requisitions are retrieved from the database:

Zoom to **Generate Approved Requisition List**.

We shall add a process which checks if a requisition is urgent:

Add a **Basic/Action**. Name it ***Check if Requisition is Urgent***. Add to it a **trigger** and an **exit**. Name the exit ***Yes***. Mark the process as **repetitive**.

Create flow linking **Find/<Records>** to the trigger of **Check if Requisition is Urgent**.

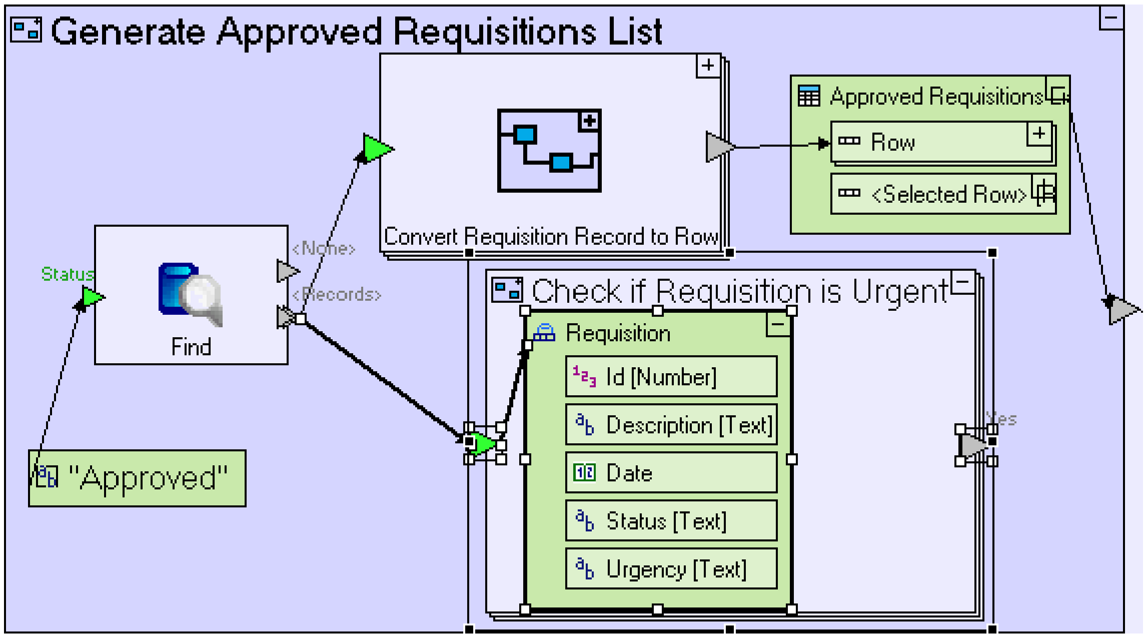
Next we model the actual urgency check:

Zoom into **Check if Requisition is Urgent**.

Drag a **Requisition** database record.

Create a flow linking the trigger of **Check if Requisition is Urgent** to **Requisition**.

The **Generate Approved Requisition List** model should look similar to the following:



The check is performed using a new template **Branch**, which accepts a value as input and activates the exit whose name matches the value.

Add a **Flow Control/Branch** template ( ).

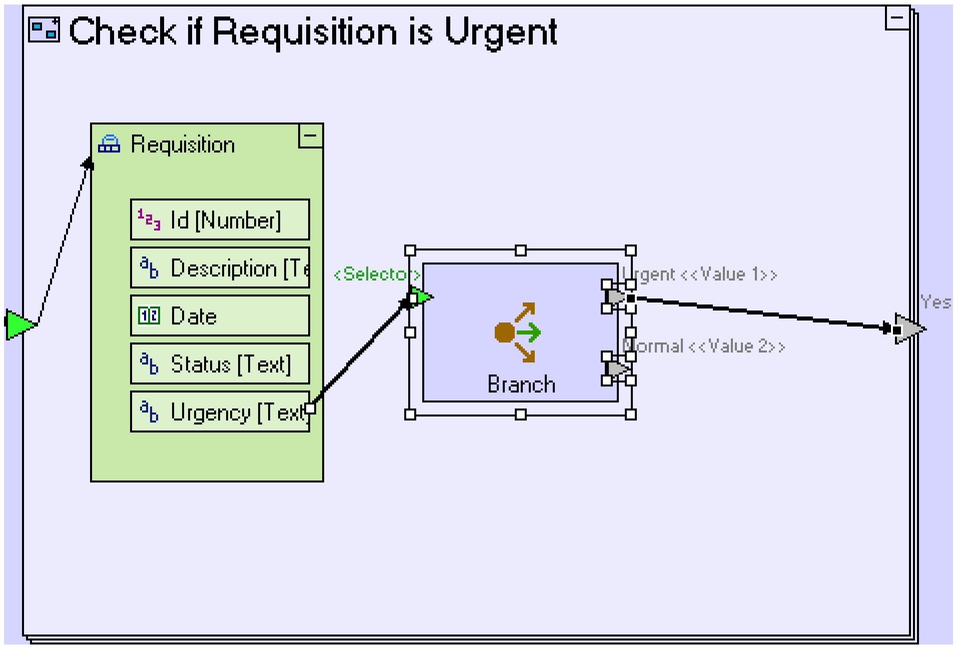
Create a flow linking **Requisition/Urgency** to the **<Selector>** trigger of **Branch**.

We need to distinguish between **Urgent** and **Normal** values:

Rename (pressing **[F2]**) the **Value 1** exit to ***Urgent***, and **Value 2** exit to ***Normal***

Create a flow linking the **Branch/Urgent** exit to the **Yes** exit.

The **Check if Requisition is Urgent** model looks as follows:



When an urgent requisition is received, **Branch** will activate its **Urgent** exit, causing an application flow that activates the **Yes** exit of **Check if Requisition is Urgent**.

Activating an exit means that regardless of whether the exit outputs a value or not, any flow originating from it will be activated, so that if the exit is linked to a trigger, that trigger will also be activated, which in-turn means that the process containing that trigger may be activated.

The functionality we are implementing does not need to perform anything when a non-urgent requisition is retrieved; therefore no flow is defined from the **Branch/Normal** exit. The exit can actually be deleted – we leave it here to improve the model's readability and for possible future enhancements of the model.

Use an Alert Template (to display an alert to the user)

If the **Check if Requisition is Urgent/Yes** exit is activated at least once, we would like to display an alert to the user. This is done using the **Alert** template.

Before we use the alert template, we should consider where in the model to position it. There are 3 options:

1. **Check if Requisition is Urgent** – in which case the alert will be displayed repeatedly for each urgent requisition.
2. **Generate Approved Requisitions List** – in which case the alert will be displayed only once, regardless of the number of urgent requisitions, but before the requisition list is populated (which occurs outside the process).
3. **Populate Approved Requisitions List** – in which case the alert will be displayed simultaneously with the requisition list being populated.

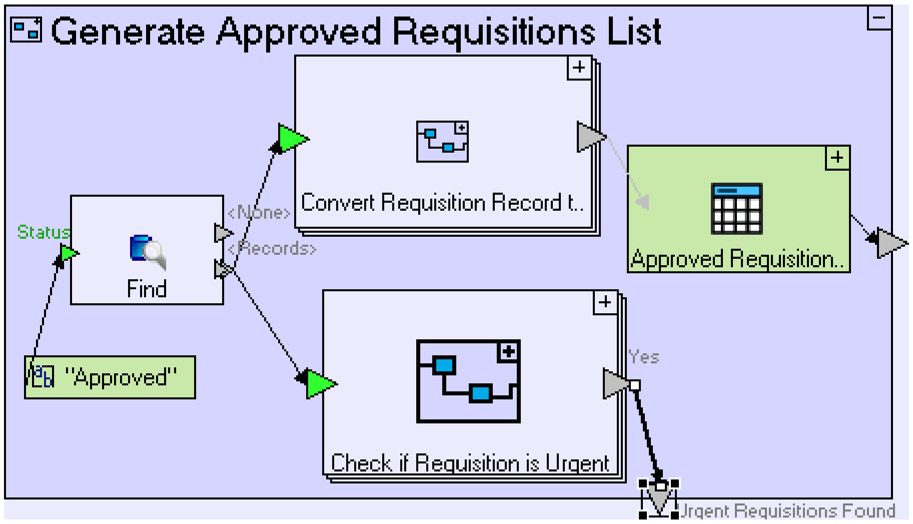
The third option seems the most plausible:

Zoom out to **Generate Approved Requisition List**.

Add an exit to it. Name it ***Urgent Requisitions Found***.

Create a flow linking the **Check if Requisition is Urgent/Yes** exit to the **Generate Approved Requisition List/Urgent Requisition Found**.

The **Generate Approved Requisition List** model should look similar to the following:


Generate Approved Requisitions List
Convert Requisition Record t..
Approved Requisition..
Status
Find
<None>
<Records>
"Approved"
Check if Requisition is Urgent
Yes
Urgent Requisitions Found

The alert will be displayed using the **Alert** template:

Zoom out to **Populate Approved Requisition List**.

The alert needs a message to feed its **<Message>** trigger:

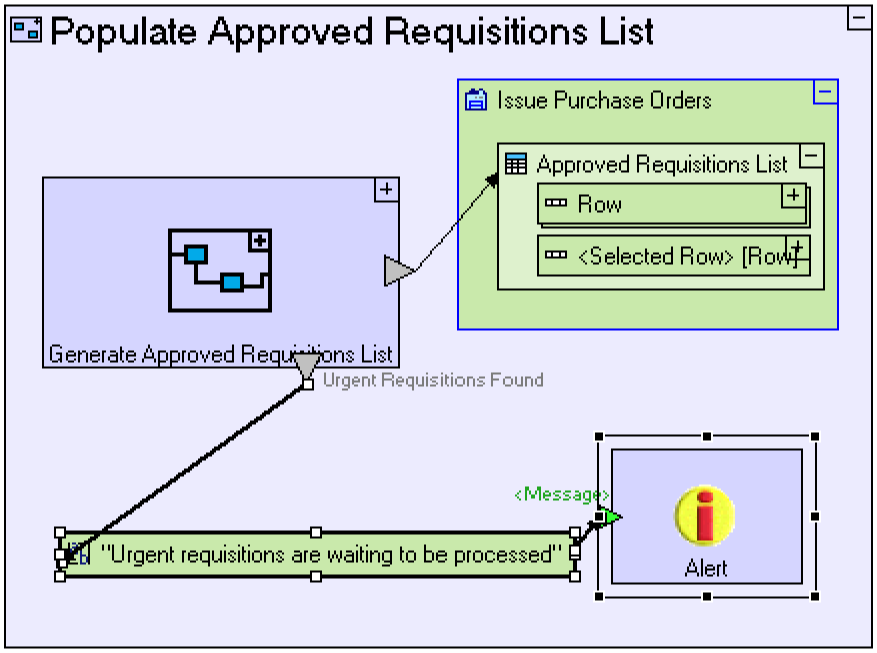
Add a **Constants/Text**. Name it ***Urgent requisitions are waiting to be processed***.

Create a flow linking the constant to **Alert/<Message>**.

The **Alert** will ultimately be activated by activating the constant holding it's message:

Create a flow linking **Generate Approved Requisition List/Urgent Requisition Found** to the "**Urgent requisitions are waiting to be processed**" constant.

The **Populate Approved Requisition List** model should now look similar to the following:

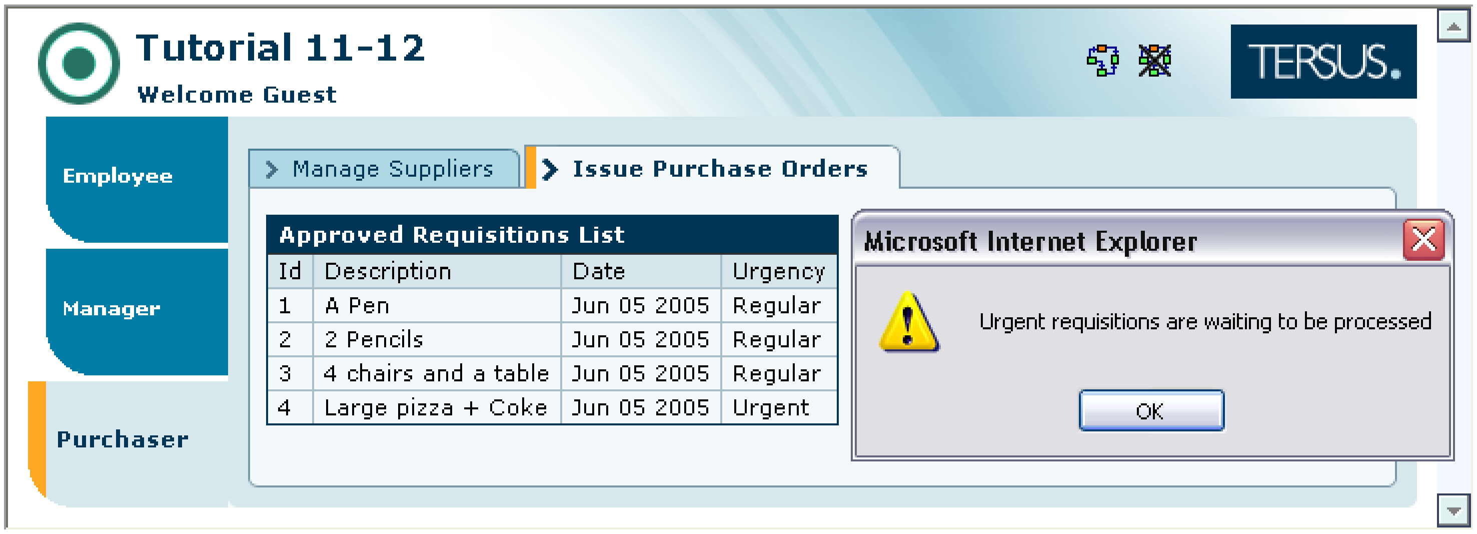

Populate Approved Requisitions List
Issue Purchase Orders
Approved Requisitions List
Row
<Selected Row> [Row]
Generate Approved Requisitions List
Urgent Requisitions Found
<Message>
"Urgent requisitions are waiting to be processed"
Alert

Save your work, and view the application in the browser.

Click the **Purchaser** perspective tab.

Click the **Issue Purchase Orders** view tab.

You should see an alert popup, as in the screenshot below.



Approved Requisitions List

Id	Description	Date	Urgency
1	A Pen	Jun 05 2005	Regular
2	2 Pencils	Jun 05 2005	Regular
3	4 chairs and a table	Jun 05 2005	Regular
4	Large pizza + Coke	Jun 05 2005	Urgent

Urgent requisitions are waiting to be processed

Display a Popup Conditionally

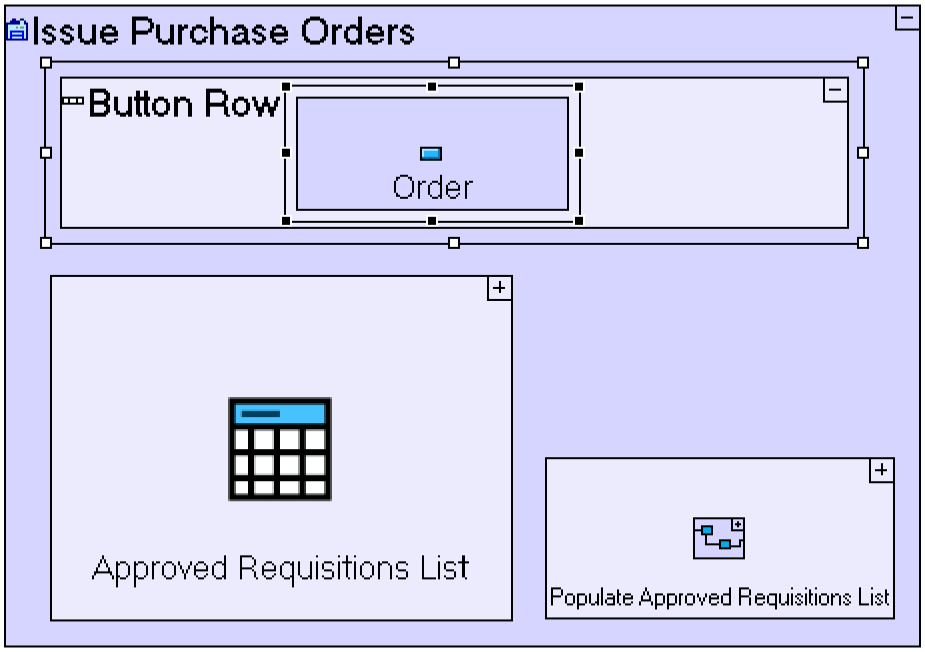
We shall now add an **Order** button to the view. It will be used in subsequent stages to issue a purchase order for a selected requisition:

Zoom out to the **Issue Purchase Orders** view.

Add a **Display/Row**. Name it **Button Row**. Zoom into it.

Add a **Display/Button**. Name it ***Order***.

The **Issue Purchase Orders** view model should now look similar to the following:


Issue Purchase Orders
Button Row
Order
Approved Requisitions List
Populate Approved Requisitions List

This button should open a popup for entering purchase orders, but it makes sense only if a requisition has been selected from the list:

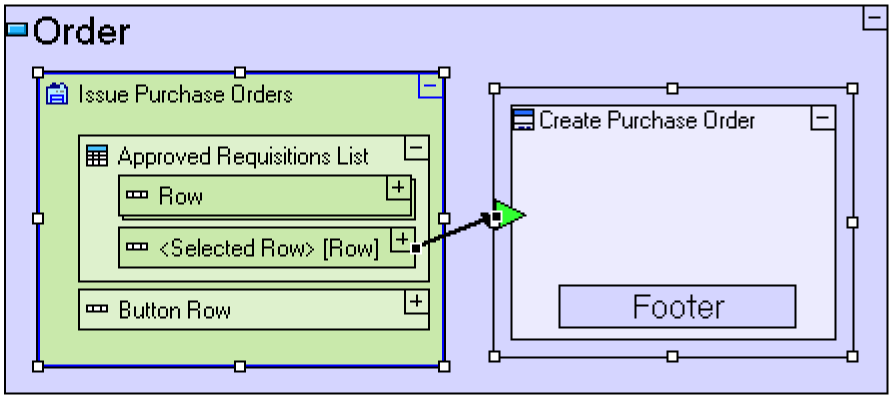
Zoom into the **Order** button.

Add an ancestor reference to the **Issue Purchase Orders** view.

Add a **Display/Popup**. Name it ***Create Purchase Order***. Add a **trigger** to it.

Create a flow linking **Issue Purchase Orders/Approved Requisitions List/Selected Row** to the trigger of **Create Purchase Order**.

The **Order** button model should look similar to the following:



The **Create Purchase Order** popup will only open if **Get Table Selection** exits through the **<Selected Row>** exit, and since the selected **Requisition Row** data structure is also passed in the flow, we will have immediate access the details of the selected requisition inside **Create Purchase Order**, without having to retrieve it again.

Save your work, and view the application in the browser.

Click the **Purchaser** perspective tab.

Click the **Issue Purchase Orders** view tab.

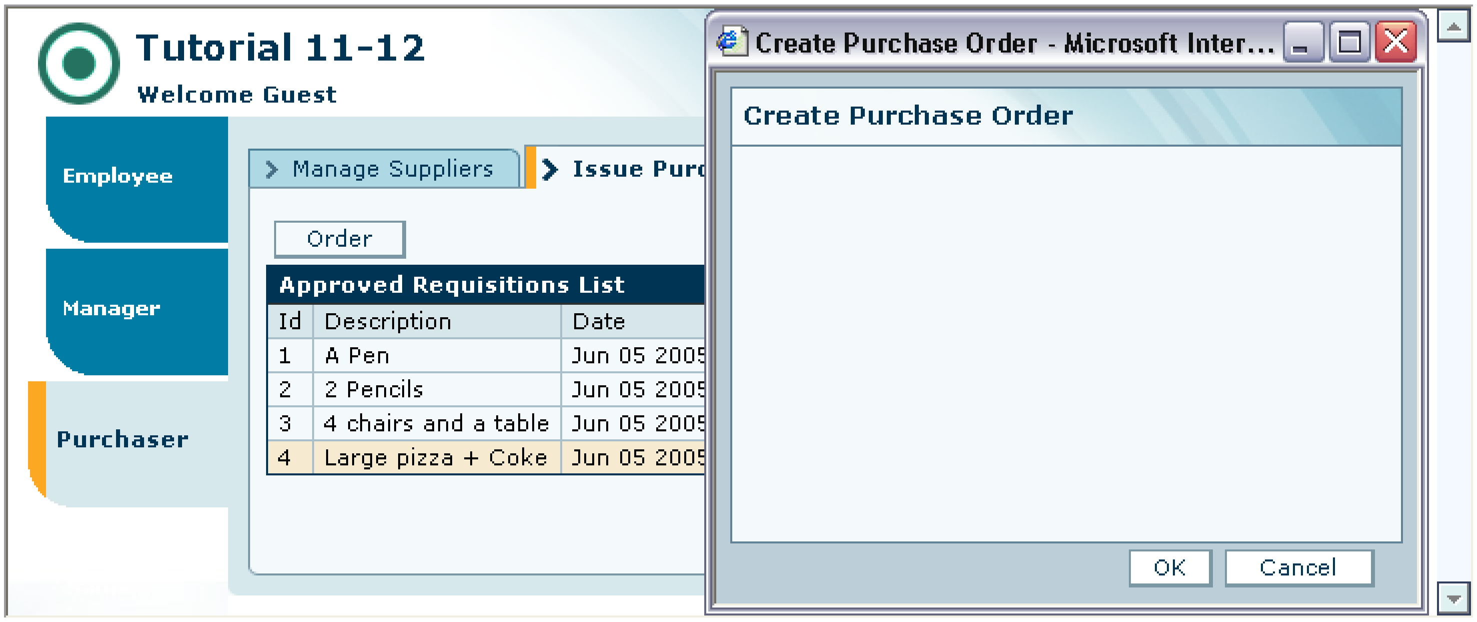
Click the **Order** button (before selecting a row).

Nothing should happen, as no row has been selected.

Select a row from **Requisition List**.

Click the **Order** button.

The **Create Purchase Order** popup should appear looking similar to the following:



Approved Requisitions List

Id	Description	Date
1	A Pen	Jun 05 2005
2	2 Pencils	Jun 05 2005
3	4 chairs and a table	Jun 05 2005
4	Large pizza + Coke	Jun 05 2005

Completing Stage 12

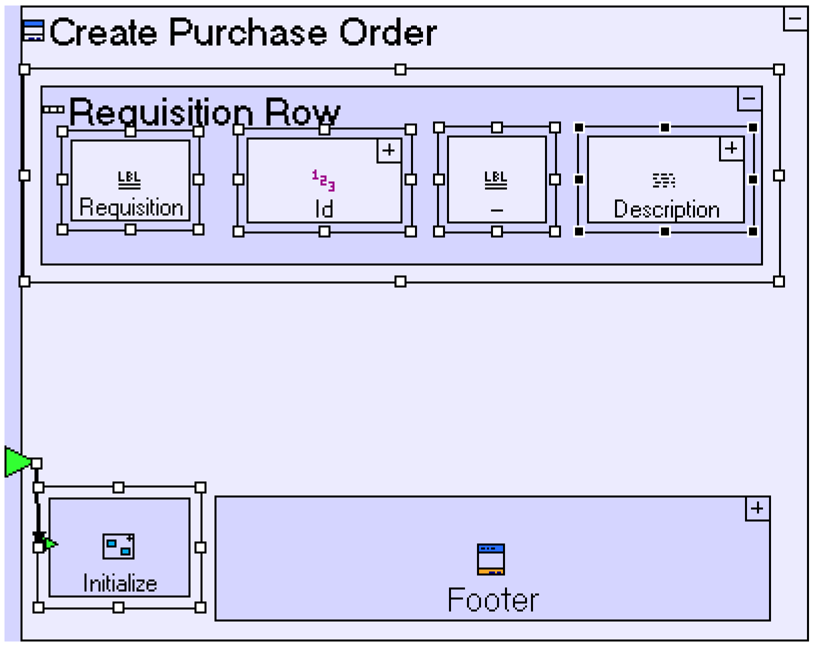
Import the sample project **Tutorial 12-13** and use it as the basis for the next stage of the tutorial.

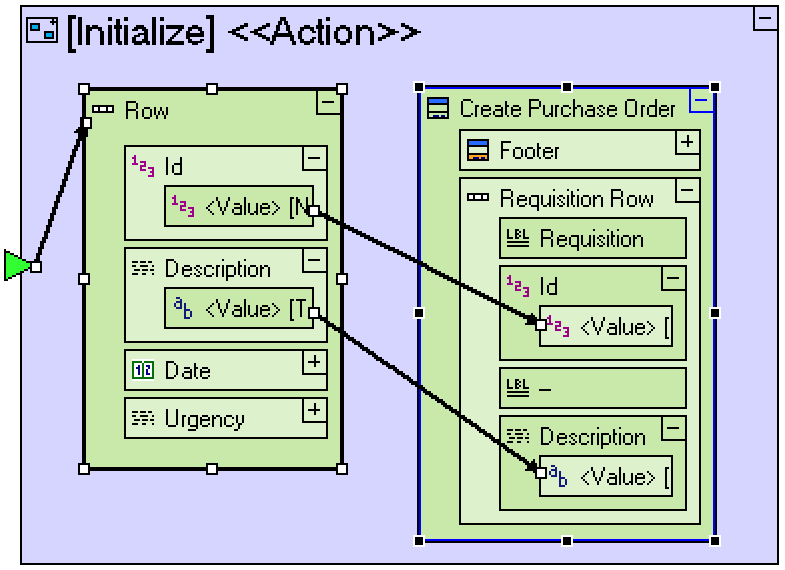
For a reminder on how to import a sample project, see the **Importing a Sample Project** section at the end of **Stage 2**.

This sample project contains all the functionality modeled thus far.

The sample project also includes additional functionality, as follows:

1. Add display elements, for displaying **Requisition** details in the **Create Purchase Order** popup. Initialize it with data from the selected row.

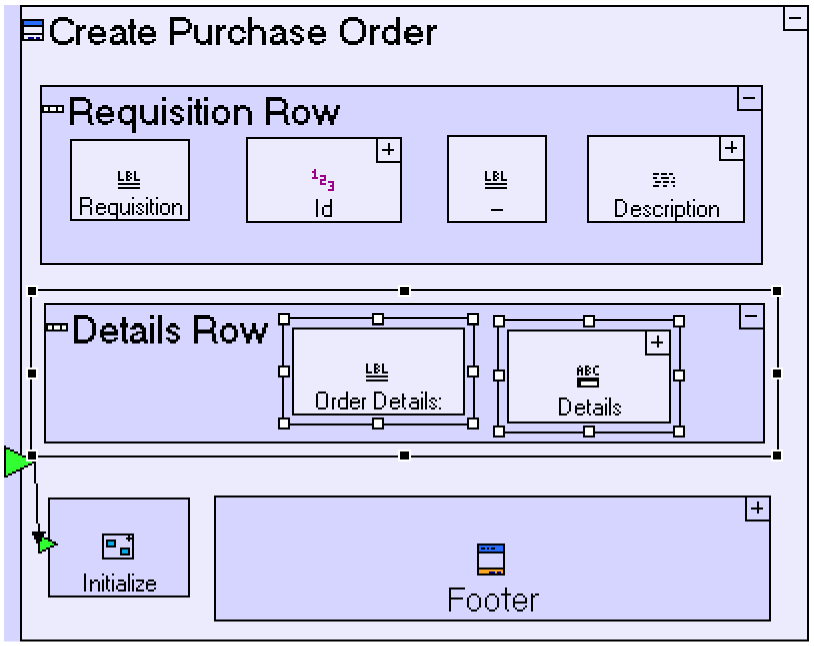


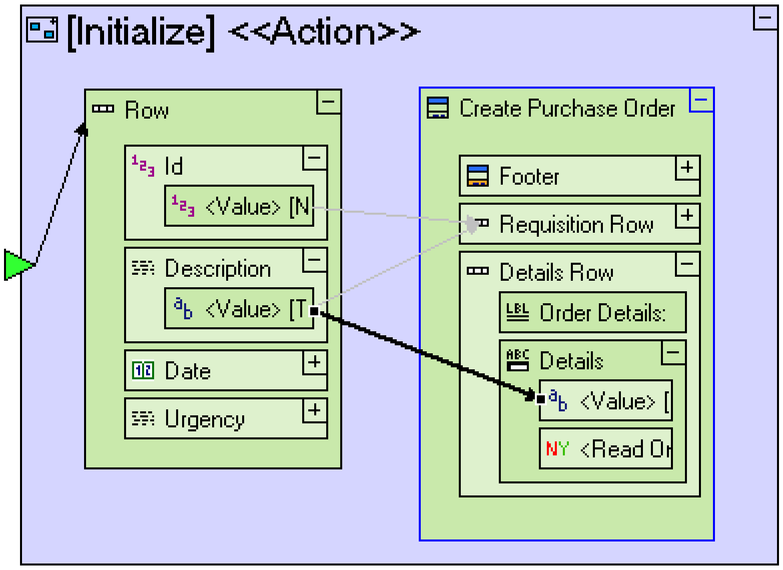


How to Model	Located in
Add **Display/Row**. Name it ***Requisition Row***. Add a **Basic/Action**. Name it ***Initialize***. Add to it a trigger. Create a flow linking the trigger of **Create Purchase Order** to the trigger of **Initialize**.	**Create Purchase Order** popup
Add a **Display/Label**. Name it ***Requisition***. Add a **Display/Number Display**. Name it ***Id***.	**Requisition Row**

Add a **Display/Label**. Name it – (hyphen). Add a **Display/Text Display**. Name it ***Description***.	
Drag a **Requisition Row** (to create a display data element). Link the trigger of **Initialize** to it. Add an ancestor reference to the **Create Purchase Order** popup. Link **Requisition Row/Id/<Value>** to **Create Purchase Order/Requisition Row/Id/<Value>**. Link **Requisition Row/Description** to **Create Purchase Order/Requisition Row/Description/<Value>**.	**Initialize** process

2. Add a **Details** input field to the **Create Purchase Order** popup. Initialize the **Details** field with the **Requisition**'s **Description**

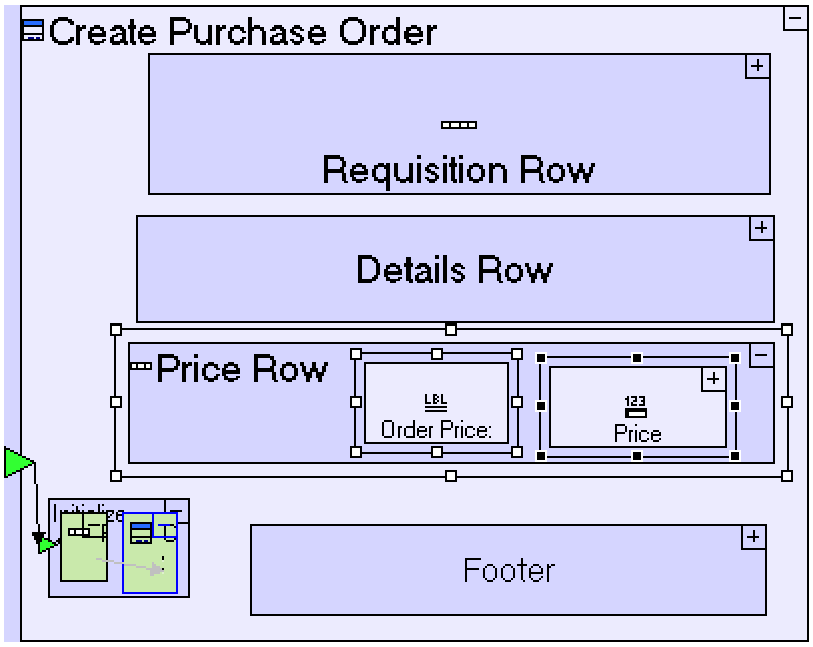




How to Model	Located in
Add **Display/Row**. Name it ***Details Row***.	**Create Purchase Order** popup
Add **Display/Label**. Name it ***Order Details:***.	**Details Row**

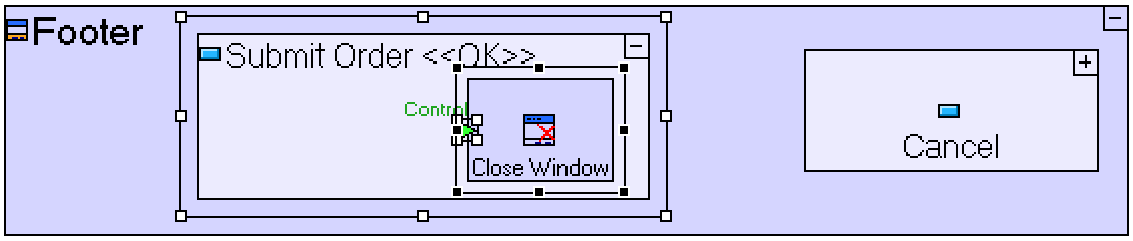
Add **Display/Text Input Field**. Name it ***Details***.	
Link **Requisition Row/Description/<Value>** to **Create Purchase Order/Details Row/Details/<Value>**.	**Initialize** process

3. Add a **Price** input field to the **Create Purchase Order** popup.



How to Model	Located in
Add **Display/Row**. Name it ***Price Row***.	**Create Purchase Order** popup
Add **Display/Label**. Name it ***Order Price:***. Add **Display/Number Input Field**. Name it ***Price***.	**Price Row**

4. Create a **Submit Order** button, used after selecting a supplier to issue the purchase order.



How to Model	Located in
Rename the **OK** button to ***Submit Order***.	**Create Purchase Order/Footer**
Add **Display Actions/Close Window**. Add a **Control** trigger (through **Add Element)**.	**Submit Order** button

Note that in the sample project provided, the application model includes a validation warning, due to the fact that the **Close Window** sub-process of the **Submit Order** button has a mandatory trigger (**Control**) with no flow into it, which means that as it stands, clicking the **Submit Order**, will not close the popup.
This issue will be solved in the next stage, once flow is defined in the next stage.

You may now proceed to **Stage 13**, in which we are going to learn techniques for managing relational data**.**

See It Live

Object 11

Click here to open the live project in a separate window.

Stage 13 – Modeling Relational Data

Stage Goals

Tersus Concepts Covered

On completion of this stage you should be familiar with the following concepts:

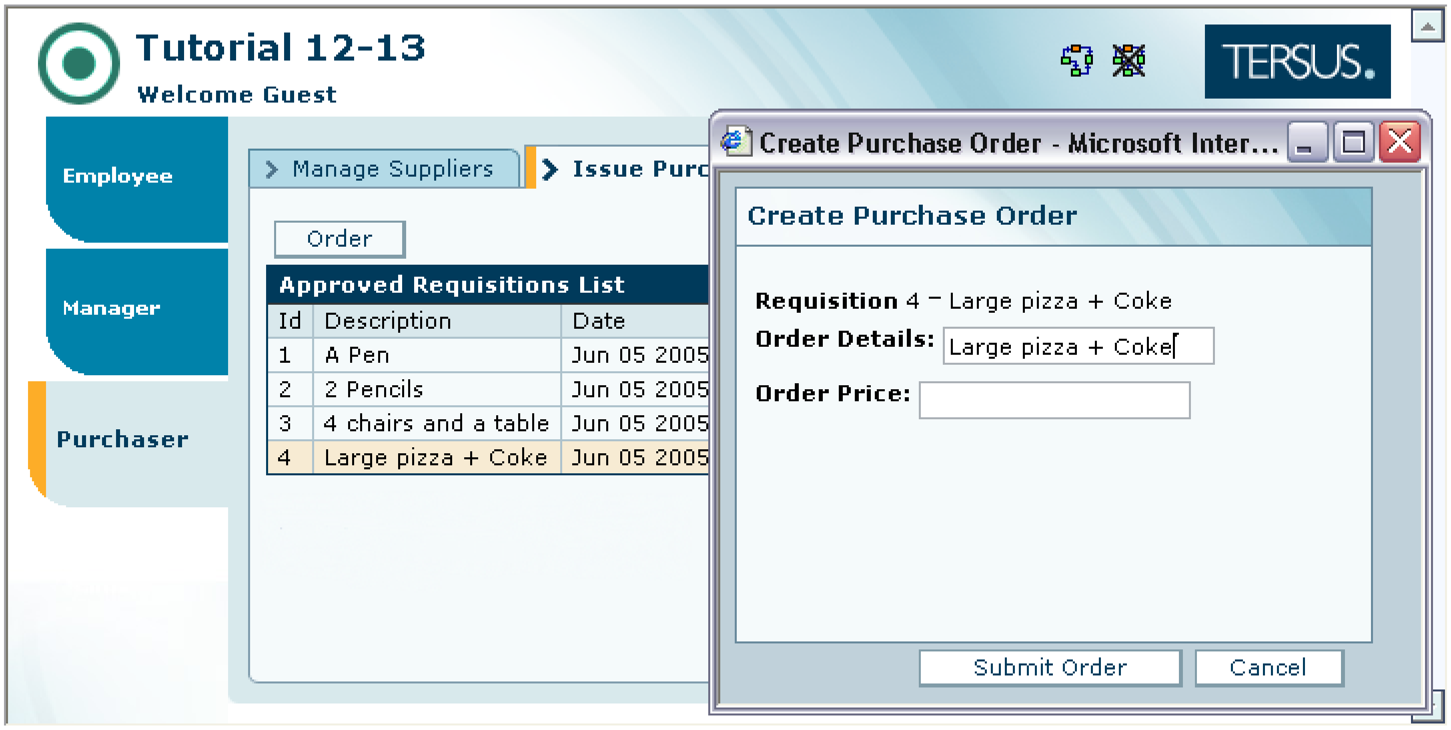
Modeling notions:	Service vs. Action processes, Automatic transaction support, <Done> exit
Modeling techniques:	Modeling relational data, Maintaining referential integrity, Chooser based on a composite data structure
Useful process templates:	Service

Application Functionality Modeled

The sample project you have loaded contains modeling of the **Create Purchase Order** popup user interface:

- The selected requisition's **Id** and **Description**.
- A **Details** input field. Initialized with the requisition's description, in case the purchase order's content is identical to the requisition. The purchaser may update the purchase requisition details.
- A **Price** input field (the order's total price).
- A **Submit Order** button, which contains a **Close Window** process that is **not** executed since it has no flow attached to it.

If you view the application in the browser, it should look as follows:



Approved Requisitions List

Id	Description	Date
1	A Pen	Jun 05 2005
2	2 Pencils	Jun 05 2005
3	4 chairs and a table	Jun 05 2005
4	Large pizza + Coke	Jun 05 2005

Requisition 4 – Large pizza + Coke

Order Details: Large pizza + Coke

Order Price:

The Create Purchase Order popup needs additional modeling:

4. A **Supplier** chooser displaying a list of suppliers retrieved from the **Supplier** database table.
5. The **Submit Order** button needs additional modeling to handle database operations: storing the purchase order details entered by the user and updating the requisition to signify that an order has been issued.

This stage's modeling should be performed in the **Tutorial 12-13** project, you imported at the end of the previous stage.

User Modeling

Use a Chooser Based on a Data Structure

We would like to add a drop-down list displaying a list of suppliers:

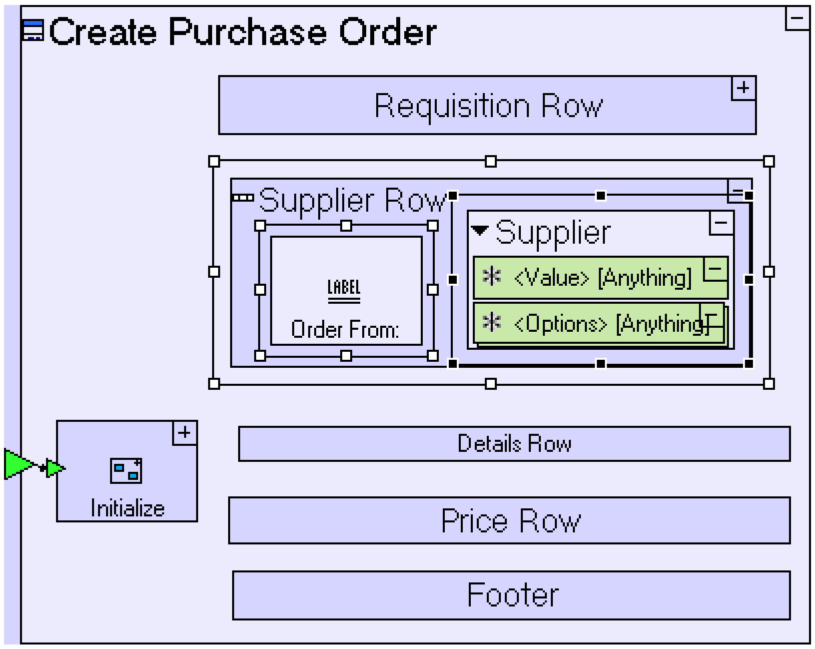
Zoom to **Create Purchase Order** popup.

Add a **Display/Row**. Name it ***Supplier Row***. Zoom into it.

Add **Display/Label**. Name it ***Order From:***.

Add **Display/Chooser**. Name it ***Supplier***.

The **Create Purchase Order** popup model should look similar to the following:



Recall that when we previously used the **Chooser** display (to specify requisition urgency), we populated it with text values ("**Regular**" and "**Urgent**") – defined as constants.

This time the values to be displayed in the chooser, the supplier names, must be retrieved from the **Supplier** table. We will populate **Supplier/<Options>** with **Supplier** records we retrieve from the database, and the chooser will automatically display the company name of each supplier (since **Company Name** is the first text field in the **Supplier** data structure):

Zoom to the popup's **Initialize** process.

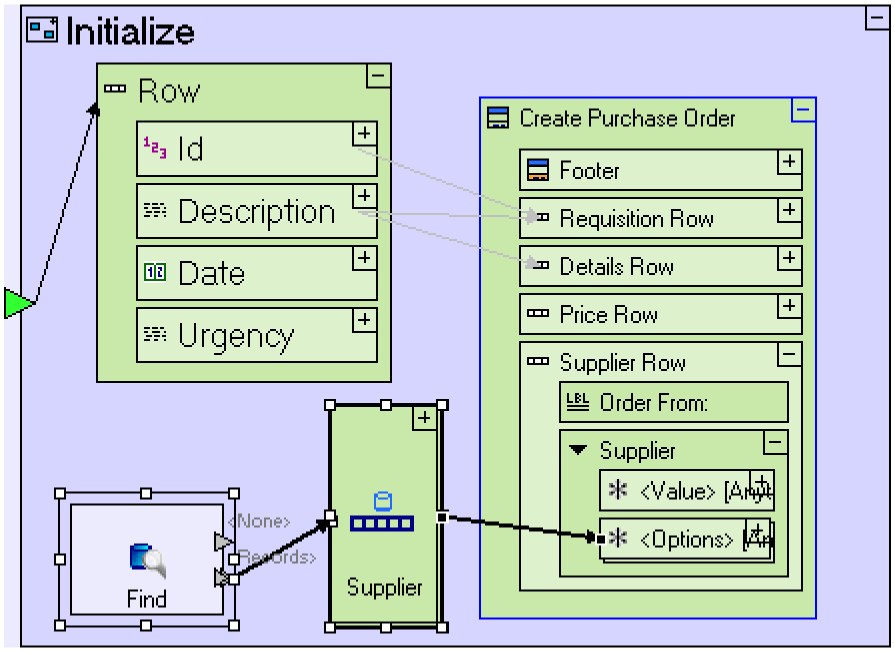
Add a **Database/Find**.

Drag a **Supplier** database record (used in the **Manage Suppliers** view, **Supplier List** table) from the outline/repository. Set it repetitive.

Create a flow linking **Find/<Records>** to the **Supplier** data structure.

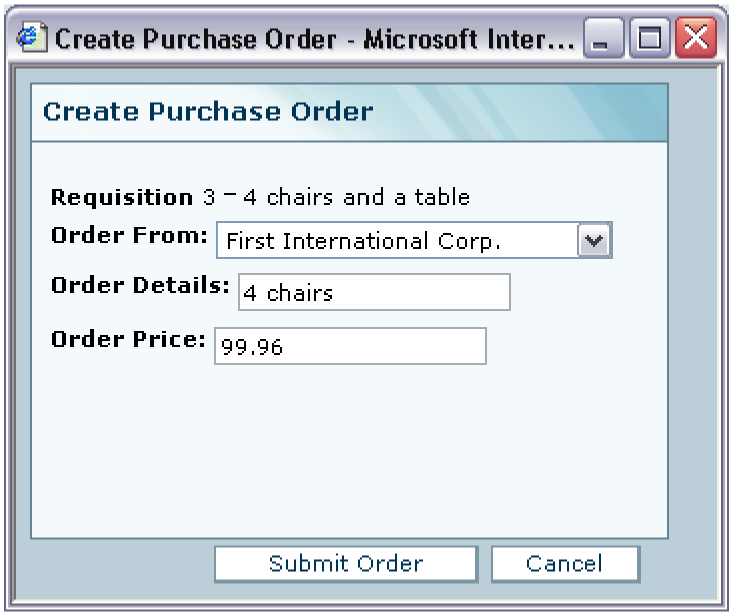
Create a flow linking the **Supplier** data structure to **Create Purchase Order/Supplier Row/Supplier/<Options>**.

The **Initialize** model should look similar to the following:



Notice that while the drop-down list displays only the company name of each supplier, once the user chooses a supplier, **Supplier/<Value>** will hold the selected supplier's record in full (we will be interested in the **Supplier/<Value>/Id** field later), without having to retrieve the details once more from the database.

Save your work, and view the application in the browser. It should look similar to the following:


Create Purchase Order - Microsoft Inter...

Create Purchase Order

Requisition 3 – 4 chairs and a table
Order From: First International Corp.
Order Details: 4 chairs
Order Price: 99.96

Use a Service Template (when a process must run on the server)

Now we turn to completing the modeling of the **Submit Order** button. We shall add a process to the **Submit Order** button, to handle the database issues mentioned above.

Up to this point in the tutorial, we primarily used the **Action** template as a container for our modeling. Using **Action** as a container lets the Tersus runtime engine decide where each of its sub-models is to be executed – on the client or on the server.

In most situations it is obvious where a process should be executed. When its purpose is to display (or reference) user interface elements, it must, and will automatically, execute on the client (browser). Opposite examples are the various database actions, which are always executed on the server.

There are, however, situations where a process can be executed on both tiers, and in these situations, the Tersus runtime engine will execute processes on the client side, unless they are explicitly assigned by the modeler to run on the server. The main factor in the modeler's decision where a process should be executed is performance. As a rule of thumb, it is a good idea to locate processes manipulating a lot of data or accessing centrally managed data on the server tier. When there is a process with a lot of internal flow of data, and some of its sub-processes are server-side only, it makes sense to perform the whole process on the server, to avoid a lot of traffic between the server and the client.

Aside from performance issues, there are other situations where running a model on the server side is recommended if not imperative. One of these situations is the case of database **transactions**. When the model performs multiple database operations, such as updates to multiple records in one or more tables in a database and these updates are related to one another, all updates must either succeed or fail, as one.

Tersus provides an automated transaction mechanism. Assigning a process to run on the server ensures that when it is called from a client process, a separate database transaction will be opened. The transaction will be committed if and only if the process is successful (and it will be rolled back if the process is not successful).

To define that a process is to be executed on the server, use the **Service** template:

Zoom to **Submit Order**.

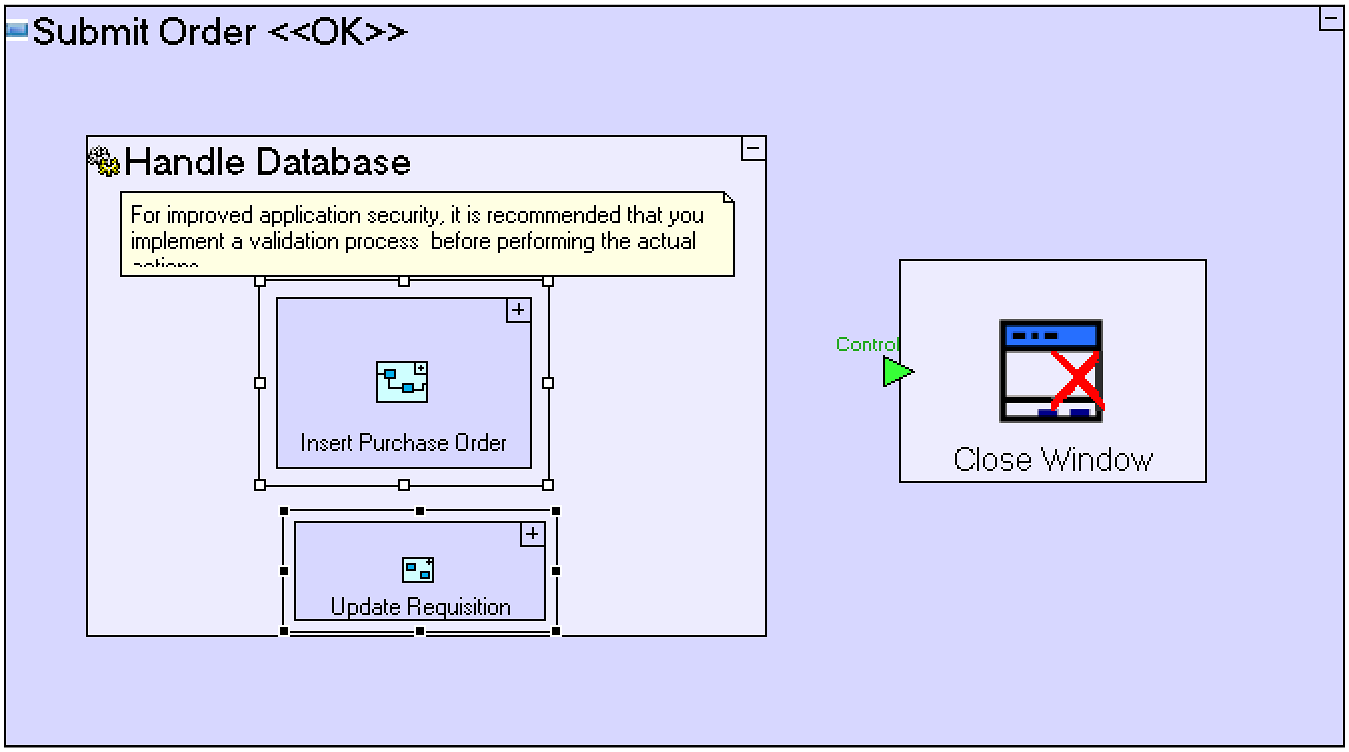
Add **Basic/Service** (). Name it ***Handle Database***. Zoom into it.

The **Handle Database** process will include 2 sub-processes, one to handle the purchase order, and the other to handle the requisition's status update:

Add a **Basic/Action**. Name it ***Insert Purchase Order***.

Add a **Basic/Action**. Name it ***Update Requisition***.

The **Submit Order** button model should look similar to the following:


Submit Order <<OK>>
Handle Database
For improved application security, it is recommended that you implement a validation process before performing the actual
Insert Purchase Order
Update Requisition
Control
Close Window

Note that both sub-processes are based on the generic **Action** template, and not the **Service** template – they will execute on the server, because they are contained in **Handle Database**, which is a **Service**. Both sub-processes do not create separate transactions. They are included in the parent transaction, so the insert and update will either both succeed or both fail.

Define a Database Record (for the purchase order)

Begin with the definition of the **Purchase Order** database record:

Zoom into **Insert Purchase Order**.

Add a **Data Types/Database Record**. Name it ***Purchase Order***. Zoom into it.

Each purchase order should include the following data:

1. A unique identifier:

   Add **Data Types/Number**. Name it ***Id***.

2. The requisition for which the purchase order is issued. We shall store the requisition's identifier identifying it in the **Requisitions** table:

   Add **Data Types/Number**. Name it ***Requisition Id***.

3. The supplier to which the purchase order is issued. We shall store the supplier's identifier identifying it in the **Suppliers** table:

   Add **Data Types/Number**. Name it ***Supplier Id***.

Readers inexperienced with relational data models may find it surprising that we are storing the **Id**s of the **Requisition** & **Supplier**, instead of the actual data which will be displayed to the user when referencing a purchase order (in the case of the **Supplier** table this will usually be **Company Name** or **Contact Name**). In fact, this is the correct way to model relational data, since the **Id** will never change, whereas the details may change frequently (**Contact Name** is a good example). Once you have the **Id** at hand, it is very easy to find and display all supplier details.

4. The details of the purchase order:

   Add **Data Types/Text**. Name it ***Details***.

5. The date of issuing the order:

   Add **Data Types/Date**.

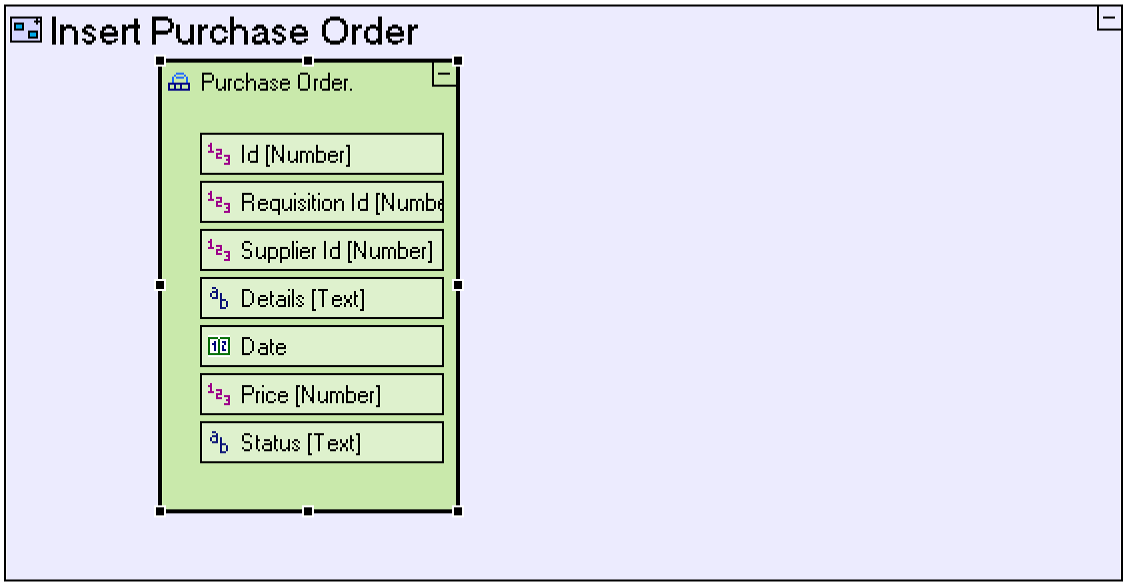
6. The price agreed with the supplier:

   Add **Data Types/Number**. Name it ***Price***.

7. The order's status (issued or delivered):

   Add **Data Types/Text**. Name it ***Status***.

The **Insert Purchase Order** model should look similar to the following:



Positioning an Ancestor Reference Correctly (not in a service)

Now that the **Purchase Order** data structure is defined, we should populate it with the relevant data, part of which comes from the **Create Purchase Order** popup, so we need to use an **Ancestor Reference**.

The question is where in the model should this data population process be located?

We might want to place it in the **Insert Purchase Order** process, but since the process is executed on the server, it cannot reference the popup (a client side element) directly. The solution is to place the ancestor reference in an ancestor process and pass the data to the process through triggers. Since the process' grandparent (**Handle Database**) is a **Service** process, we shall place the ancestor reference in the "great-grandparent" process:

Zoom out to **Submit Order**.

Add an ancestor reference to **Create Purchase Order**.

There are four data elements which we need to retrieve from the display: The requisition's identifier, the chosen supplier's identifier, the purchase order details, and the price. We shall add a process which creates the database record and passes it to **Handle Database** and into **Insert Purchase Order**, so flow and triggers must be defined:

Add a **Basic/Action**. Name it ***Create Purchase Order Record***.

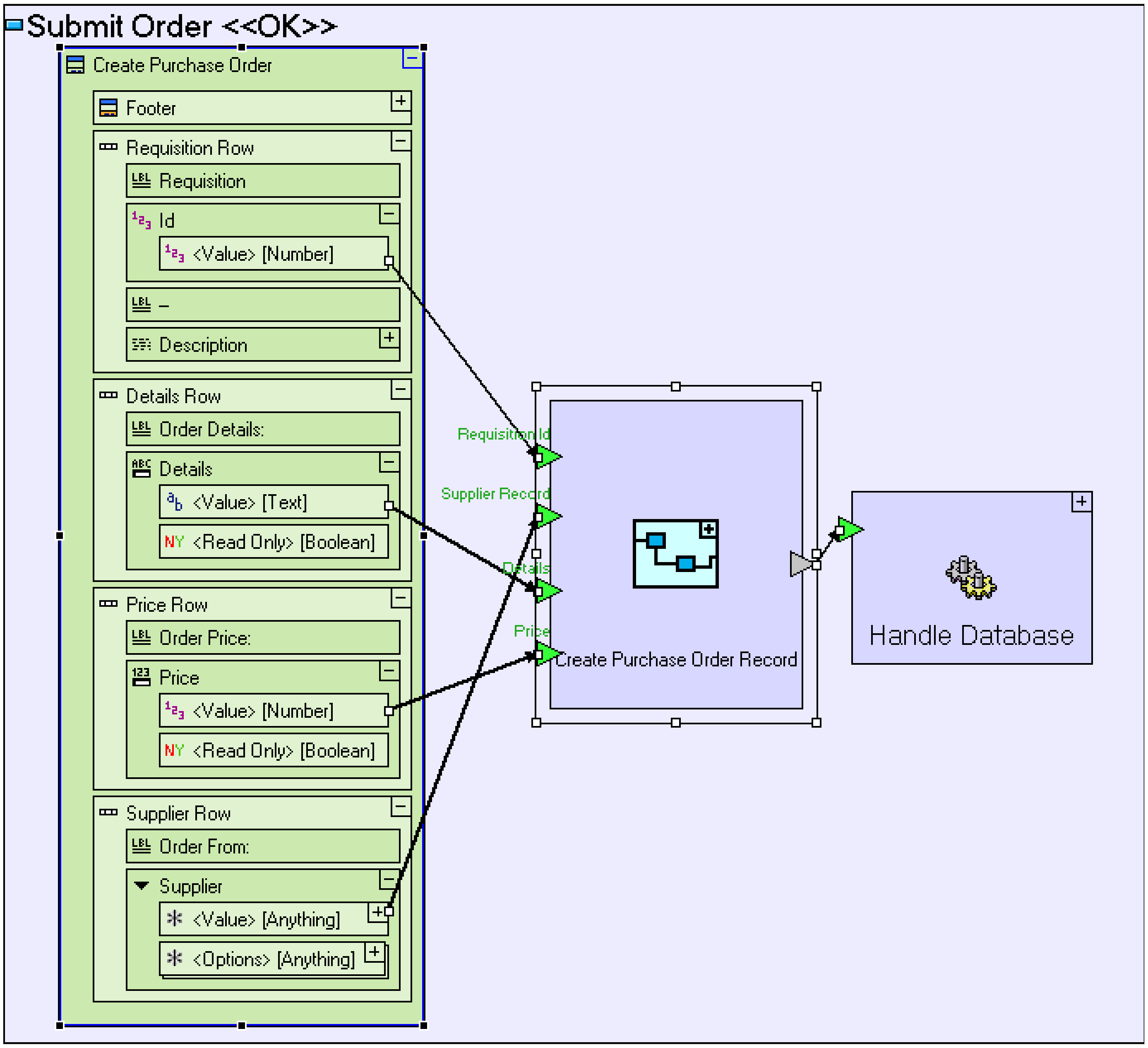
Add 4 triggers to **Create Purchase Order Record**. Name them ***Requisition Id***, ***Supplier Record***, ***Details*** and ***Price***.

Create the following 4 flows:

Source (in **Create Purchase Order**)	Target (**Create Purchase Order Record** trigger)
Requisition Row/Id/<Value>	**Requisition Id**
Supplier Row/Supplier/<Value>	**Supplier Record**
Details Row/Details/<Value>	**Details**
Price Row/Price/<Value>	**Price**

Add an exit **Create Purchase Order Record**. Create a flow linking the exit to the **Handle Database/** trigger.

The model should look as follows:



Populate the Purchase Order Record (with purchase order details)

Zoom into **Create Purchase Order Record**.

Drag a **Purchase Order** database record from the repository/outline.

Create a flow linking **Purchase Order** to the **Create Purchase Order Record** exit.

1. The **Requisition Id** should be set to the **Id** of the **Requisition** data structure received through the **Requisition Id** trigger:

   Create a flow linking the **Requisition Id** trigger to **Purchase Order/Requisition Id**.

2. The **Supplier Id** should be set to the **Id** of the supplier selected by the user. In our case, the drop-down list was populated with **Supplier** database records, so we can extract the

Id from the selected record:

Drag a **Supplier** data structure from the repository/outline.

Create a flow linking the **Supplier Record** trigger to the **Supplier** data structure.

Create a second flow linking **Supplier/Id** to **Purchase Order/Supplier Id**

3. The **Details** should be set to the text entered by the user in the popup, which is passed through the **Details** trigger:

   Create a flow linking the **Insert Purchase Order/Details** trigger to **Purchase Order/Details**.

4. The **Price** should be set to the value entered by the user in the popup, which is passed through the **Price** trigger:

   Create a flow linking the **Insert Purchase Order/Price** trigger to **Purchase Order/Price**.

5. The **Date** field should be set to today's date:

   Add a **Dates/Today**.

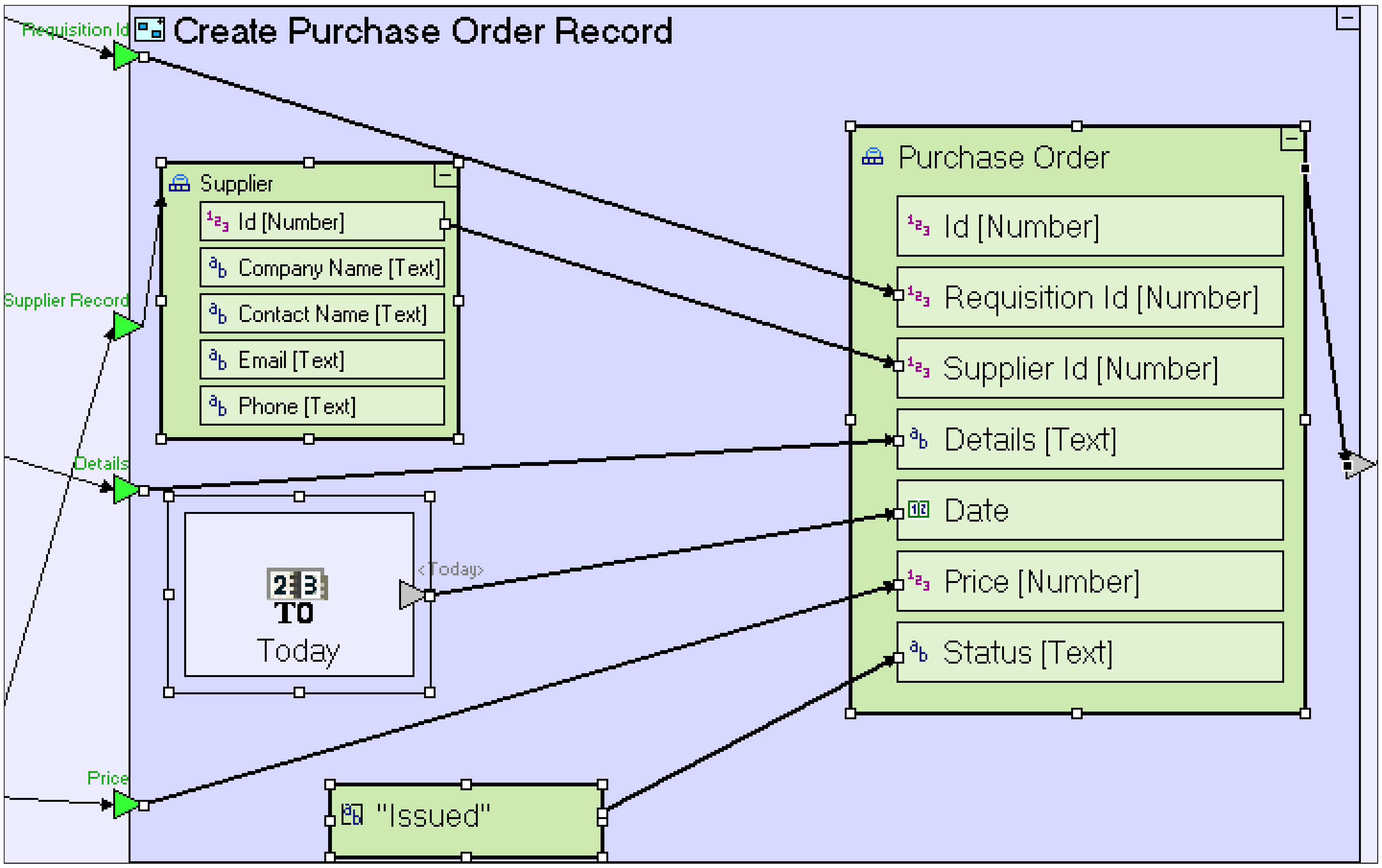
   Create a flow linking **Today/<Today>** to **Purchase Order/Date**.

6. The purchase order's **Status** should be set as issued:

   Add a **Constants/Text**. Name it ***Issued***.

   Create a flow linking "**Issued**" to **Purchase Order/Status**.

The **Create Purchase Order Record** process model should look similar to the following:



Notice that the **Purchase Order/Id** field has not been set. This is because it should be a unique identifier set using a **Database/Sequence Number** and therefore should be part of the database transaction, which means it must be set in the **Handle Database** service.

Zoom to **Handle Database**.

Add a trigger to **Insert Purchase Order**.

Create a flow linking the **Handle Databae** trigger to the **Insert Purchase Order** trigger.

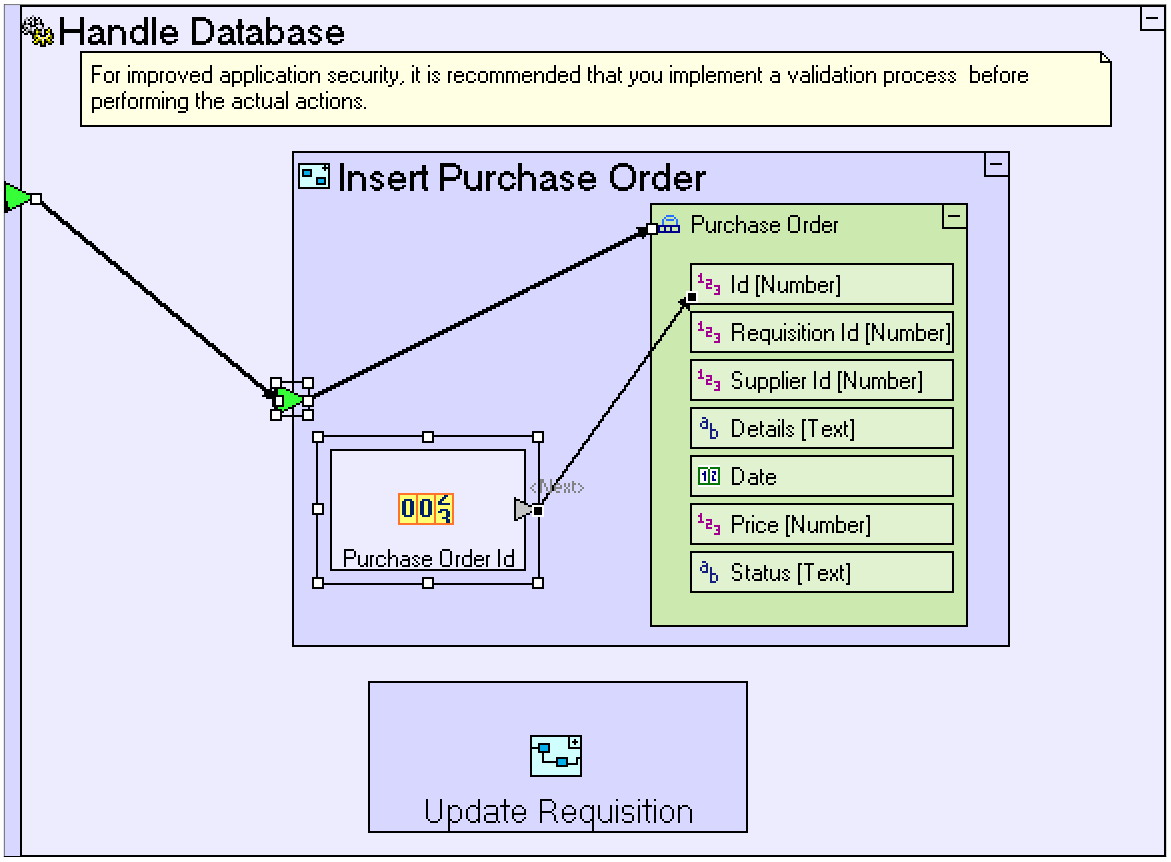
Zoom into **Insert Purchase Order**.

Create a flow linking the **Insert Purchase Order** trigger to the **Purchase Order** database record.

Add **Database/Sequence Number**. Name it ***Purchase Order Id***.

Create a flow linking **Purchase Order Id/<Next>** to **Purchase Order/Id**.

The **Handle Database** process model should look similar to the following:



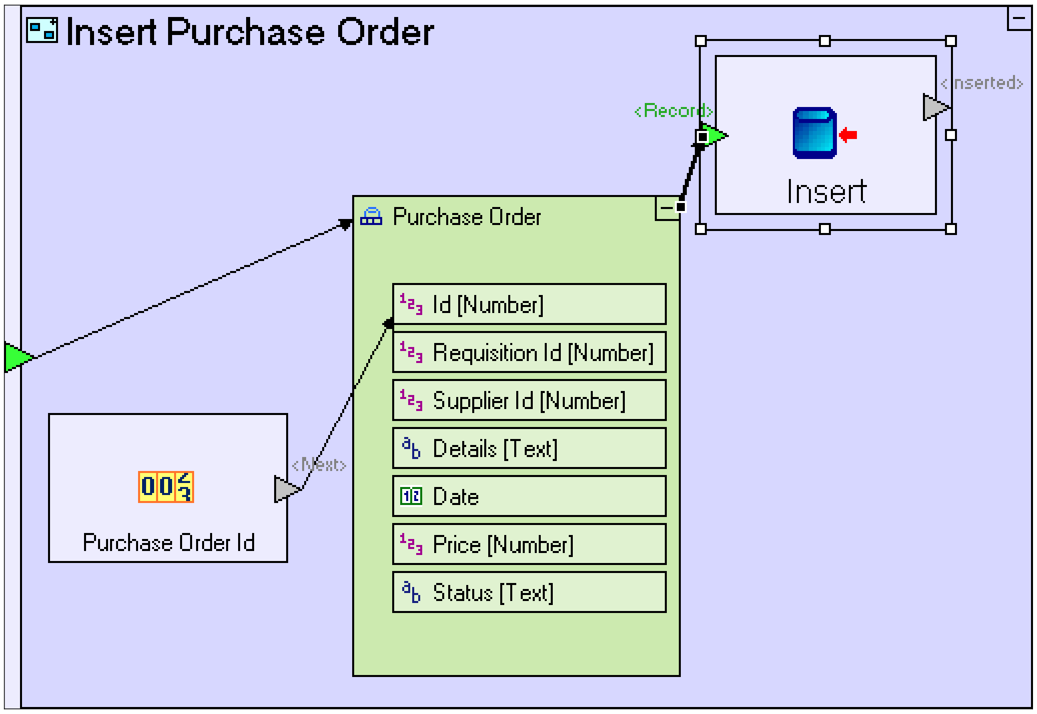
Now that the **Purchase Order** database record is fully populated, we can add it to the **Purchase Orders** table:

Zoom to **Insert Purchase Order**.

Add a **Database/Insert**.

Create a flow linking the **Purchase Order** data structure to **Insert/<Record>.**

The **Insert Purchase Order** process should look as follows:



Update the Requisition

Now we move to the **Update Requisition** process. The requisition should be updated with a new status that signifies an order has been issued for that requisition. The requisition's identifier is available in the **Purchase Order** record we created, so:

Zoom to **Handle Database**.

Add a trigger to **Update Requisition**.

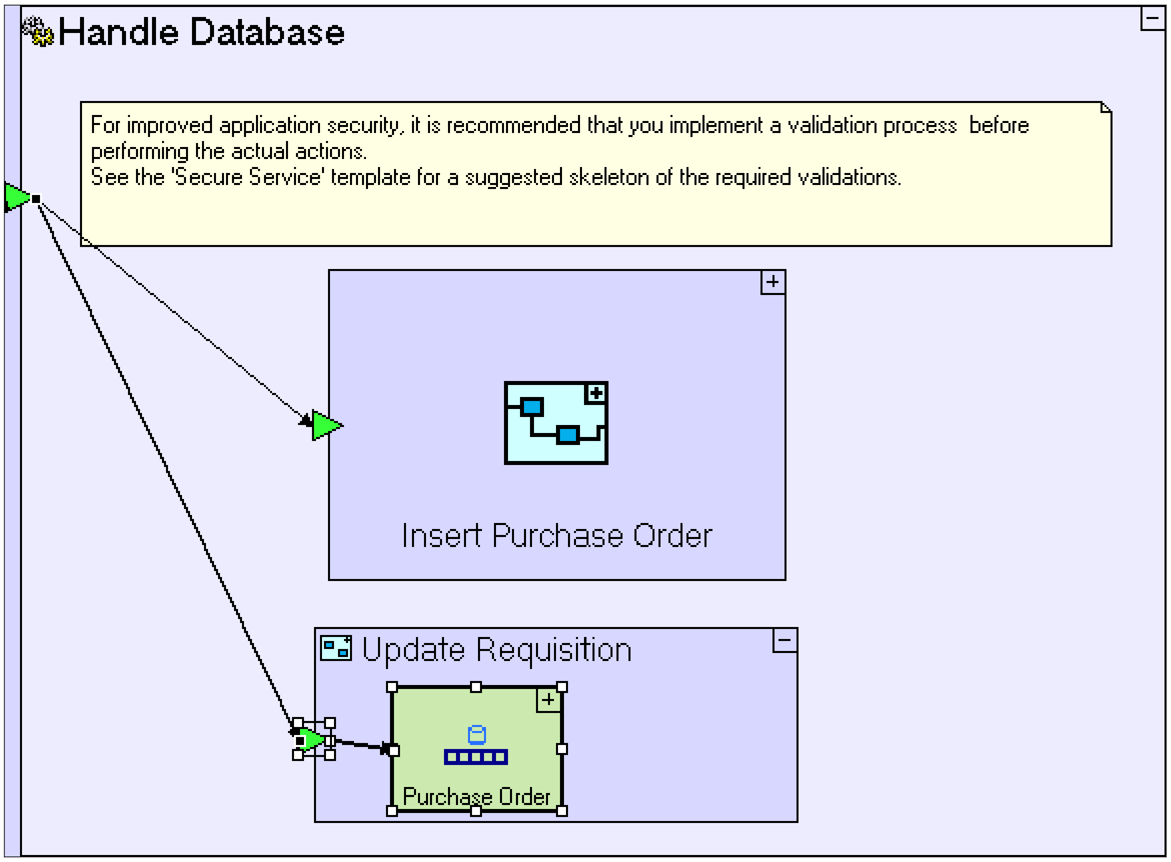
Create a flow linking the **Handle Database** trigger to the **Update Requisition** trigger.

Zoom into **Update Requisition**.

Drag a **Purchase Order** database record from the repository/outline.

Create a flow linking the **Update Requisition** trigger to the **Purchase Order** database record

The **Handle Database** process model should look similar to the following:



For improved application security, it is recommended that you implement a validation process before performing the actual actions.
See the 'Secure Service' template for a suggested skeleton of the required validations.

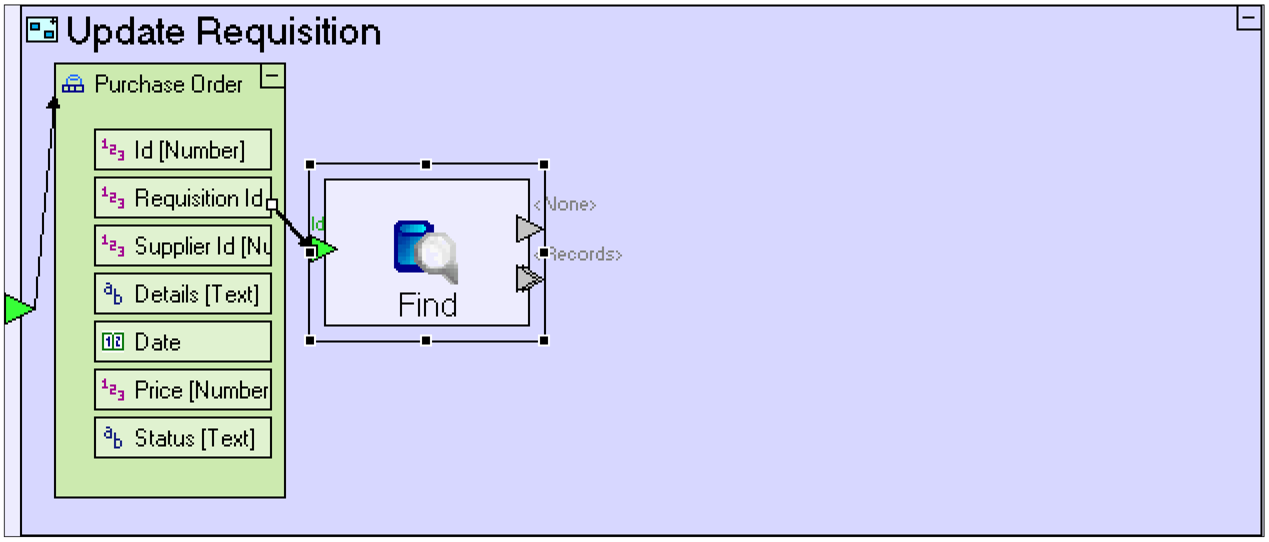
We need to retrieve the requisition from the database and update its status:

Zoom to **Update Requisition**.

Add a **Display/Find**. Add a new trigger. Name it ***Id***.

Create a flow linking **Purchase Order/Requisition Id** to **Find/Id**.

The **Update Requisition** model should look similar to the following:



We can reuse **Change Requisition Status**, a process we modeled in a previous stage (in **Manager** system/**Requisition Approval** view/**Button Row**/**Approve Requisition** button/**Update Requisition** process) in order to perform the requisition status update:

Drag **Change Requisition Status** from the repository/outline.

Create a flow linking the **Find/<Records>** to **Change Requisition Status/Original Requisition**.

Add a **Constants/Text**. Name it ***Order Issued***.

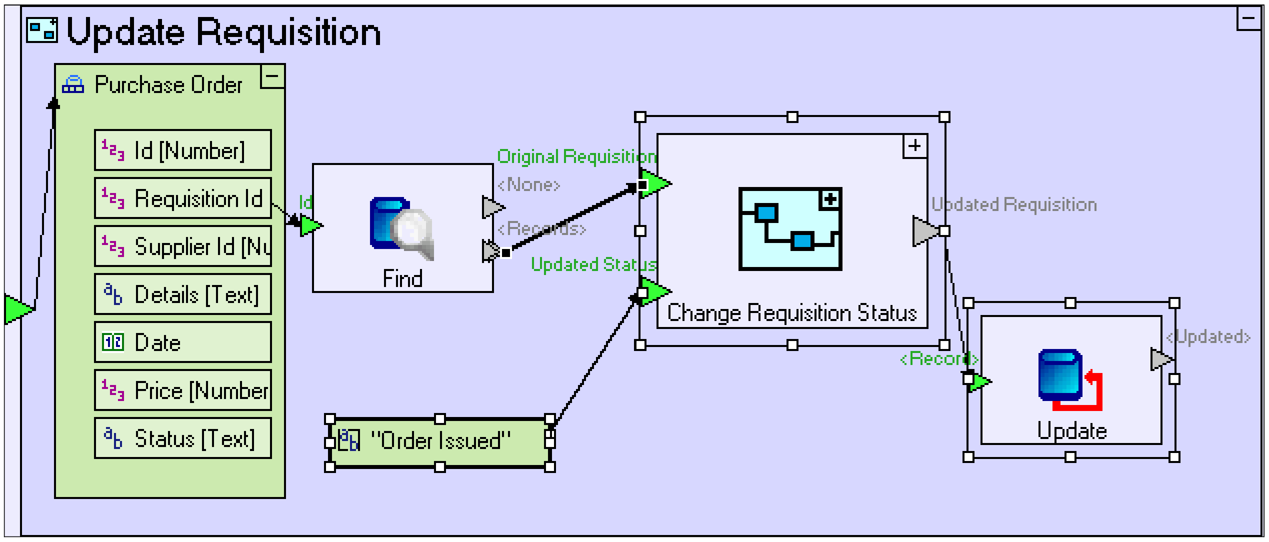
Create a flow linking "**Order Issued**" to **Change Requisition Status/Updated Status**.

Finally we need to perform the update in the database:

Add a **Database/Update**.

Create a flow linking **Change Requisition Status/Updated Requisition** to **Update/<Record>**.

The **Update Requisition** process should look similar to the following:



Use a <Done> Exit (to specify the order of execution)

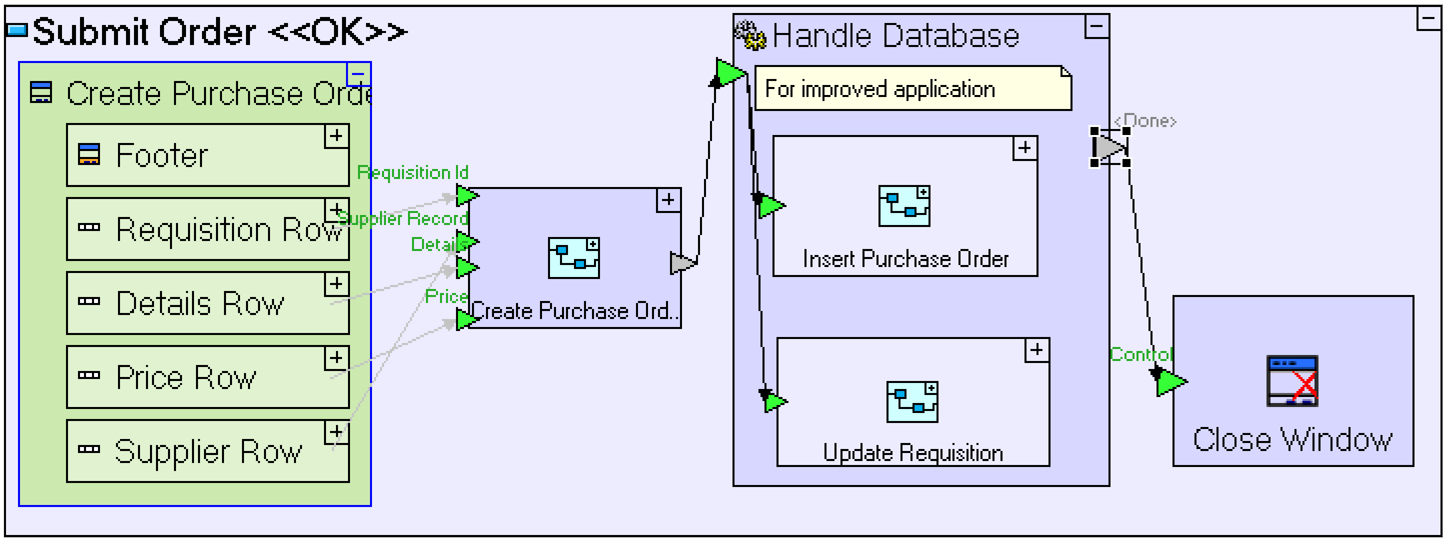
Now that the database issues are taken care of, we should take care of one last issue – ensuring that **Close Window** executes only after **Handle Database** has finished:

Zoom out to **Submit Order**.

Right-click **Handle Database**. From the **Add Element** sub-menu Select the **<Done>** exit.

Create flow linking **Handle Database/<Done>** to the trigger of **Close Window**

The **Submit Order** process should look similar to the following:



Naming a process' exit **<Done>** ensures that it will be activated when the process has completed executing, even though there is no flow (inside **Handle Database**) activating it. If the exit is named anything but <**Done**> (or remains unnamed), it will not be activated.

An alternate method to creating the **<Done>** method would be to add a regular exit, and name it **<Done>**. You will then need to set the exit's type to **Nothing**, by selected **Data Types/Nothing** and dropping it on the exit. You can see a slot's type, when you rename it (the type is the **Model Name**).

Save your work, and launch the application in a browser.

Click the **Purchaser** perspective tab.

Click the **Issue Purchase Orders** view tab.

Select a row from **Requisition List**.

Click the **Order** button.

Select a **Supplier** and enter **Price**.

Click the **Submit Order** button.

In the next stage we shall model the display of purchase orders related to each requisition.

Completing Stage 13

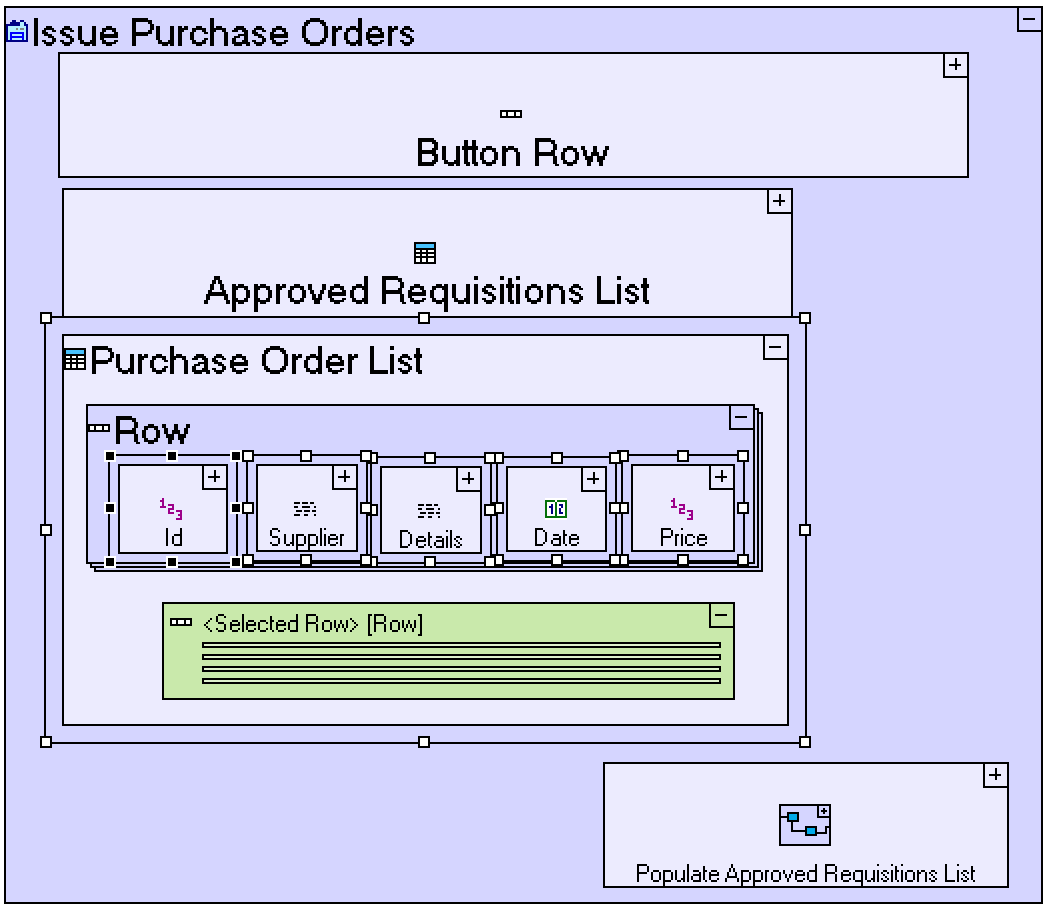
Import the sample project **Tutorial 13-14** and use it as the basis for the next stage of the tutorial.

For a reminder on how to import a sample project, see the **Importing a Sample Project** section at the end of **Stage 2**.

This sample project contains all the functionality modeled thus far.

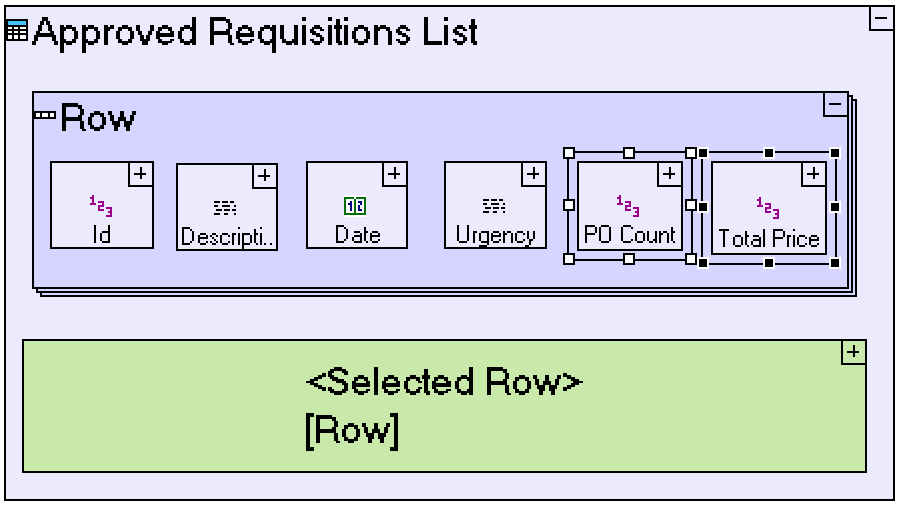
The sample project also includes additional functionality, as follows:

1. Add a second table displaying purchase orders to the **Issue Purchase Orders** view.



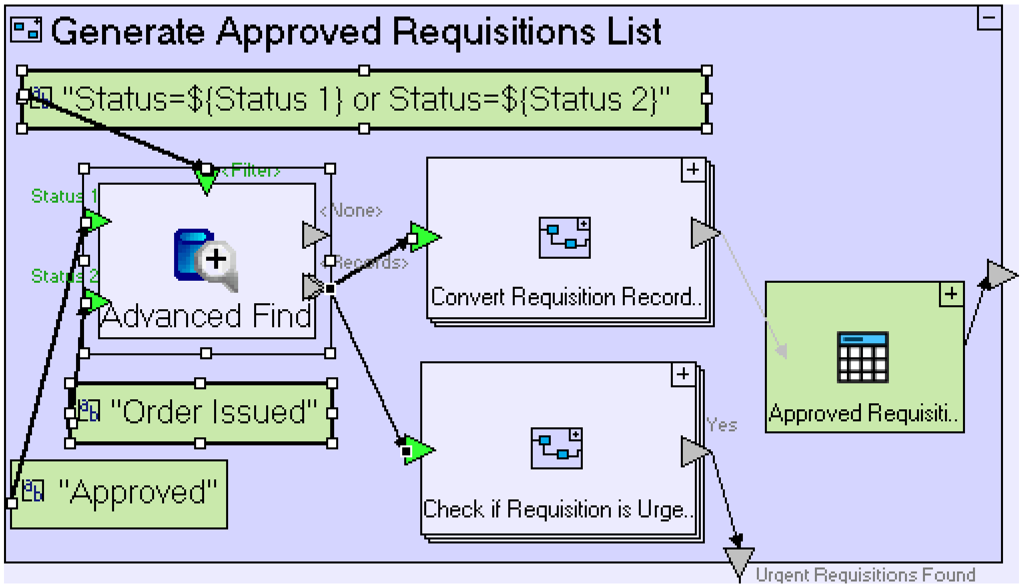
How to Model	Located in
Add **Display/Table**. Name it ***Purchase Order List***.	**Issue Purchase Orders** view
Add a **Display/Number Display**. Name it ***Id***. Add a **Display/Text Display**. Name it ***Supplier***. Add a **Display/Text Display**. Name it ***Details***. Add a **Display/Date Display**. Name it ***Date***. Add a **Display/Number Display**. Name it ***Price***.	**Purchase Order List** table/**Row**

2. Add columns to the **Approved Requisitions List** table.



How to Model	Located in
Add a **Display/Number Display**. Name it ***PO Count***. Add a **Display/Number Display**. Name it ***Total Price***.	**Approved Requisitions List** table/**Row**

3. Update **Generate Approved Requisitions List** to include requisitions with status **Order Issued**.


Generate Approved Requisitions List
"Status=${Status 1} or Status=${Status 2}"
<Filter>
Status 1
Status 2
<None>
<Records>
Advanced Find
Convert Requisition Record..
"Order Issued"
"Approved"
Check if Requisition is Urge..
Yes
Approved Requisiti..
Urgent Requisitions Found

How to Model	Located in
Replace **Find** with **Database/Advanced Find**. Re-link flows from the **Advanced Find/**<**Records**> exit. Add 2 triggers; name them ***Status 1*** and ***Status 2***. Link "**Approved"** to **Status 1**. Drag "**Order Issued**". Link it to **Status 2**. Add **Constants/Text**. Name it ***Status=${Status 1} or Status=${Status 2}***. Link it to **<Filter>**.	**Generate Approved Requisitions List** process

You may now proceed to **Stage 14**, in which we are going to make use of the **Purchase Order List** table to display purchase orders linked to the currently selected requisition and calculate purchase order totals for each requisition**.**

See It Live

Object 12

Click here to open the live project in a separate window.

Stage 14 – Displaying Multiple (Linked) Tables

Stage Goals

Tersus Concepts Covered

On completion of this stage you should be familiar with the following concepts:

Modeling notions:	<On Click> process
Modeling techniques:	Displaying one-to-many relationship, summing
Useful process templates:	Count, Sum
Useful display templates:	Refresh

Application Functionality Modeled

We shall continue with modeling the **Issue Purchase Order** View.

In the previous stage, we modeled the relationship between a purchase order and a requisition. Specifically the modeling is of a one-to-many relationship, where multiple purchase orders may be related to the same requisition, each containing some of the requested items.

The sample project you have loaded includes the following additions to the **Issue Purchase Order** view:

- A purchase order (PO) list appearing under the requisition list.
- 2 new columns added to the requisitions list: **PO Count** & **Total Price**

These additions to the view, do not display any data yet. We need to model the following:

6. Whenever a requisition is selected in the requisition list, the PO list is updated with the POs linked to the selected requisition.
7. Calculate **PO Count** and **Total Price** for each requisition, counting the number of POs linked to each requisition, and calculating their total price.

> This stage's modeling should be performed in the **Tutorial 13-14** project, you imported at the end of the previous stage.

User Modeling

<On Click> Process (to execute a process when a row is clicked)

Our first task is to create a process that updates the purchase order list when a requisition is clicked in the requisition list.

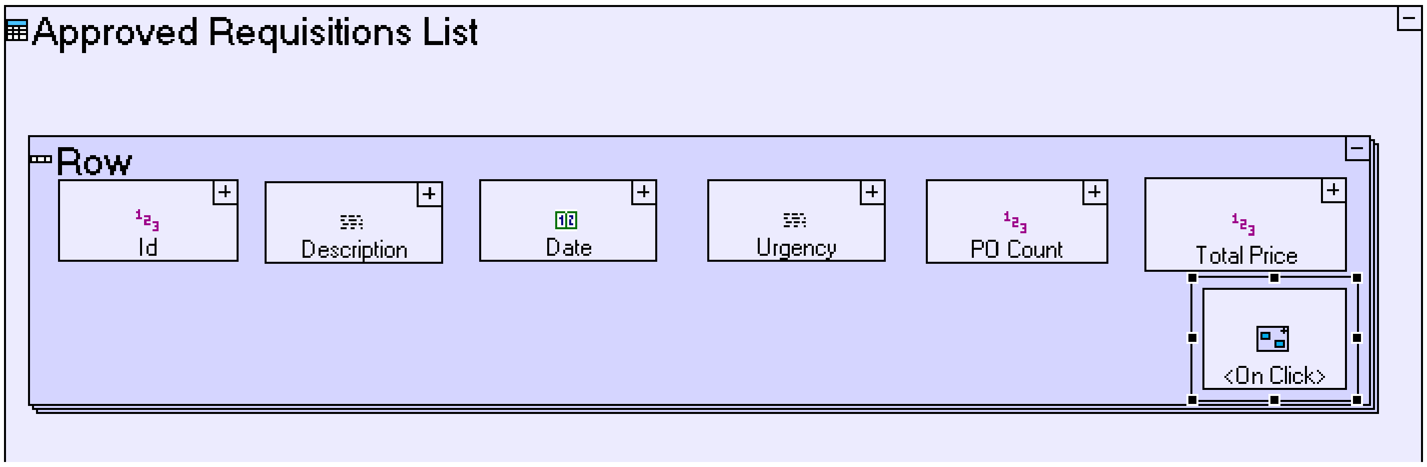
Tersus supports this through a process with a reserved name – **<On Click>**:

> Zoom to **Approved Requisitions List/Row**.
>
> Add a **Basic/Action**. Name it ***<On Click>***.

> Alternatively, you can add the **<On Click>** process, by opening (through right-click) the **Row** element's context menu and selecting it from the **Add Element** sub-menu.

> This is another example of a situation where a table display cannot be based on the **Simple Table** template – since the **<On Click>** process is part of the **Row** display element, while a **Simple Table** contains just a data structure.

The **Approved Requisitions List** model should look similar to the following:



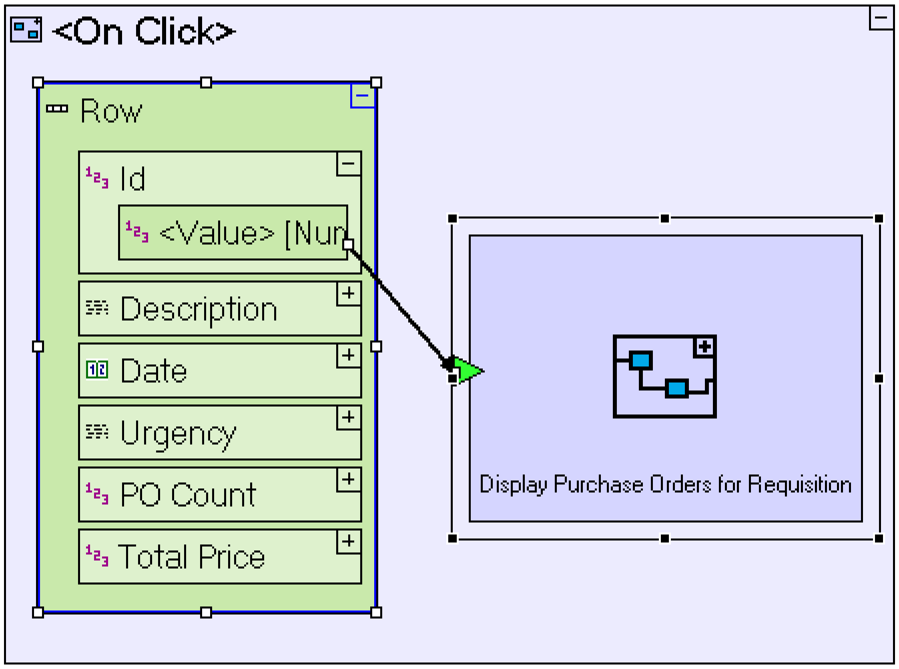
The <On Click> process should retrieve all purchase orders which are linked to the clicked requisition, so it first needs to get the Id of the requisition represented in this row. The requisition id for the the clicked row is available through an ancestor reference:

> Zoom into **<On Click>**.
>
> Add ancestor reference of **Row**.

Now we should create a sub-process which receives the requisition's **Id** and populates the **Purchase Order List** with the matching purchase orders:

> Add a **Basic/Action**. Name it ***Display Purchase Orders for Requisition***.
>
> Add a trigger to it, and create a flow linking **Requisition Row/Id/<Value>** to the trigger.

The **<On Click>** model should look similar to the following:



We will retrieve any purchase order whose **Requisition Id** field matches the **Id** of the current requisition:

Zoom into **Display Purchase Orders for Requisition**.

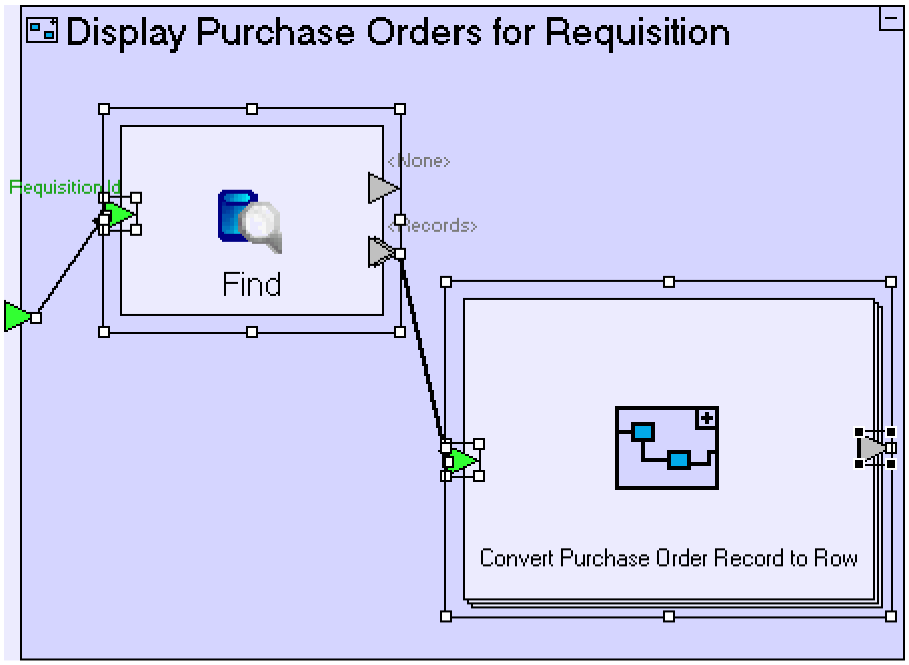
Add a **Database/Find**.

Add a trigger to it. Name it ***Requisition Id***. Create a flow linking the trigger of **Display Purchase Orders for Requisition** to it.

The purchase orders retrieved should be converted into rows in the **Purchase Order List** table:

Add a **Basic/Action**. Name it ***Convert Purchase Order Record to Row***. Mark it repetitive. Add to it a trigger and an exit. Create a flow linking **Find/<Records>** to the trigger.

The **Display Purchase Orders for Requisition** model should look similar to the following:


Display Purchase Orders for Requisition
RequisitionId
<None>
<Records>
Find
Convert Purchase Order Record to Row

Zoom into **Convert Purchase Order Record to Row**.

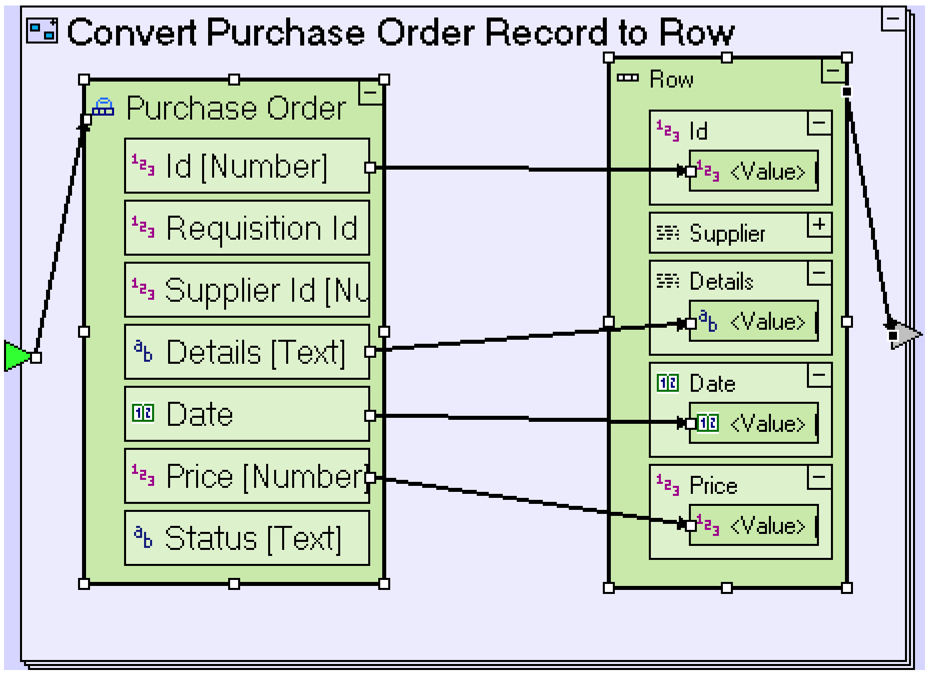
Drag a **Purchase Order** data structure from the repository/outline (it's available in **Order** button/**Create Purchase Order** popup/ **Submit Order** button). Add a flow linking the trigger of **Convert Purchase Order Record to Row** to the data structure.

Drag **Purchase Order List/Row** from the repository/outline (to create a display data element). Add a flow linking it to the exit of **Convert Purchase Order Record to Row**.

Create the following flows to map the fields of the purchase order database records to the fields of the rows in the display table:

Source	Target
Purchase Order/Id	**Row/Id/<Value>**
Purchase Order/Date	**Row/Date/<Value>**
Purchase Order/Details	**Row/Details/<Value>**
Purchase Order/Price	**Row/Price/<Value>**

The **Convert Purchase Order Record to Row** model should look similar to the following:



There's one additional field in the **Row** which we have not yet populated: The **Supplier** field. The **Purchase Order** data structure has a **Supplier Id** field which contains a numeric identifier of the supplier, which is not informative. Instead, we should display the supplier's company name from the **Supplier** table in the database:

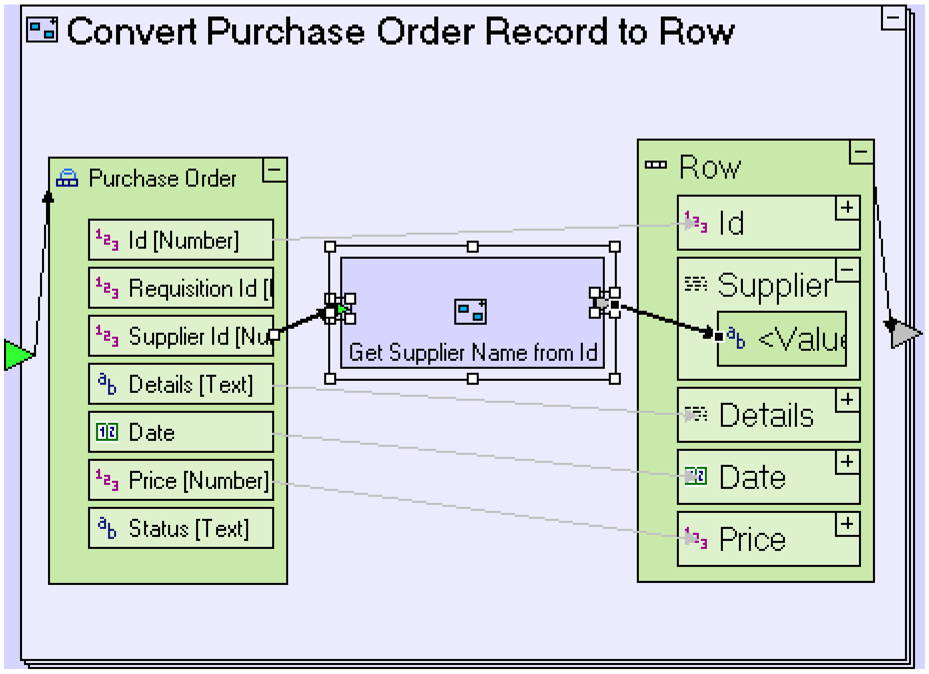
Zoom into **Convert Purchase Order Record to Row**.

Add a **Basic/Action**. Place it in between the **Purchase Order** database record and the **Row** display data element. Name it ***Get Supplier Name from Id***. Add to it a trigger and an exit.

Create a flow linking **Purchase Order/Supplier Id** to the trigger of **Get Supplier Name from Id**.

Create a flow linking the exit of **Get Supplier Name from Id** to **Row/Supplier/<Value>**.

The **Convert Purchase Order Record to Row** model should look similar to the following:



Zoom into **Get Supplier Name from Id**.

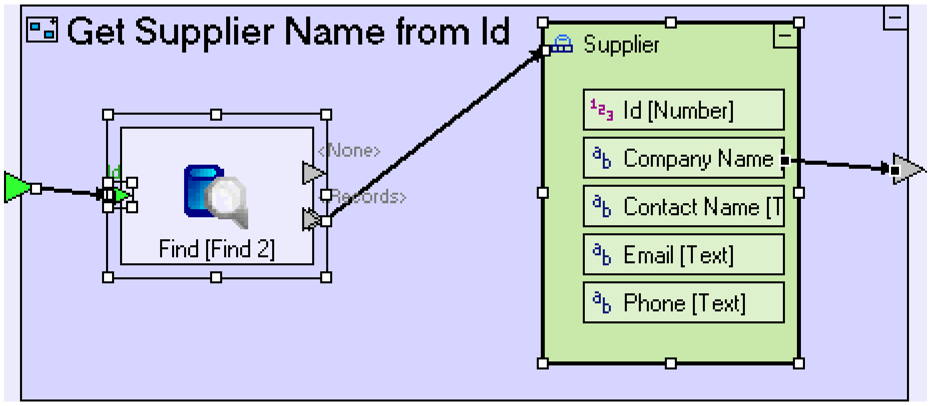
Add a **Database/Find**. Add to it a trigger and name it ***Id***. Create a flow linking the trigger of **Get Supplier Name from Id** to **Find/Id**.

Drag a **Supplier** database record (it's available in **Manage Suppliers** view/**Supplier List** table, among other locations) into **Get Supplier Name from Id**. Create a flow linking **Find/<Records>** to **Supplier**.

Note that although the **<Records>** exit is a repetitive exit, it is linked to a non-repetitive target data structure. This is not a problem because we're using **Id** as the retrieval criteria for the **Find** process, and since **Id** is unique, the process will never return more than one record.

Create a flow linking **Supplier/Company Name** to the exit of **Get Supplier Name from Id**.

The **Get Supplier Name from Id** model should look as follows:



You've probably noticed that the **Find** element we've just added**,** appear slightly different than usual, displaying 2 names: **Find [Find 2]**.
Find, is the **Element Name**. **Find 2**, appearing in square brackets, is the **Model Name**. As explained previously (See **Stage 2),** elements have both a **Model Name** and **Element Name** defined automatically – but since they are usually identical, they are usually displayed as one.
When adding an element to the model, the Tersus Studio will automatically modify either of the names in order to maintain uniqueness, by adding an index number, in cases where the name provided by the user is already in use.
In the current scenario, the **Find** element in **Get Supplier Name from Id** is stored (by default) in the **Approved Requisitions List** package (check this by right-clicking **Find** in the model editor and selecting **Show in Repository Explorer**). This package already contained a model called **Find** (the one we created in **Display Purchase Orders for Requisition** just beforehand), and so the newer **Find** must have a different **Model Name**.

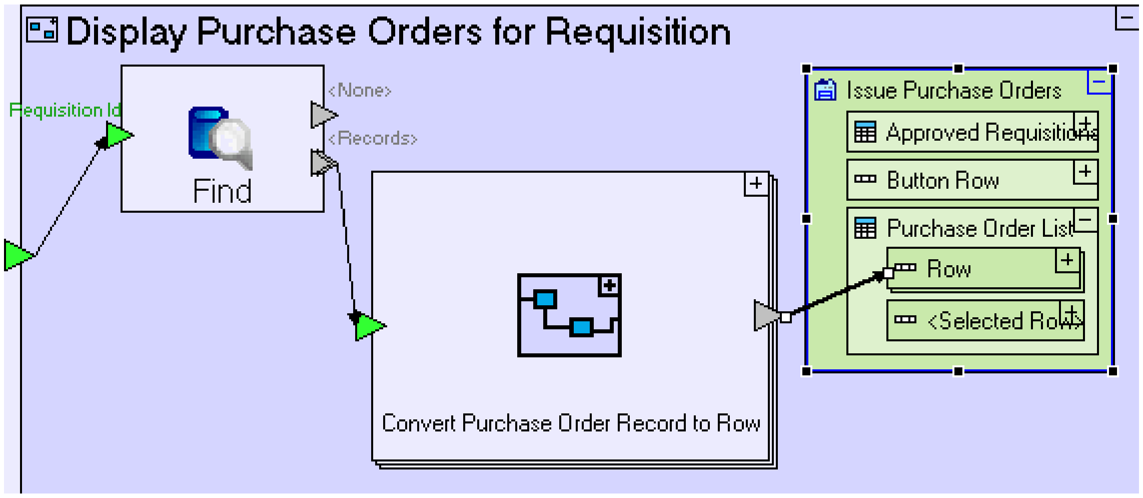
To wrap up the modeling of **Display Purchase Orders for Requisition**, we need to send the created rows to the display:

Zoom out to **Display Purchase Orders for Requisition**.

Add an ancestor reference to the **Issue Purchase Orders** view.

Create a flow linking the **Convert Purchase Order Record to Row** exit to **Issue Purchase Orders**/**Purchase Order List**/**Row**.

The **Display Purchase Orders for Requisition** model should look similar to the following:



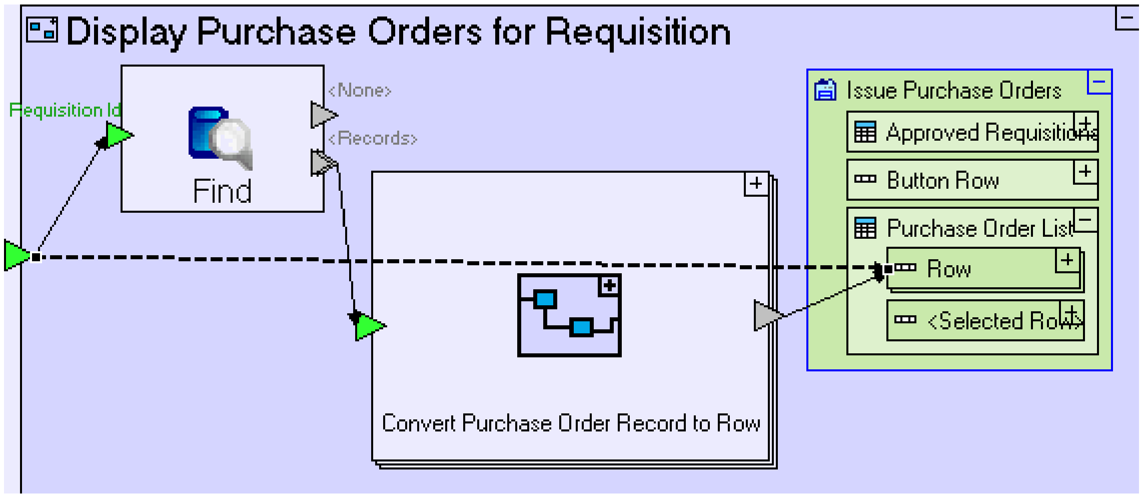
Note that we are now using a different method to populate a table with rows (compared to the various **Populate…** processes we modeled in previous stages): Instead of sending the rows to an intermediate **Purchase Order List** data element which is then sent to the display ancestor reference, the rows are sent directly to the display.

The model is simpler then previously modeled **Populate…** processes, but has one behavioral difference which needs to be addressed: Since the display is now updated at the row level and not at the table display level, the rows are accumulated by the repetitive row element, meaning that if rows were displayed in the table before the process was executed, the new rows created by the process will be added to the existing rows, instead of replacing them.

The solution is to make sure that the table is cleared, before the display is updated:

Create a **Remove** flow, linking the trigger of **Display Purchase Orders for Requisition** to **Issue Purchase Order/Purchase Order List/Row**.

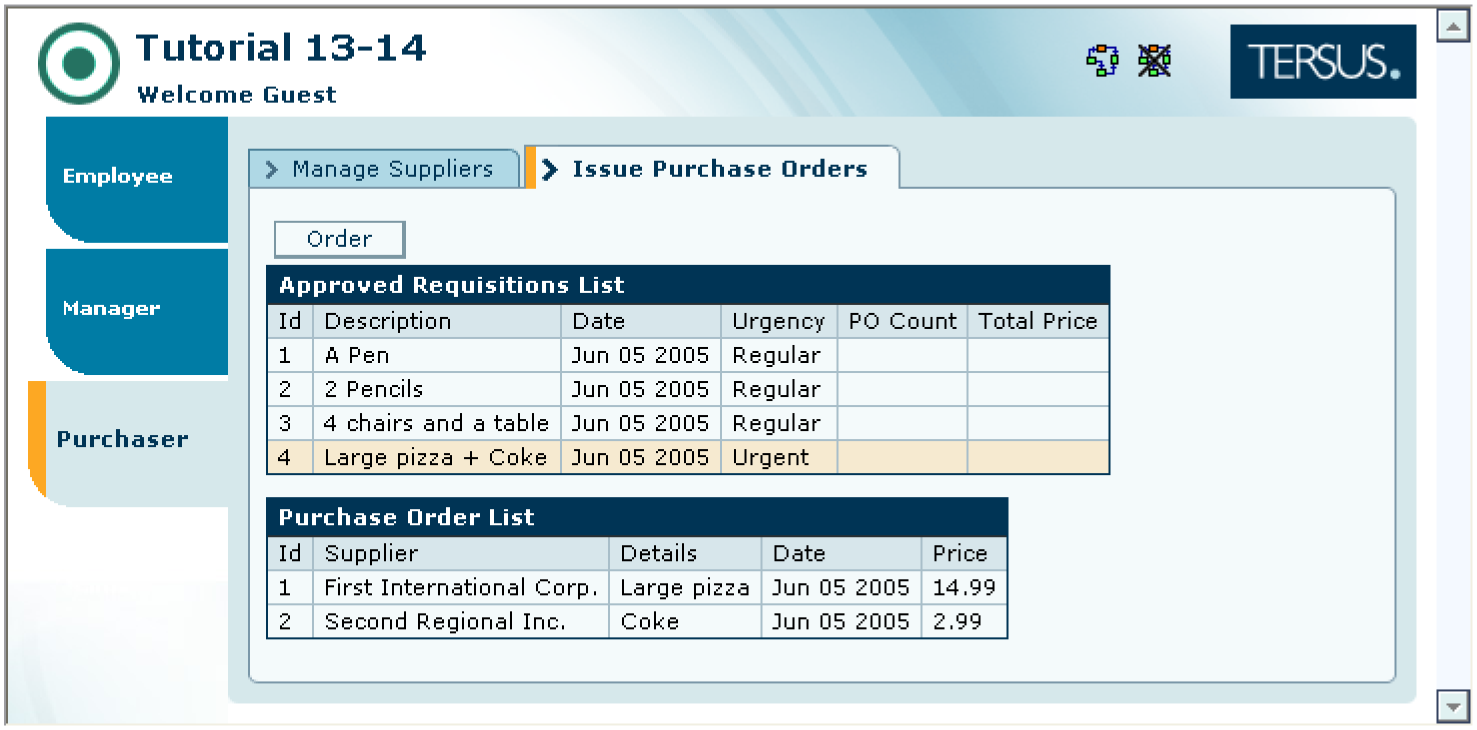
The resulting model should look similar to the following:



You might think that we could have set the source of the remove flow from the **Find/<None>** exit (as we did in the various **Generate ...** models previous modeled), but that would actually be the wrong thing to do, since it would only clear the table in those cases where **Find** did not locate any purchase orders, and so will not clear the table when records are found, causing duplicates to appear.
Setting the remove source as we did, ensures that clearing the table occurs regardless of **Find**'s result, and actually before **Find** is executed**.**

Save your work, and view the application in the browser.

If you click on a requisition with issued purchase orders, they should appear in the lower table, in a similar fashion to the following:



Approved Requisitions List

Id	Description	Date	Urgency	PO Count	Total Price
1	A Pen	Jun 05 2005	Regular		
2	2 Pencils	Jun 05 2005	Regular		
3	4 chairs and a table	Jun 05 2005	Regular		
4	Large pizza + Coke	Jun 05 2005	Urgent		

Purchase Order List

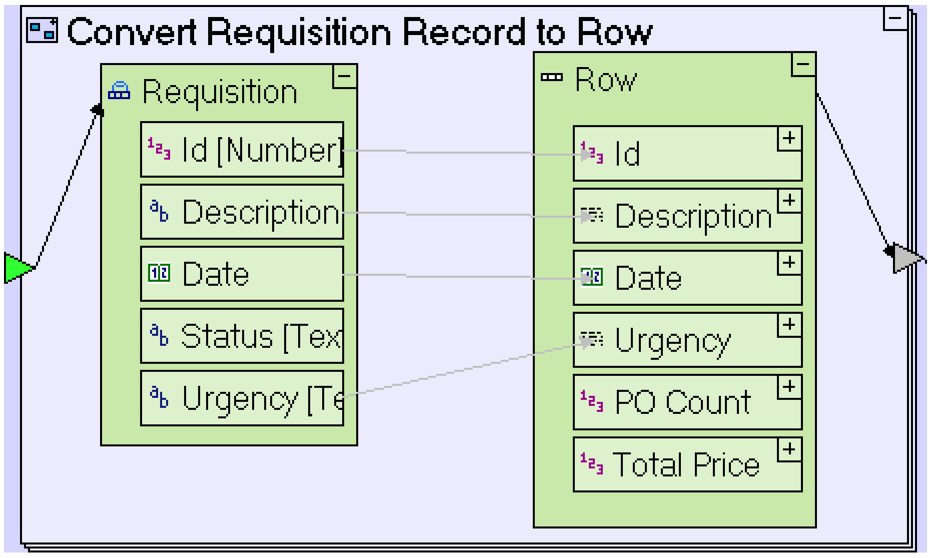
Id	Supplier	Details	Date	Price
1	First International Corp.	Large pizza	Jun 05 2005	14.99
2	Second Regional Inc.	Coke	Jun 05 2005	2.99

Add a Process (to calculate PO aggregates for each requisition)

Approved Requisitions List has 2 columns, **PO Count** and **Total Price**, which are currently empty. We shall now calculate these aggregate values for each requisition. These aggregates will be calculated on the fly for all requisitions in the requisition list:

> Zoom to **Issue Purchase Orders/Populate Approved Requisitions List/Generate Approved Requisitions List/Convert Requisition Record to Row**.

The **Convert Requisition Record to Row** model currently looks similar to the following:



We’ll add to it a sub-process which calculates the aggregates for a given requisition:

Add a **Basic/Action** template. Name it ***Calculate PO Aggregates for Requisition***.

Add a trigger to the process. Name it ***Requisition Id***. Create a flow linking **Requisition/Id** to it.

Add an exit to the process. Name it ***PO Count***. Create a flow linking it to **Requisition Row/PO Count/<Value>**.

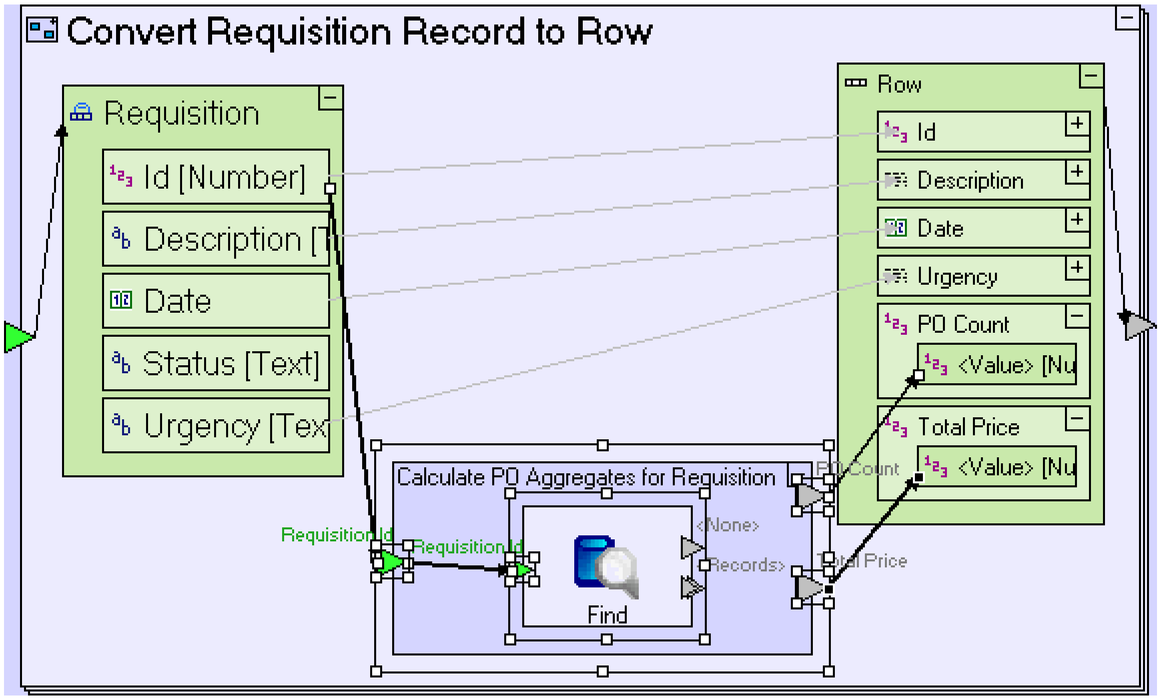
Add a second exit to the process. Name it ***Total Price***. Create a flow linking it to **Requisition Row/Total Price/<Value>**.

The process should retrieve all purchase orders for the given requisition:

Zoom into **Calculate PO Aggregates for Requisition**.

Add a **Database/Find** template. Add a trigger, named ***Requisition Id***. Create a flow linking the **Calculate PO Aggregates for Requisition/Requisition Id** trigger to it.

The **Convert Requisition Record to Row** model should now look similar to the following:



We are actually interested in 2 separate results based on the **Find** process:

2. Counting the number of returned **Purchase Order** records – this is the **PO Count**.
3. Summing the **Price** of each returned **Purchase Order** record – this is **Total Price**.

Use a Count Template (to count the number of records found by Find)

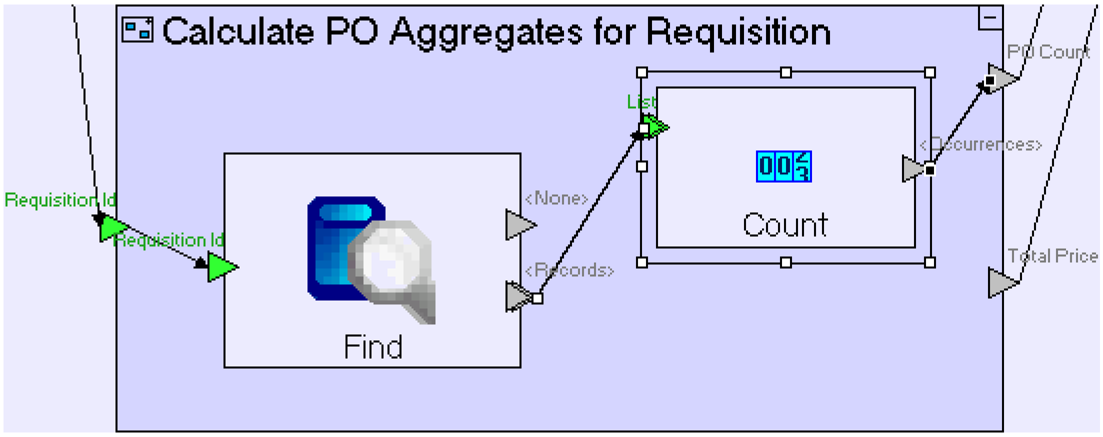
To count the number of records returned by **Find**, use the **Count** template, which counts the number of objects sent to its trigger (or triggers):

Add a **Collections/Count** (). Delete the unnamed non-repetitive trigger, leaving the repetitive **List** trigger intact.

Create a flow linking **Find/<Records>** to **Count/List**.

Create a flow linking **Count/<Occurrences>** to the **Calculate PO Aggregates for Requisition/PO Count** exit.

The **Calculate PO Aggregates for Requisition** model should now look similar to the following:



Count does just that; it counts all objects sent to it, and when no more objects are sent, it outputs the object count.
Note that **Collections/Count** has a similar icon to **Database/Sequence Number** - make sure you do not mix them up.

Use a Sum Template (to calculate the total price of a requisition)

To calculate the total price of a requisition, we first need to extract the price for each purchase order:

Drag a **Purchase Order** data structure from the repository/outline. Set it repetitive.
Create a flow linking **Find/<Records>** to it.

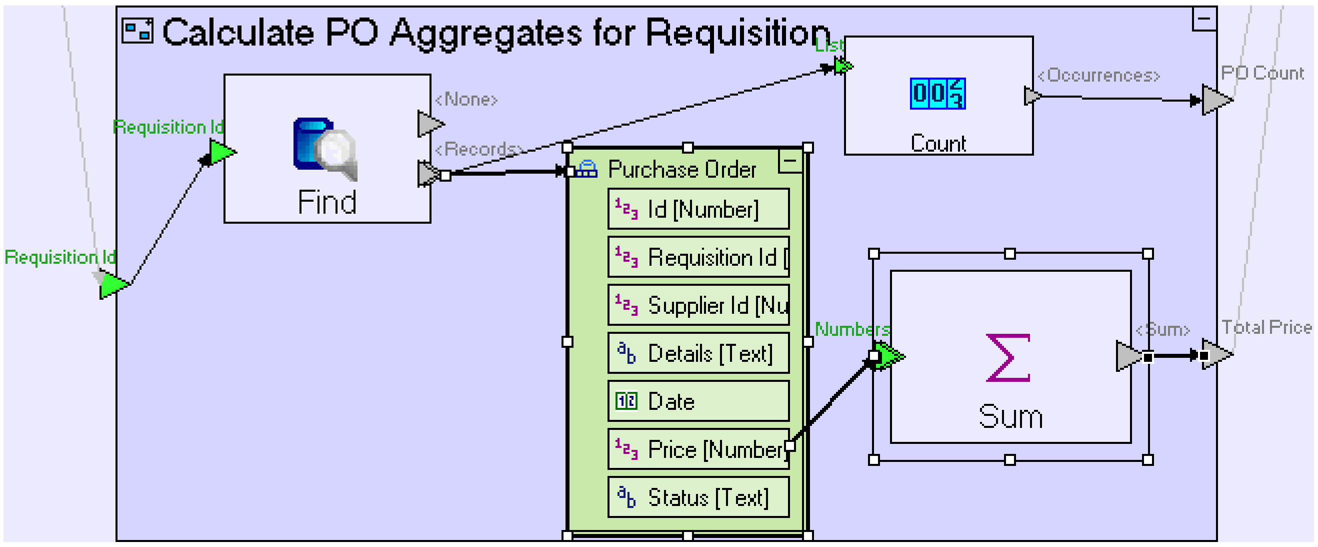
We shall use the **Sum** template to calculate the total price for the requisition.

Add a **Math/Sum** (Σ).

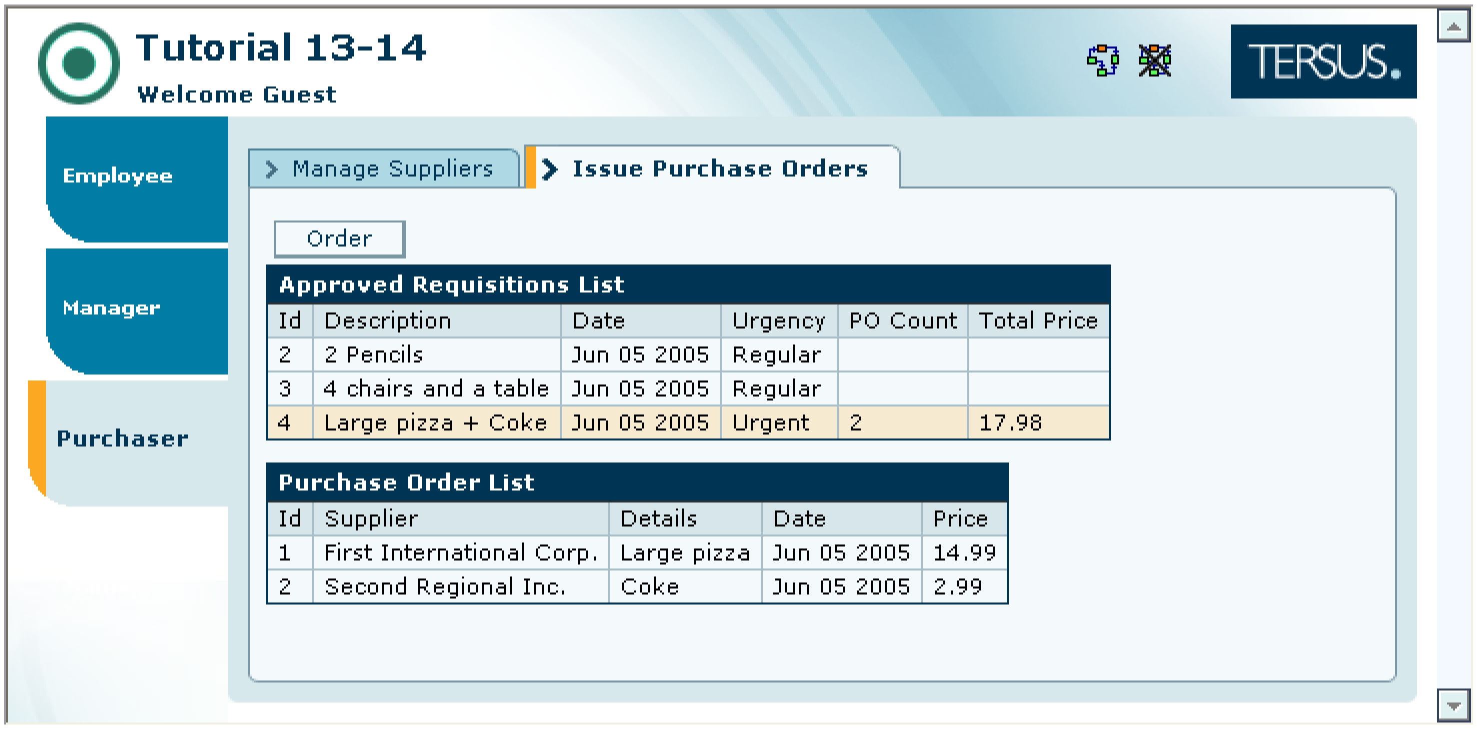
Create a flow linking **Purchase Order/Price** to **Sum/Numbers**.

Create a flow linking **Sum/<Sum>** to **Calculate PO Aggregates for Requisition/Total Price**.

The **Calculate PO Aggregates for Requisition** model should look similar to the following:



Save your work, and view the application in the browser, which should display aggregates for requisitions with issued orders, similar to the following:



Approved Requisitions List

Id	Description	Date	Urgency	PO Count	Total Price
2	2 Pencils	Jun 05 2005	Regular		
3	4 chairs and a table	Jun 05 2005	Regular		
4	Large pizza + Coke	Jun 05 2005	Urgent	2	17.98

Purchase Order List

Id	Supplier	Details	Date	Price
1	First International Corp.	Large pizza	Jun 05 2005	14.99
2	Second Regional Inc.	Coke	Jun 05 2005	2.99

Use a Refresh Template (to refresh data in your display)

When the **Issue Purchase Orders** view is opened, and the **Approved Requisitions List** table is populated, the aggregates are shown correctly. However, when a new order is issued, the display (including the aggregates) is not updated.

In previous stages we've reused the **Populate...** process to update the table, and although we could do the same here, that will not be sufficient, since it does not clear the table (although that could be fixed). There's also the issue of the **Purchase Order List** table, which should be cleared since no requisition is selected.

An alternative strategy would be to force the view to completely refresh, as follows:

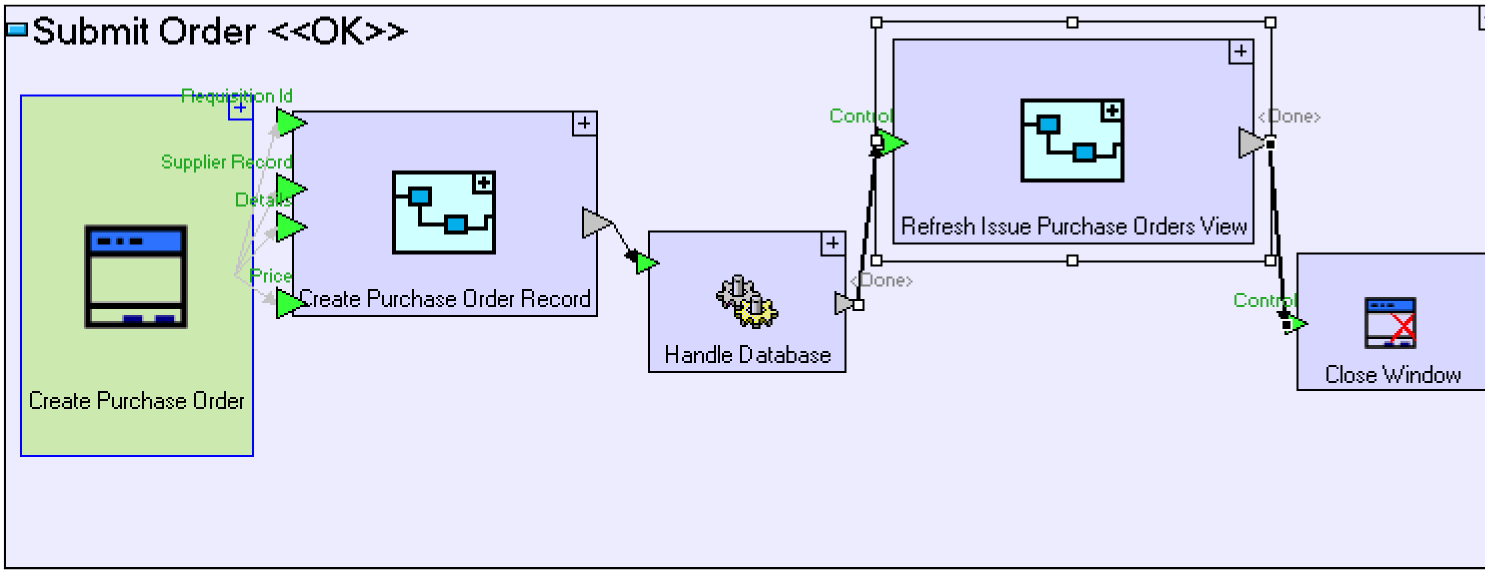
Zoom to **Submit Order** button.

Add a **Basic/Action**. Name it **Refresh Issue Purchase Orders View**. Add a **Control** trigger and a **<Done>** exit to it (using the **Add Element** context sub-menu).

Select the flow linking the **Handle Database/<Done>** exit to the **Close Window/Control** trigger, and drag the target to the **Refresh Issue Purchase Orders View/Control** trigger.

Create flow linking the **Refresh Issue Purchase Orders View/<Done>** exit to the **Close Window/Control** trigger.

The **Submit Order** button model should look similar to the following:



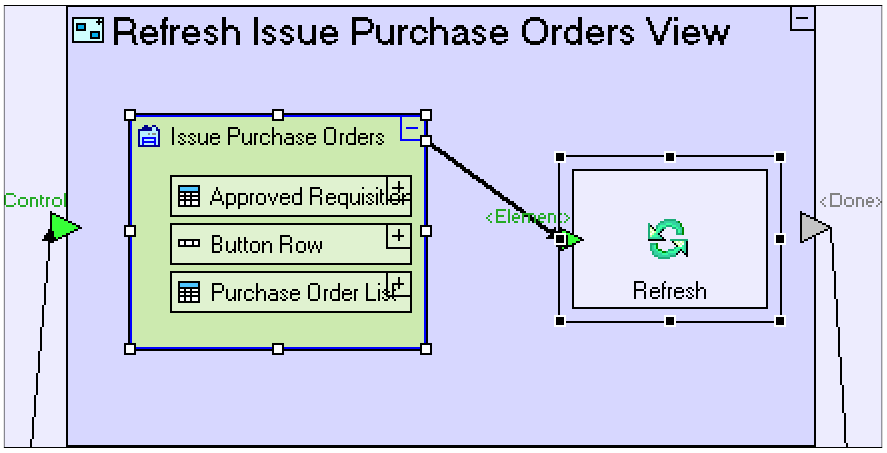
Zoom into **Refresh Issue Purchase Orders View**.

Add an ancestor reference to the **Issue Purchase Orders** view.

Add a **Display Actions/Refresh** ( ).

Create flow linking **Issue Purchase Orders** to **Refresh/<Element>**

The **Refresh Issue Purchase Orders View** model should look similar to the following:



The **Refresh** action reloads the display element passed to it, clearing the display element in question and executing any processes it contains – this is identical to the behavior when the element is loaded in the first place.

Save your work, and view the application in the browser.

Create a new order and note that the aggregates for the requisition in question are updated

Completing Stage 14

Import the sample project **Requisition Management System**.

If you installed the **Tersus Studio** using the installer, there is no need to import the project and it should be available in your workspace.
If you do need to import the project, see the **Importing a Sample Project** section at the end of **Stage 2**.

This project contains all the functionality modeled thus far plus additional functionality required to complete the application.

Take a look at the added **Manage Purchase Orders** view model, which implements the following functionality:

3. A list of issued purchase orders
4. An **Order Received** button

The **Order Received** button does the following:

4. Updates the status of the selected purchase order to **Received**.
5. If the requisition to which the purchase order belongs has additional purchase orders not yet received, then it changes the status of the requisition to **Partly Fulfilled**. Otherwise, the status of the requisition is changed to **Fulfilled**.
6. Refreshes the display of purchase orders.

See It Live

Object 13

Click here to open the live project in a separate window.

Appendix A – Tersus Studio Features and Tools

Appendix Goals

This appendix provides a general introduction to the **Tersus Studio**, and the different tools provided to support modeling, such as the **Model Editor**, **Outline**, **Repository Explorer**, **Template Library**, and embedded application and database servers.

The Eclipse Platform

The Tersus Studio uses the **Eclipse** platform, which is an industry standard IDE framework, providing various features and significant flexibility, through the menus and toolbar, including the possibility to rearrange the display to suit your taste.

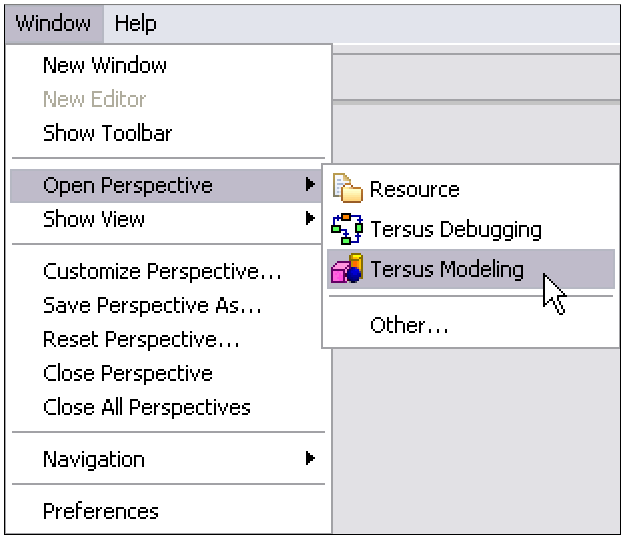
The **Eclipse** platform uses the notion of a **Perspective** displaying one or more **Views**, arranged in a specific way. Switching between perspectives changes the make-up and arrangement of views. The screenshot above displays the **Tersus Modeling** perspective which, by default, includes the following views: **Model Editor, Palette**, **Outline**, **Repository Explorer**, **Properties** and **Validation**.

- To switch to a different perspective, use **Window -> Open Perspective**
- To change the arrangement of views in a perspective, click on a specific view's title bar, drag it around and drop it in its new position.

For further information regarding the features provided by Eclipse, see the Eclipse platform help system (accessible through **Help -> Help Contents**).

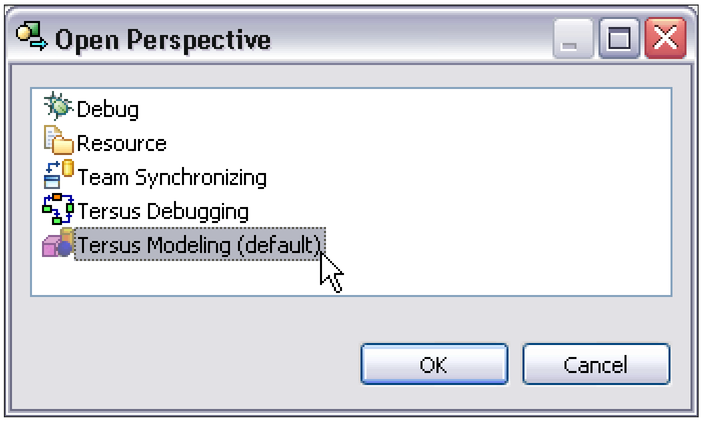
Before we continue, make sure the **Tersus Modeling** perspective is displayed. To find out which perspective is currently in use, take a look at the Eclipse window title bar. If it does not start with "**Tersus Modeling …**", then another perspective is currently displayed. If this is the case:

Select **Window -> Open Perspective -> Tersus Modeling**



Or, if the **Tersus Modeling** option does not appear:

Select **Window -> Open Perspective -> Other…**



Select **Tersus Modeling** from the list, and click **OK**.

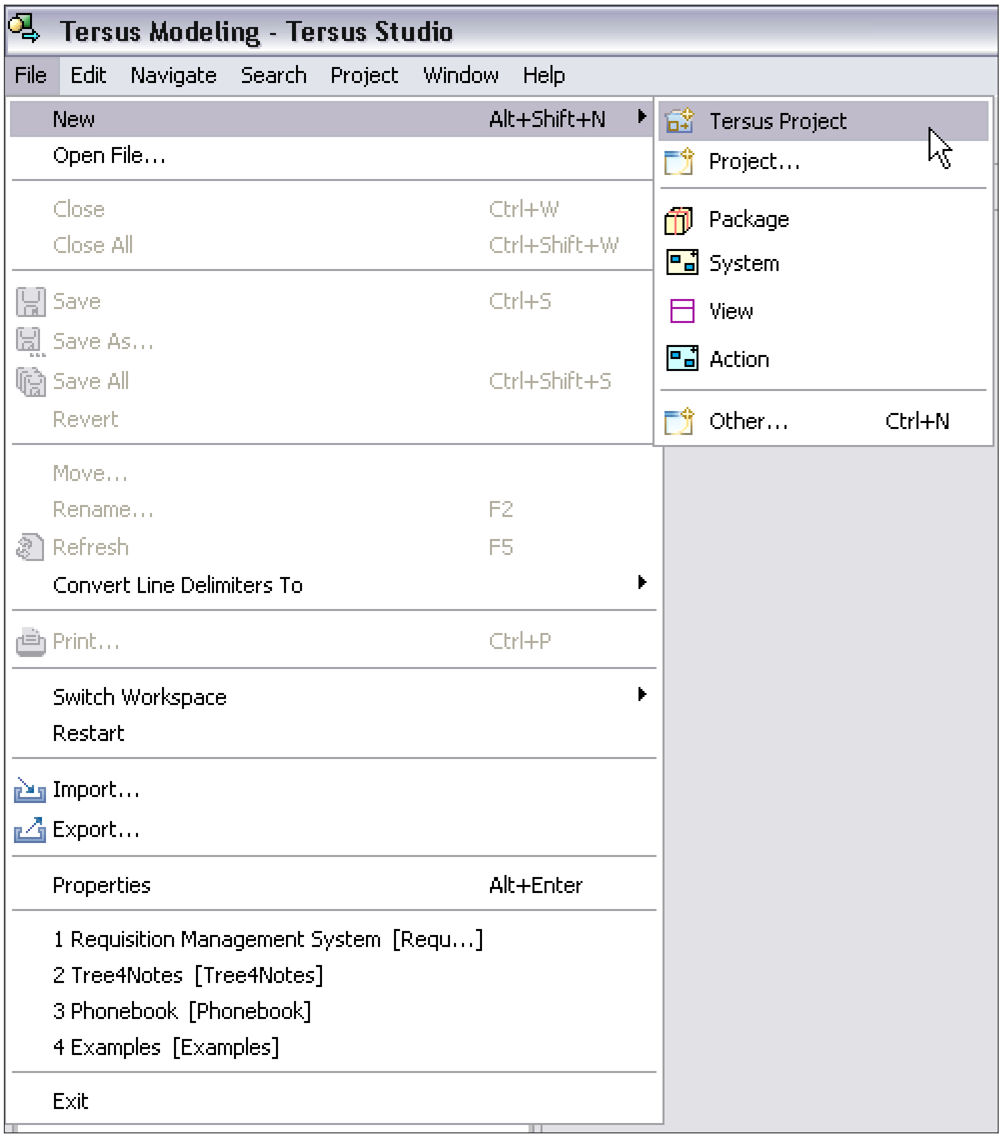
Eclipse should switch to the **Tersus Modeling** perspective, displaying the perspective name on Eclipse window title bar.

Creating a New Project

To start a new Tersus project do the following:

Select **File -> New**

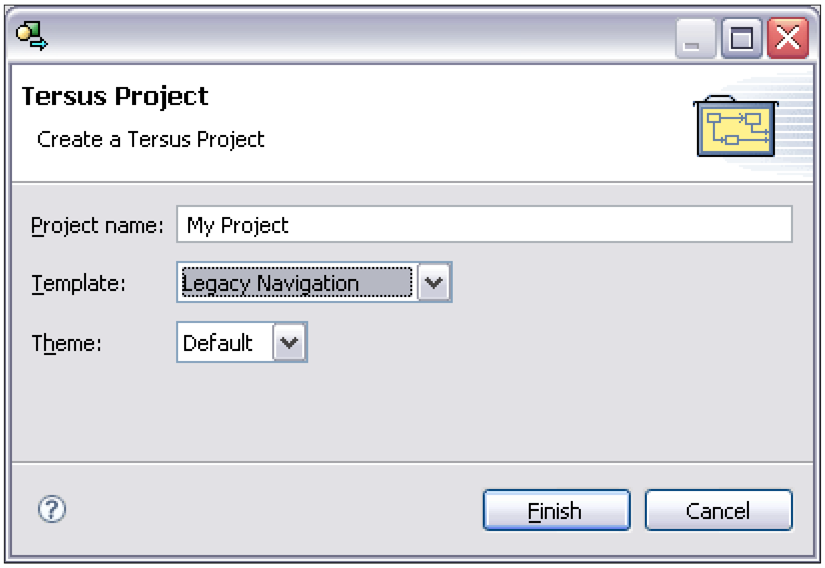
The following submenu will appear:



Select **Tersus Project**.

Note that there are two Project options in the menu – make sure you chose the first one (**Tersus Project**) rather than the second (**Project** ...).

The following dialog box will appear:



Tersus Project

Create a Tersus Project

Project name: My Project

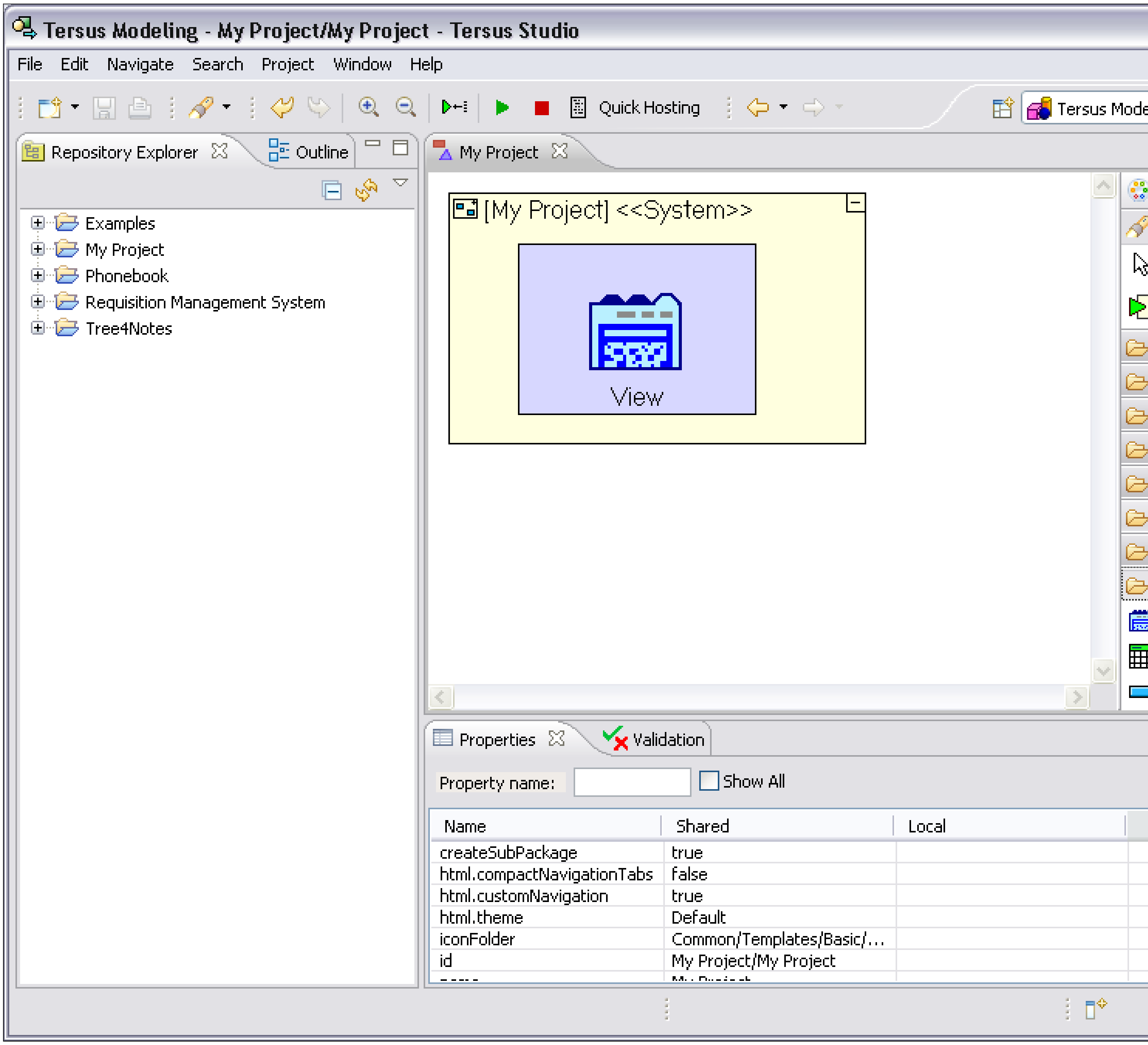
Template: Legacy Navigation

Theme: Default

Enter a **Project name** for your new project: ***My Project***.

Press the **Finish** button to create your first Tersus project.

You should see the following:



Name	Shared	Local
createSubPackage	true	
html.compactNavigationTabs	false	
html.customNavigation	true	
html.theme	Default	
iconFolder	Common/Templates/Basic/...	
id	My Project/My Project	

You can use the **My Project** project you have just created to test the different platform features discussed in this stage.

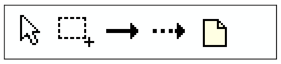
Familiarizing Yourself with the Model Editor

The Palette

On the right side of the **Model Editor** you can see the **Palette** (Palette ▷).

The palette contains four types of elements:

8. **Editing Tools**



Used to define and control the layout of elements in the editor.

(**Select**) Selects and moves elements in the editor.

(**Marquee**) Selects all elements in the specified region of the editor.

(**Flow**) Creates a (regular) flow between elements in the editor.

(**Remove**) Creates a remove flow between elements in the editor.

(**Note**) Adds a note (documentation/comment) to an existing element in the editor

4. **Slots**

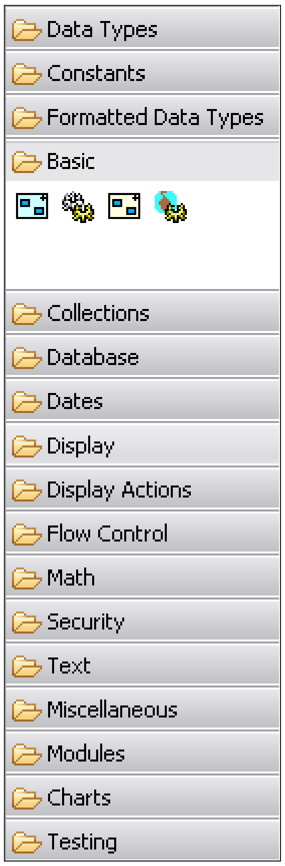
Used to define input/output "ports" of processes.

(**Trigger**) A port used to activate and pass input into a process

(**Exit**) A port used to pass output from a process

(**Error Exit**) A port used to pass errors (exception) from a process

5. **Template Library**



Templates are predefined elements which serve as building blocks for modeling. There are several categories of templates (such as Data Types, Database, Display etc.), and each category contains several templates.

7. **Search** button ( )

   Provides the option to search for a specific template in the palette by name (or part of it).

Inserting a New Element to the Model

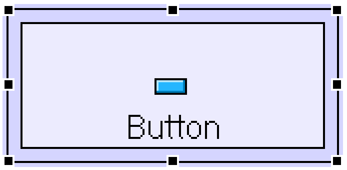
When you select a template from the Palette and return to the editor, the mouse pointer changes to signify that a new element is going to be inserted into the model. There are 2 methods of insertion from the palette:

5. Click – Creates a new element with default size (hinted by a dashed rectangle).
6. Click & drag – Creates a new element with the size specified by the user.

New elements may also be inserted from the Repository and the Outline, using drag-and-drop. They will be created with a default size.

Selecting an Element

When you click on an element displayed in the editor, it becomes selected, marked with a frame, as in the following screenshot:



Selecting multiple elements is possible using the following methods:

6. After selecting the first element, hold down the [CTRL] key and continue to select additional elements.
7. Use the **Marquee** tool ( ) in the palette to specify a selection area in the editor. All elements in the marked area will be selected.

Moving an element

When an element has been selected, drag it around to change its position in the editor.

Resizing an element

After an element has been selected it can be resized by dragging any of the 8 selection anchors which appear around the selection frame.

Certain types of elements (specifically Display, Process and System elements) resize differently when the corner anchors are used. These elements may contain other elements, therefore when the corner anchors are used to resize, the aspect ratio of the element and its sub-elements (width-to-height ratio) is maintained.

Drill-down

Models are hierarchical, meaning that an element may contain other elements (making up sub-models), and each of these sub-elements can in-turn contain additional sub-models, and so on.

The model editor provides drill-down functionality which lets us view the different parts of the model at different levels of detail.

There are 2 methods to drill-down (or up):

7. **Expand/Collapse** – elements in the model which contain other elements, display a small ⊞ (expand button) or ⊟ (collapse button). Clicking on the expand/collapse button will cause the model editor to display the contents of the element (expand), or hide them (collapse).
8. **Double-clicking** an element in the model expands it to display its contents, and in addition causes the editor to center the view on the model and zoom in or out so the model fits in the view.

Zoom-in/out

There are various methods for zooming in and out:

8. **Double-clicking** an element in the **model editor** will cause the editor to zoom in/out and center on the element (in addition to expand if applicable).
9. **Double-clicking** an element in the **outline** is similar to double clicking in the editor. This method is useful when we want to move directly to part of the model which is not in the current scope of the display.
10. The **toolbar** provides buttons for zoom-in (🔍), and out (🔍).

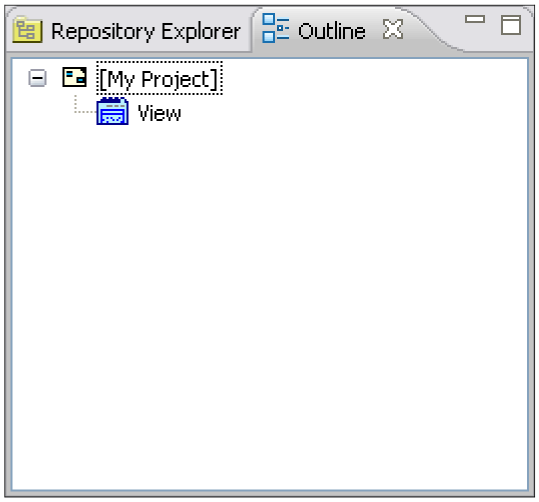
Zoom works slightly differently for data-elements. Double-clicking will only function when performed on the top data element, and not on any of its descendant data elements.

Undo/Redo

The model editor provides full **Undo/Redo** functionality. This allows you to try out different modeling strategies, making quick changes to the model, without the fear of losing working functionality. Undo/Redo is available throughout an editing session, as long as the model editor window is open (saving and running the application does not prevent a later **Undo**)..

Undo/Redo functionality is available through the **Edit** menu as well as through **[CTRL-Z]/ [CTRL-Y]** keyboard shortcuts.

The Outline



The Outline pane, which appears to the left of the model editor, provides a different view of the application model, as a hierarchical tree view of the elements making up the complete model. The outline is always synchronized with and reflects the hierarchical nature of the model itself (elements containing other elements, and so on).

Synchronization with the Model Editor

The selected item in the Outline is always kept in sync with the selected element in the model editor. Selecting in one will always select the matching item in the other.

Double-Click Behavior

Double-clicking a model in the outline will locate and zoom to it in the current model editor.

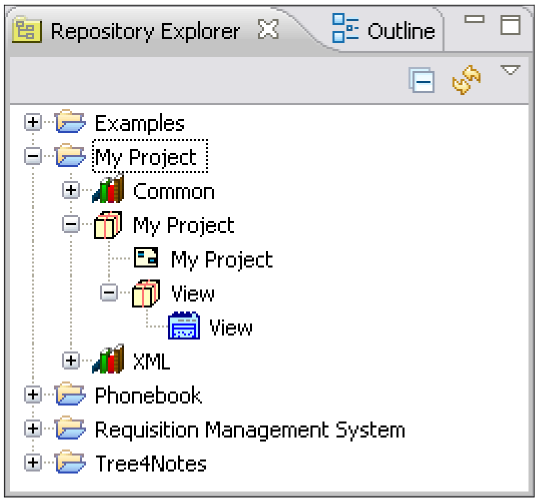
This is useful when trying to zoom to a specific element (especially in complex model hierarchies).

Drag-and-Drop Behavior

Dragging a model from the outline and dropping it into the editor will reuse the model, which means that under certain restrictions (which will be covered in later stages of the tutorial), functionality which has already been modeled can be used again, instead of having to remodel it from scratch.

In the case of reuse, the same element appears multiple times (at different locations) in the outline tree. There are however situations where separate, unrelated models share the same name; this is possible when they are in different packages (see the section regarding **Repository Explorer** for further details).

The Repository Explorer



The Repository Explorer, which appears to the left of the model editor, provides a complete list of all the models making up an application.

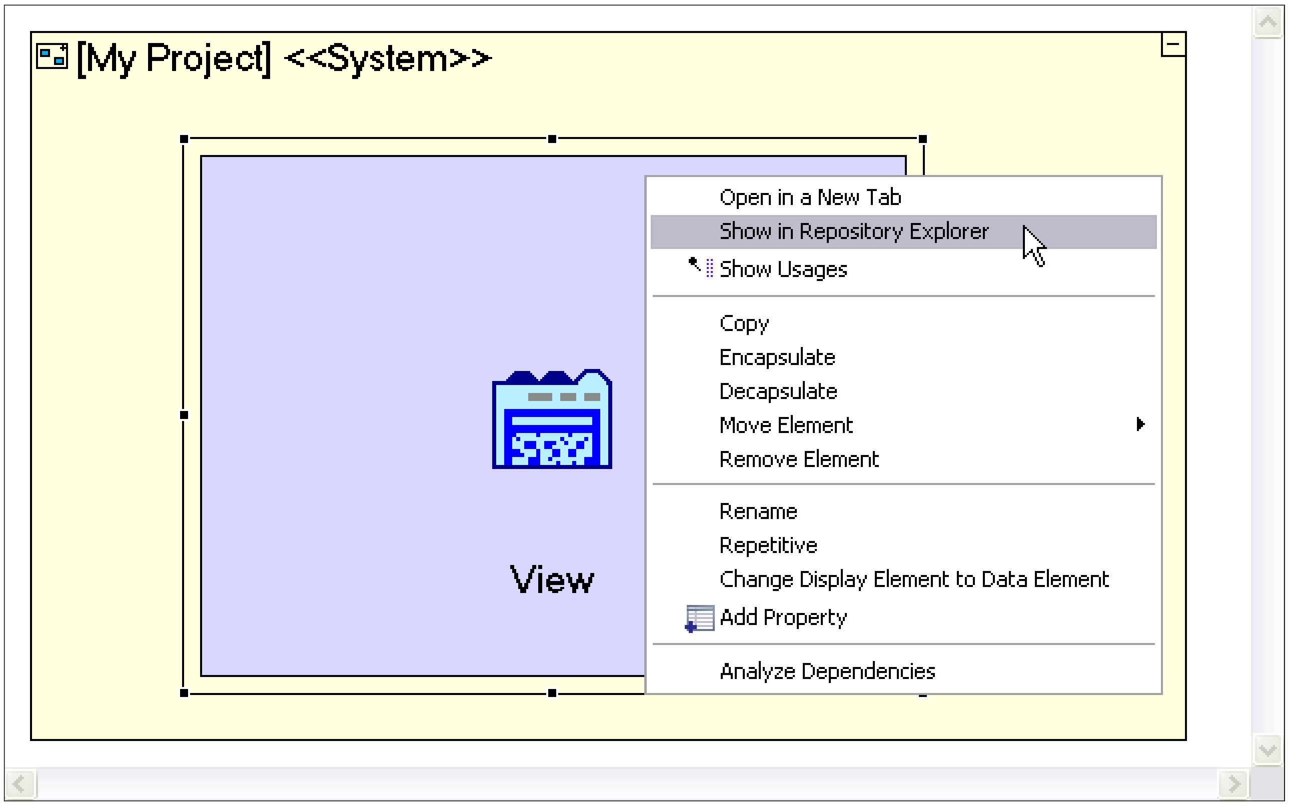
Finding your way in the Repository

The repository is organized hierarchically into **Packages** (and **sub-packages**), which group the models into functional categories. Packages are created automatically when certain templates (**System**, **View**, **Button**) are used, but you can always create additional packages and move models from one package to another to organize them as you see fit. When modeling, new elements are automatically created in the same package as the parent model to which they are added.

In order to locate, in the repository, the model you are currently editing, you can use the following shortcut:

Right-click the element in the editor.

Select the **Show in Repository Explorer** option from the context menu.



Double-click Behavior

Double-clicking a model in the repository will open it in a new model editor window. When working on complex model hierarchies, you may use this to open and edit a specific sub-model.

Drag-and-drop Behavior

Dragging a model from the repository and dropping it into the editor will reuse the model (similar to dragging from the Outline). The target model editor must be part of the same project as the source model.

Drag-and-drop inside the repository is also possible, resulting in the model being moved from one package to another.

The Repository Explorer vs. the Outline

Although both the repository and the outline display a project's content in a hierarchical manner, there are quite a few differences between them, mainly:

Repository Explorer	**Outline**
The hierarchy displays all the models in all the projects.	The hierarchy maps to and is in sync with the current model editor. Only the elements

	which make up the current model are displayed.
Reused models appear only once	Reused models appear as many times as they are used in the current model.
Unused models appear in the repository and may be used again	Unused models do not appear in the outline, since they are not part of the current model.
The name displayed for each model is the **Model Name**.	The name displayed for each model is the **Element Name**.

The Embedded Application and Database Servers

Tersus includes a bundled, lightweight Application & Database Server, which can be used to view and test the modeled application at each stage of the modeling process.

The embedded servers are controled through the studio's toolbar:

Launch the application, in the application server and open the browser.

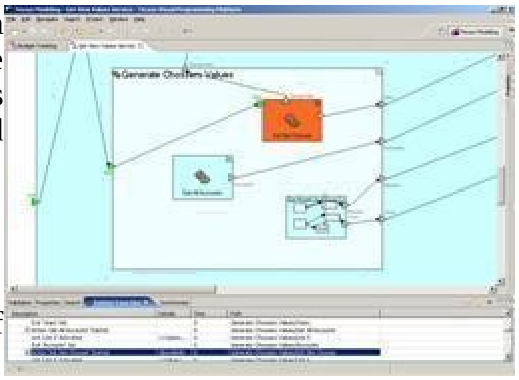
Stop the application, from running in the application server.

Show the application log file in a text editor window.

Try to view your application, as follows:

Click the button to start the application in the server, and open it in a browser.
You can switch back to the studio, change your modeling, and after saving your changes, if you switch back to the browser it will refresh automatically to apply your latest changes to the application model and database structures.

Appendix B – Visual Debugging



The Tersus Visual Programming Platform allows you to trace the execution of the application visually with the same diagrams used to define it, making it easier to understand the business logic and detect bugs.

When working in trace mode, the Tersus Server records every step during the application's execution, which can then be played back to view the flow of the application and the value of each data element.

Unlike regular debuggers, which allow you to only move forward, the Tersus tracing function acts like a "time machine," allowing you to move back and forth along the execution. If you reach a point where something looks erroneous, you can trace back to locate the root cause of the improper behavior.

On line documentation of this feature is available here.

Created on 02/13/11-17:24