CREATING LOCAL TRANSACTIONS

SAMPLE: SHOPPING CART

Last Update: June 2010

Table of Contents

1 Goal

In this tutorial we will continue walking through the development of a very simple application using Integranova. The tutorial covers the basic tools and steps involved in the development of applications from scratch as a way of introducing the reader to Integranova.

Following with the tutorial series, we will now expand the Tutorial 04.

Let's go to add new functionality for managing the purchase order in an easier way.

By the end of this tutorial, you will have developed a system in which it will be possible to delete a purchase order and all its lines, change the payment type when the purchase order is going to be paid or create a main category.

2 Defining complex functionality

In this tutorial we require more functionality for purchase orders. We will need to delete a purchase order and all its lines in only one step, to establish mechanisms to control the payment of the purchase order and to create a main category. The way to achieve that in Integranova is through the definition of *Local Transactions*.

As we saw in Tutorial 1 of the series, we can have three kinds of services in our classes: Events, Transactions and Operations. Events represent an execution unit. To group several execution units we will need another service type call *Local Transactions*.

Local transactions are defined in a specific class and allow defining complex functionality in a specific formula including events and other transactions in a specific order. At runtime, transactions are executed as **only one processing unit**. If any error occurs, for instance, a precondition is not fulfilled, all the changes that have been made during the execution of the transaction are undone (a roll back is performed) and the affected data remains with their initial values.

Transactions are very similar to events. They are defined in the same tab: *Services.* They have a *Name* (always in capital letters), an *Alias*, a *Help Message*, *Comments* and a ist of Arguments. But, instead of having valuations (as events), the way in which they can define its functionality is using *Transaction formula* (in the *Transaction* tab)*.* The *Transaction formula* describes what the transaction must do.

Operations are very similar to *transactions*. The way in which they are defined is the same, but operations are not executed in an all-or-nothing way. That means that when an error occurs during the execution, instead of undoing the changes, the operation follows with the execution of next part of the formula. When the execution is finished, no feedback is given back about the correct or incorrect execution of the operation. In this tutorial, operations will not be defined.

2.1 Creating a Category

If you see the 'create_instance' event of the 'Category' class, you can see that it requires two arguments:

- ✓ 'p_agrSuperCategory', where the super category of the new category should be introduced (it allows NULL value).
- ✓ 'p_atrCategoryName', where the name of the new category is indicated.

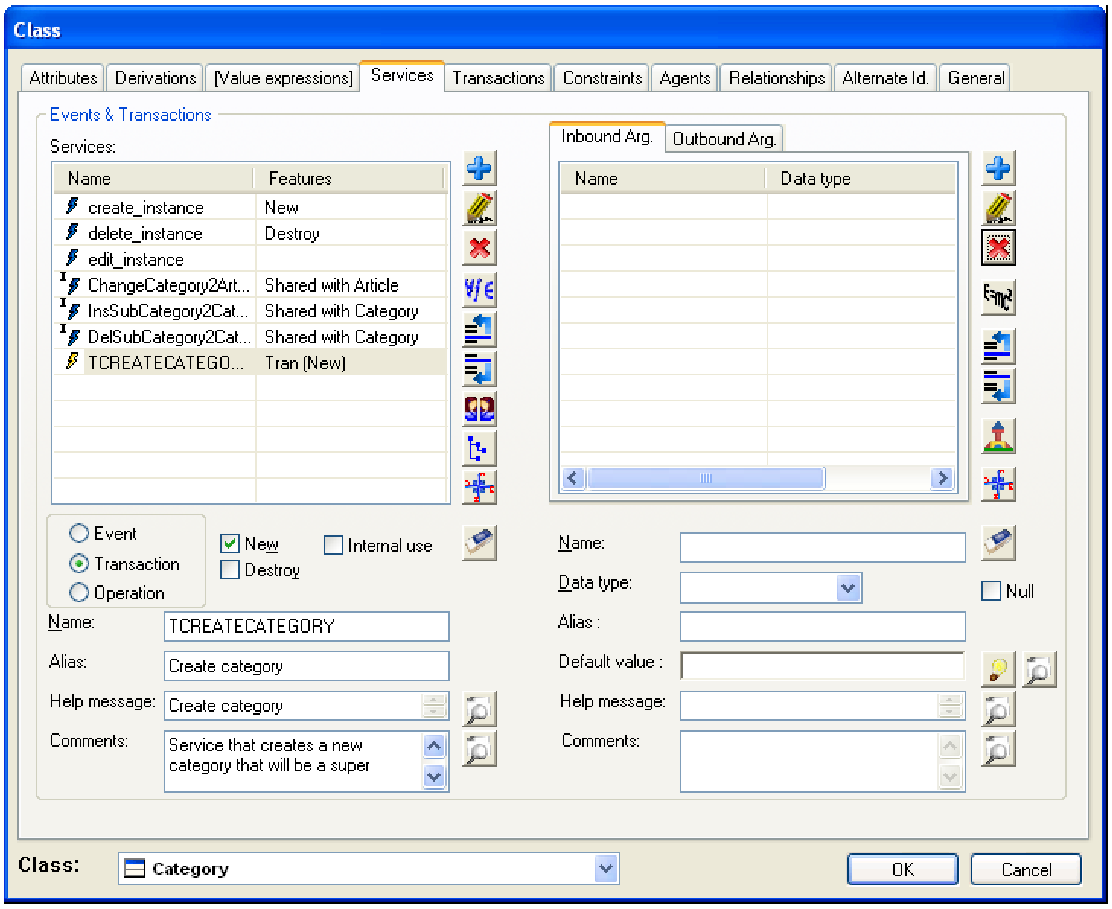
We know that when we want to create a category it will not have a super category but when we want to create a sub category it will have a category compulsory. We can add some preconditions to ensure that data introduced by the user of the application fulfils with that,

We have already defined an *Integrity Constraint* in the 'Category' class to restrict subcategory just one level.

But, if we want to avoid possible user errors, then the best option is to create two different services, one to create a main category and another one to create a sub category.

This first service to create a main category will be a transaction for 'Category' class, named 'TCREATECATEGORY'. When we are defining this new service, we have to check the *New* checkbox because this new service create a category, so is a creation service.

Let's see the data in the following figure:



Class

Attributes | Derivations | [Value expressions] | Services | Transactions | Constraints | Agents | Relationships | Alternate Id. | General

Events & Transactions

Services:

Name	Features
create_instance	New
delete_instance	Destroy
edit_instance	
ChangeCategory2Art...	Shared with Article
InsSubCategory2Cat...	Shared with Category
DelSubCategory2Cat...	Shared with Category
TCREATECATEGO...	Tran (New)

Inbound Arg. | Outbound Arg.

Name	Data type

( ) Event [✓ Transaction] ( ) Operation

[✓ New] [ ] Internal use [ ] Destroy

Name: TCREATECATEGORY
Alias: Create category
Help message: Create category
Comments: Service that creates a new category that will be a super

Name:
Data type:
[ ] Null
Alias :
Default value :
Help message:
Comments:

Class: Category

OK | Cancel

Figure 1 Creation transaction of Category

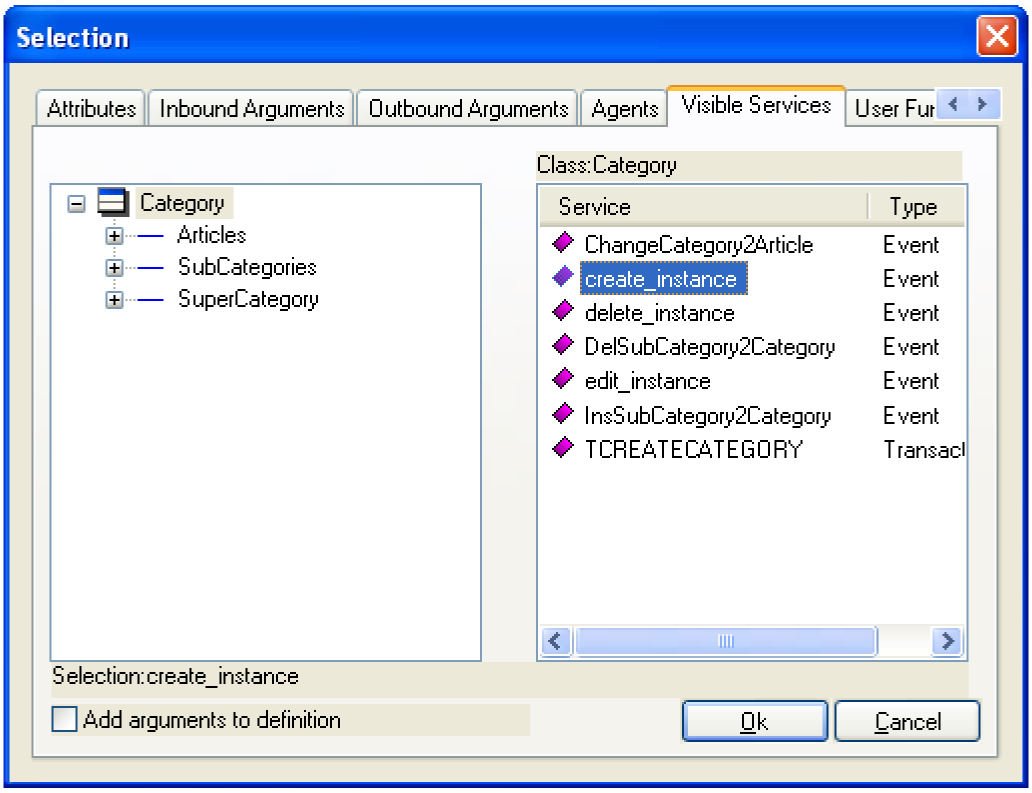
Now, you can add the argument 'pt_Name' that must have the same data type that the argument 'p_atrCategoryName' of the create_instance event, that is, string(35).

Note: There is another way of creating inbound arguments in a transaction; we will see it later on in this tutorial.

When a transaction is marked as 'New' transaction, it means that the first action of the formula must be the creation of the object (for instance, the creation event of that class). After that, any service can be executed on the recently created object referring to it with the reserved word *THIS*.

Let's write the transaction formula. Go to the *Transactions* tab where you can see that the formula is initially empty. Click on *Help* button to access to *Help Navigation Window*.

Next, go to *Visible Services* tab and select the *create_instance* event of the 'Category' class.



Class:Category

Service	Type
ChangeCategory2Article	Event
create_instance	Event
delete_instance	Event
DelSubCategory2Category	Event
edit_instance	Event
InsSubCategory2Category	Event
TCREATECATEGORY	Transacl

Selection:create_instance

Figure 2 Add create_instance of 'Category'class

Click *OK* button.

You will see that now your Transaction formula looks like:

```
create_instance(p_agrSuperCategory, pt_p_atrCategoryName)
```

Replace the argument names given by default by the corresponding arguments created for the transaction. As we are creating a main category, then there is no super category so,, p_agrSuperCategory has to be replaced by NULL (reserved word that represents the NULL value). The pt_p_atrCategoryName has to be replaced by the argument 'pt_Name'. The result is:

```
create_instance(NULL, pt_Name)
```

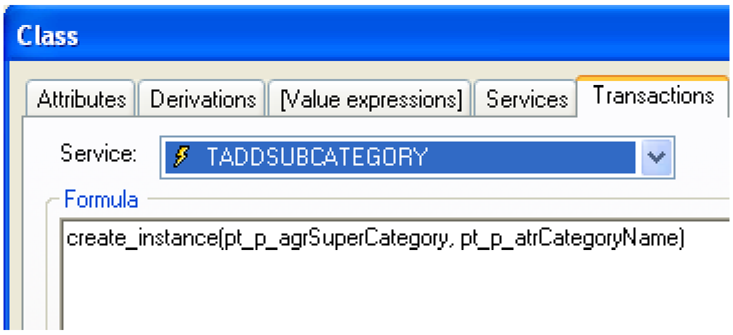
Finally, click on *OK* button.

When the *EditionClass* form is closed, the transaction formula is validated and stored.

Note: If any error appears in the formula, **Integranova Modeler** shows a message box with the error information. The analyst can choose to solve the error or to save the formula with errors. While there are errors in the model, it will not be valid and it will not be able to be translated into source code.

After creating and defining the 'TCREATECATEGORY' transaction, now, you can add another transaction named 'TADDSUBCATEGORY' that creates a subcategory related to a category. It has two arguments: 'pt_SuperCategory' of 'Category' type and 'pt_CategoryName' of string(35) type. Both of them are compulsory.

The transaction formula should be:



Service: TADDSUBCATEGORY

```
create_instance(pt_p_agrSuperCategory, pt_p_atrCategoryName)
```

Figure 3 Translation formula for add subcategory

2.2 Deleting a Purchase Order

It is very obvious that when you want to delete a purchase order, its lines should be also deleted. The lines are related with the purchase order, then, we can delete them from a *local transaction*.

Let's go to create and define a transaction to delete a purchase order and its lines.

Since we are working in an object oriented method, we are forced to delete the lines related to a purchase order before we can delete the purchase order itself.

Note: Notice that if we observe the cardinalities of the relationship between 'POLine' and 'PurchaseOrder' classes, we can see that a line must be always related with a purchase order.

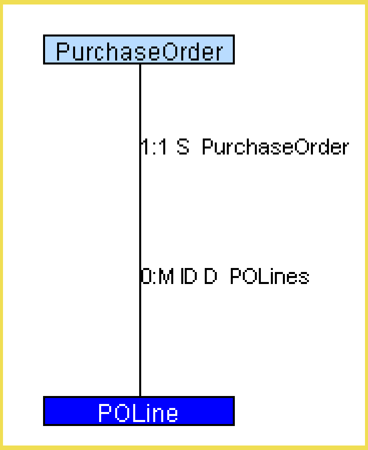
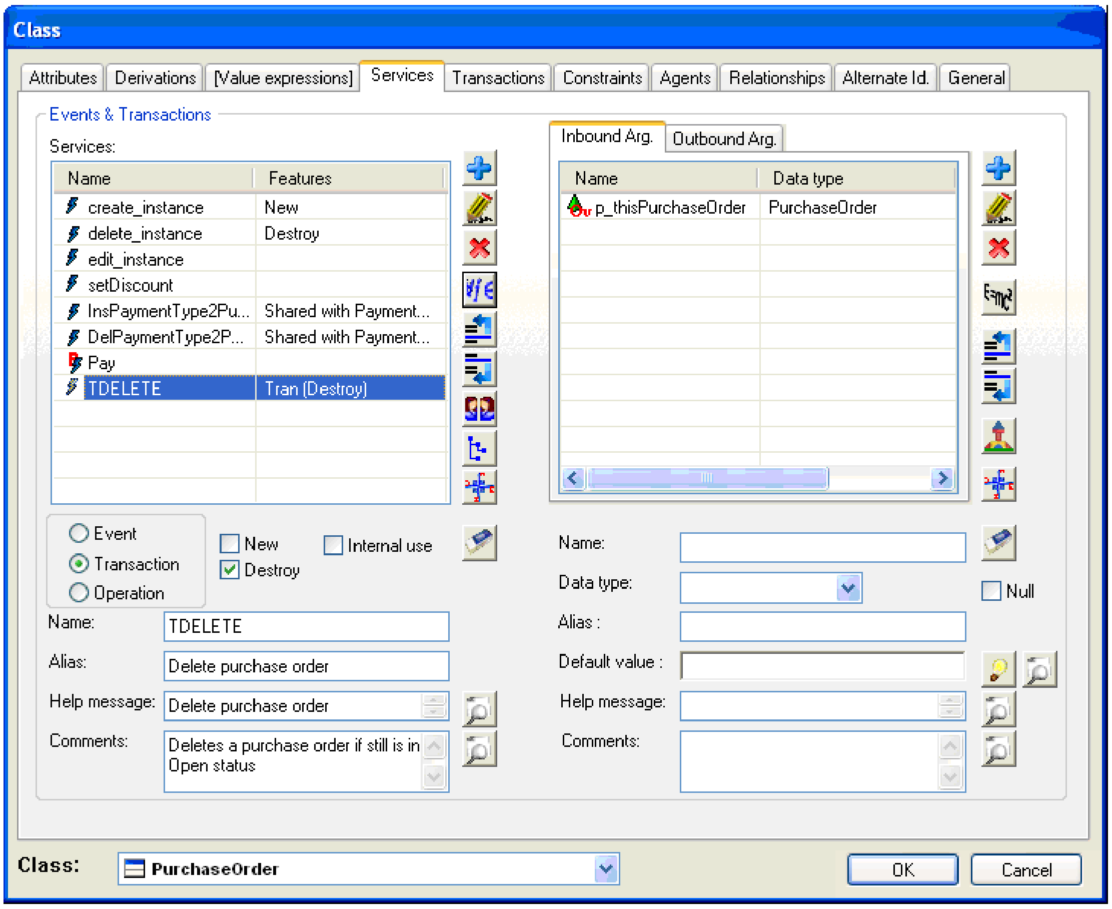


Figure 4 Relationship between 'POLine' and 'PurchaseOrder'

First of all, we have to highlight that the purchase order has to be deleted, that means that it is a 'Destroy' transaction (the *Destroy* checkbox must be checked).

So, go to 'PurchaseOrder' class and select *Service* tab. Create a transaction named 'TDELETE', with the *Destroy* check checked.


Class

Attributes | Derivations | [Value expressions] | Services | Transactions | Constraints | Agents | Relationships | Alternate Id. | General

Events & Transactions

Services:

Name	Features
create_instance	New
delete_instance	Destroy
edit_instance	
setDiscount	
InsPaymentType2Pu...	Shared with Payment...
DelPaymentType2P...	Shared with Payment...
Pay	
TDELETE	Tran (Destroy)

Event [✓ Transaction] Operation

[ ] New [ ] Internal use
[✓] Destroy

Name: TDELETE

Alias: Delete purchase order

Help message: Delete purchase order

Comments: Deletes a purchase order if still is in Open status

Inbound Arg. | Outbound Arg.

Name	Data type
p_thisPurchaseOrder	PurchaseOrder

Name:

Data type:

[ ] Null

Alias :

Default value :

Help message:

Comments:

Class: PurchaseOrder

OK | Cancel

Figure 5 TDELETE transaction for 'PurchaseOrder'

You can see in the inbound arguments area that a new argument 'p_thisPurchaseOrder' has been automatically created. This argument represents the object THIS which is indeed the object that will be deleted.

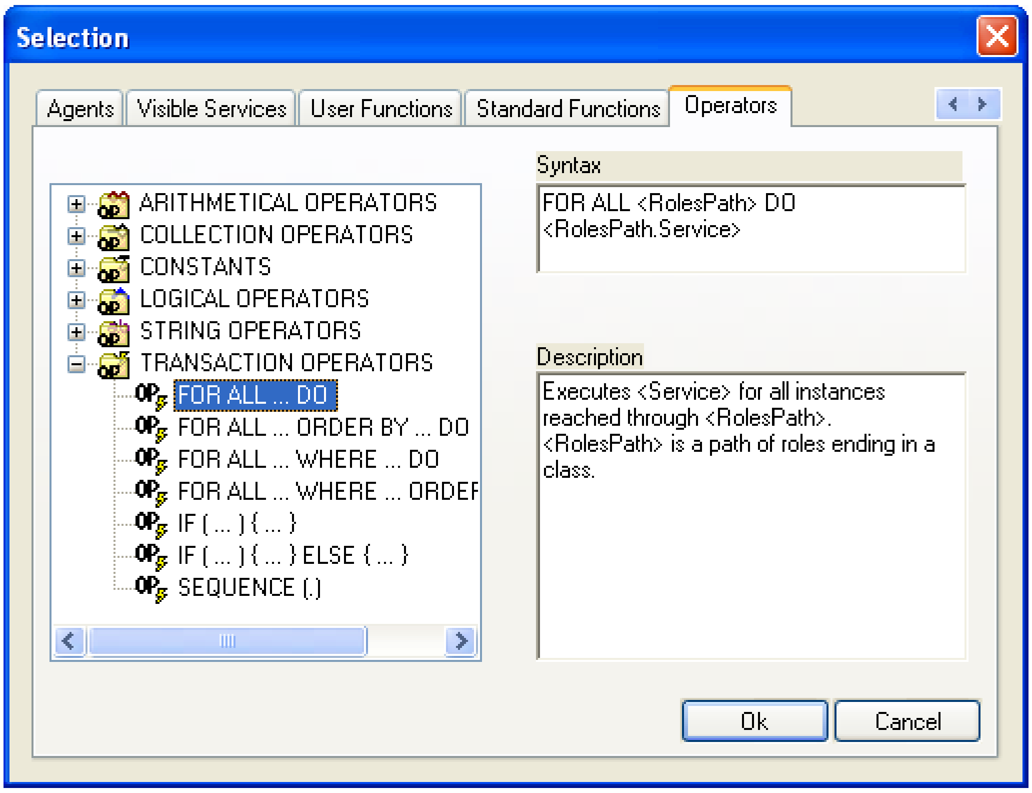
This transaction has been marked as *Destroy*. That means the last action that must be called will be the destruction of the object. However, any other action can be done before the object is deleted. That's why we have to delete the lines before deleting the purchase order itself. The way in which we can express that in Integranova is defining a transaction formula in the 'PurchaseOrder' class like:

```
FOR ALL POLines DO POLines.delete_instance(POLines).

delete_instance(THIS)
```

Let's now define this formula. Go to the *Transactions* tab and in the *Formula* text box click on *Help* button to access at *Help Navigation Window.*

Choose the *FOR ALL … DO* association operator in the *Operators* tab form *TRANSACTION OPERATORS*

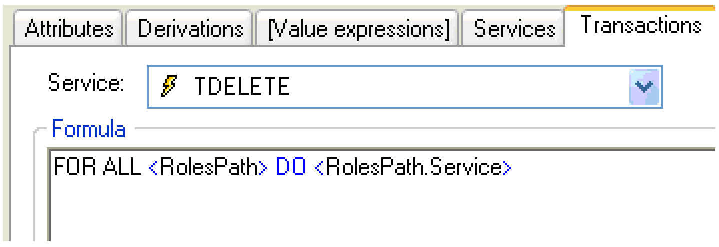


FOR ALL <RolesPath> DO <RolesPath.Service>

Executes <Service> for all instances reached through <RolesPath>. <RolesPath> is a path of roles ending in a class.

Figure 6 Add operators

After clicking the *Ok* button, we have the skeleton of this operator:

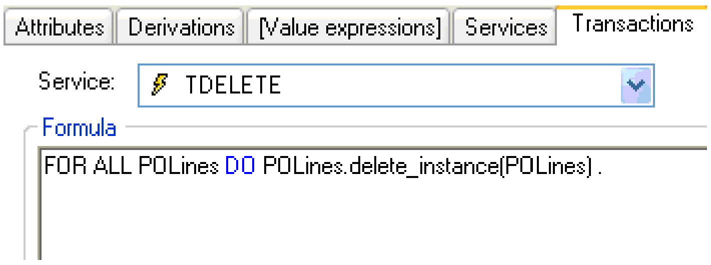


Service: TDELETE

```
FOR ALL <RolesPath> DO <RolesPath.Service>
```

Figure 7 Skeleton of the FOR ALL

We can choose the roles, service, etc. by clicking the *Help* button that access to *Help Navigation Window*. We can also write the formula manually.



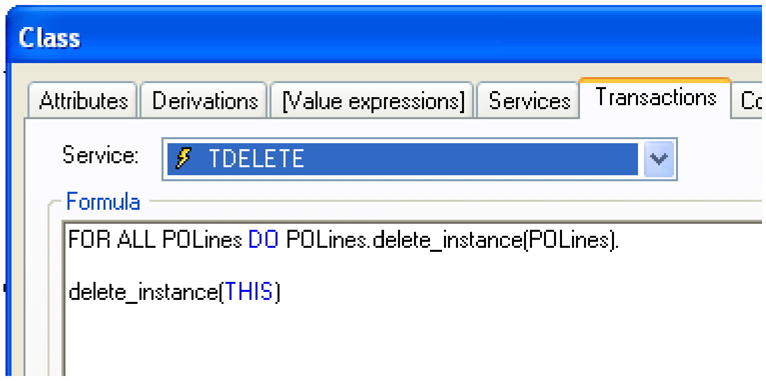
Service: TDELETE

```
FOR ALL POLines DO POLines.delete_instance(POLines) .
```

Figure 8 Delete all the lines of the purchase order

Note: We need to use the dot (".") to chain all the action we are going to do, this is very important, if it not write in the formula it does validate.

After that, we have to complete our formula adding the 'delete_instance' event of the 'PurchaseOrder' class. The current object of the 'PurchaseOrder' class is represented by the reserved word 'THIS'.



Service: TDELETE

```
FOR ALL POLines DO POLines.delete_instance(POLines).

delete_instance(THIS)
```

Figure 9 Formula of TDELETE transaction of 'PurchaseOrder'

Now, we have to add the agents for this service that are: 'Customer' and 'Admin' classes.

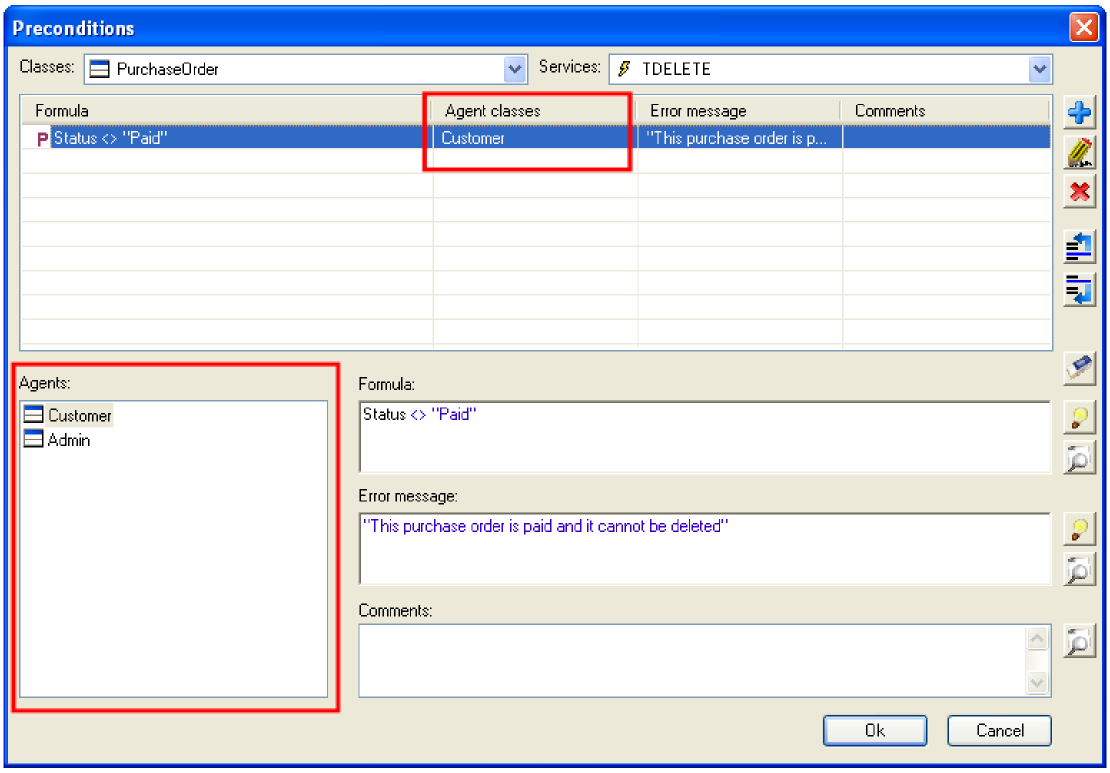
Customers and administrators can delete a purchase order and its lines, but customers can only delete them once they have been paid, because, if a customer deletes a purchase order that he has already paid, it is probably that he will never receive the ordered articles.

In another hand, the administrator can delete erroneous purchase orders, or old purchase orders (that have been already paid).

To cover this requirement, as you already know, **Integranova Modeler** offers the *preconditions*. You need to add a new precondition for the transaction, with a formula as following:

```
Status <> "Paid"
```

But, this formula should only be checked by 'Customer' class because 'Admin' class has permissions for deleting purchase orders when it is needed. The only thing you have to do is select the 'Customer' class from the *Agents* box. This precondition will only be applied when a customer tries to delete a purchase order.



Classes: PurchaseOrder   Services: TDELETE

Formula	Agent classes	Error message	Comments
P Status <> "Paid"	Customer	"This purchase order is p...	

Agents:
- Customer
- Admin

Formula:
Status <> "Paid"

Error message:
"This purchase order is paid and it cannot be deleted"

Comments:

Figure 10 Precondition to a transaction

2.3 Paying a Purchase Order

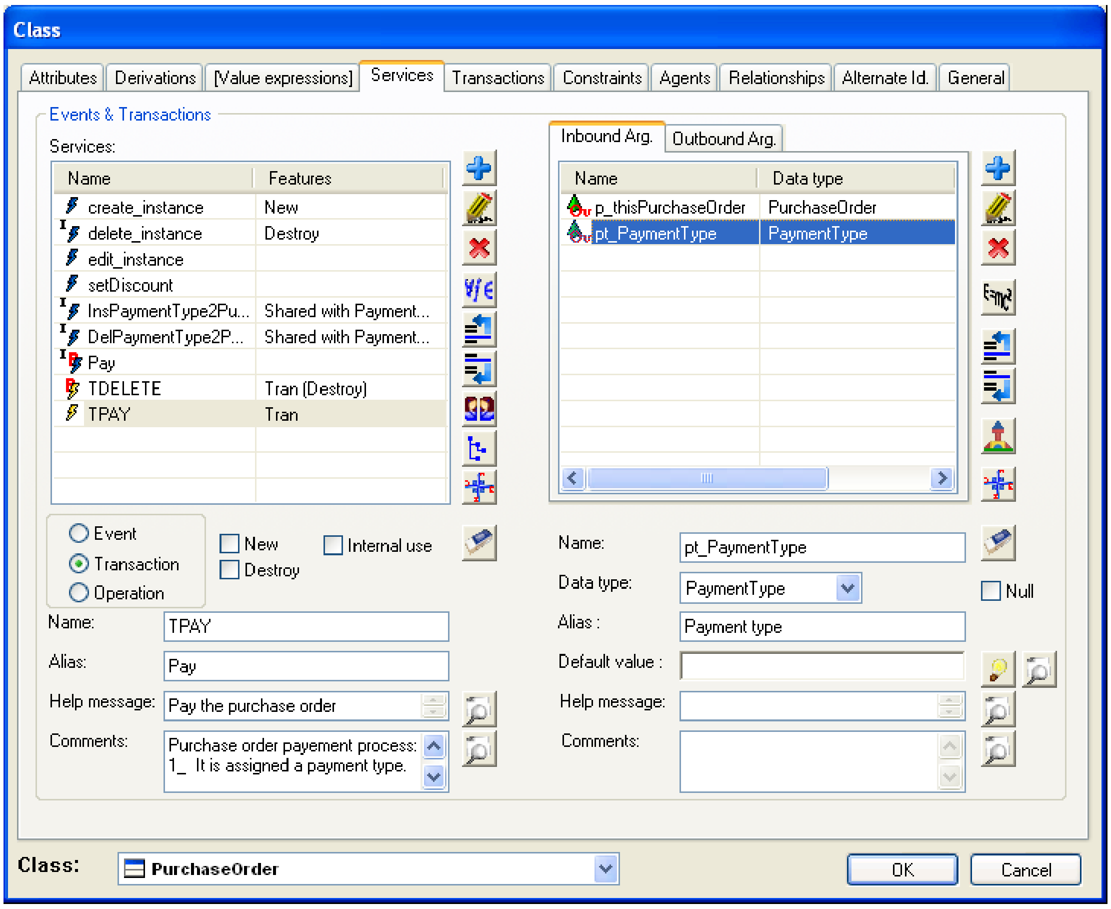
Once we have finished including order lines, it is time to pay the purchase order. When a customer pays a purchase order, it is mandatory to introduce the payment type.. To do this we are going to create a transaction called *TPAY,* this transaction sets the payment type and changes the status of the purchase order to 'Paid'.

To define the transaction formula, it will require the deletion and insert events between 'PaymentType' and 'PurchaseOrder' classes to modify the relationship between them. Besides, it will be needed to set the status to 'Paid', so the *Pay* event will be used too.

As we saw previously, to create a transaction, double click on 'PurchaseOrder' class. After that, select the *Services* tab and then, fill all required fields for the transaction definition. Finally, press the *Add* button.

Now, we need to add the arguments of *TPAY* transaction. By default, 'p_thisPurchaseOrder' argument has been automatically added by **Integranova Modeler.** This argument represents the purchase order that is going to be paid. Besides, the payment type will be required too, so add the *pt_PaymentType* argument with the data type *PaymentType*.

The transaction will look like this:



Services:

Name	Features
create_instance	New
delete_instance	Destroy
edit_instance	
setDiscount	
InsPaymentType2Pu...	Shared with Payment...
DelPaymentType2P...	Shared with Payment...
Pay	
TDELETE	Tran (Destroy)
TPAY	Tran

Name	Data type
p_thisPurchaseOrder	PurchaseOrder
pt_PaymentType	PaymentType

( ) Event [✓ Transaction] ( ) Operation
☐ New ☐ Destroy ☐ Internal use

Name: TPAY
Alias: Pay
Help message: Pay the purchase order
Comments: Purchase order payement process: 1_ It is assigned a payment type.

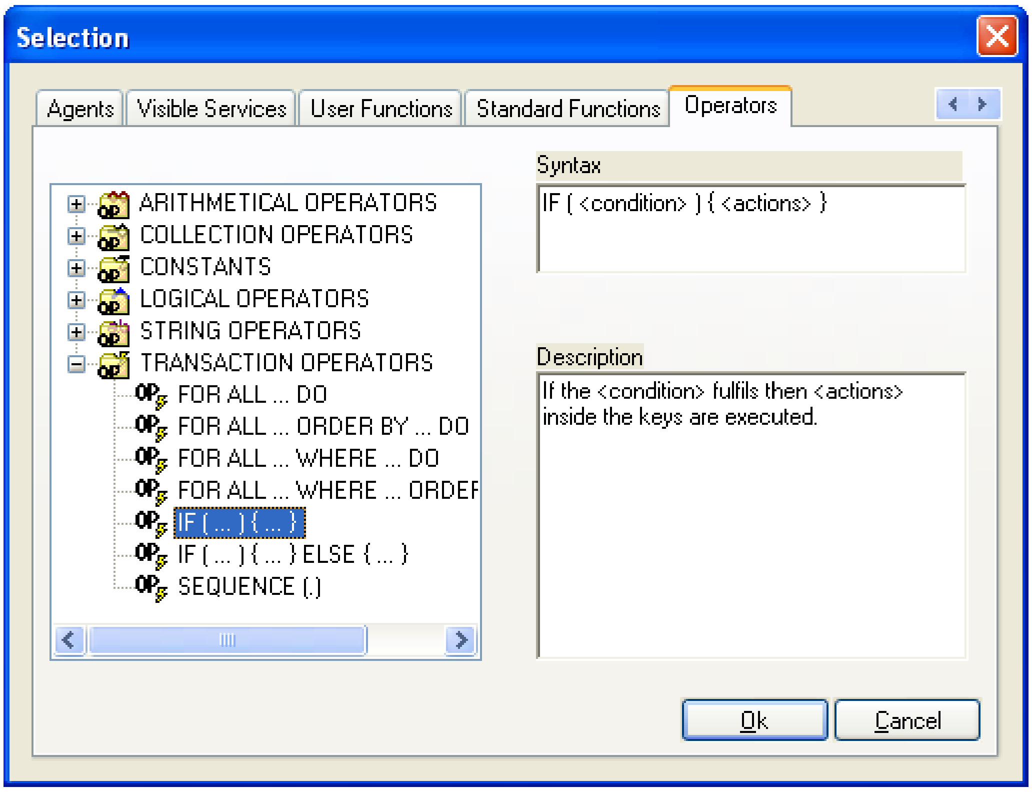
Name: pt_PaymentType
Data type: PaymentType
☐ Null
Alias : Payment type
Default value :
Help message:
Comments:

Class: PurchaseOrder

Figure 11 TPAY definition

Now, we are going to define its transaction formula. Go to *Transactions* tab and in the *Formula* text box click on the *Help (F2)* button to access the *Help Navigation Window.*

Choose the *IF (…) {…}* association operator from *TRANSACTIONS OPERATORS* located in the *Operators* tab.



Syntax

IF (<condition>) { <actions> }

Description

If the <condition> fulfils then <actions> inside the keys are executed.

Figure 12 IF operator

The *IF (…) {…} ELSE {…}* statement is a transactional operator which controls the execution flow. The complete IF-ELSE statement is:

```
IF (Condition)
{
        Sequence of actions
}
[ELSE
{
        Sequence of actions
}]
```

This operator checks a well-formed boolean formula (the condition). Depending on its result, the executed sequence of actions will be:

- ✓ **TRUE**: The sequence of actions inside the 'IF' will be executed.
- ✓ **FALSE**: The sequence of actions inside the 'ELSE' will be executed.

NOTE:

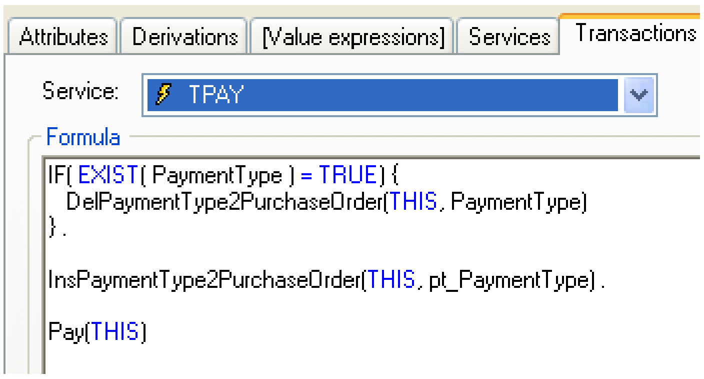
The **ELSE** part is optional.

IF-ELSE statements can be nested.

'**.**' dot operator should be added in order to separate different action:

IF (…) {…} ELSE {…}**.**

We have to complete the formula, until we obtain the following:



```
IF( EXIST( PaymentType ) = TRUE) {
  DelPaymentType2PurchaseOrder(THIS, PaymentType)
} .

InsPaymentType2PurchaseOrder(THIS, pt_PaymentType) .

Pay(THIS)
```

Figure 13 *TPAY* Transaction formula

The *translation* into text of the formula is:

First, if the purchase order has already assigned a PaymentType, then, this relationship is deleted since to define a new relationship with the payment type passed as inbound argument. After that, the payment type passed as inbound argument is related to the current purchase order. Finally, the state of the current purchase order is set to 'Paid'.

2.4 Editing an Article

Now, we want to make the life of an administrator easier and we want to allow him to edit (using only one service) the name, the description, the price and the category of an Article. In order to do different actions in one, we need a transaction.

The transaction will use the *edit_instance* service to edit the name, the description and the price. Moreover it will use the *ChangeCategory2Article* change event to replace the related category. The formula will be placed in 'Article' class.

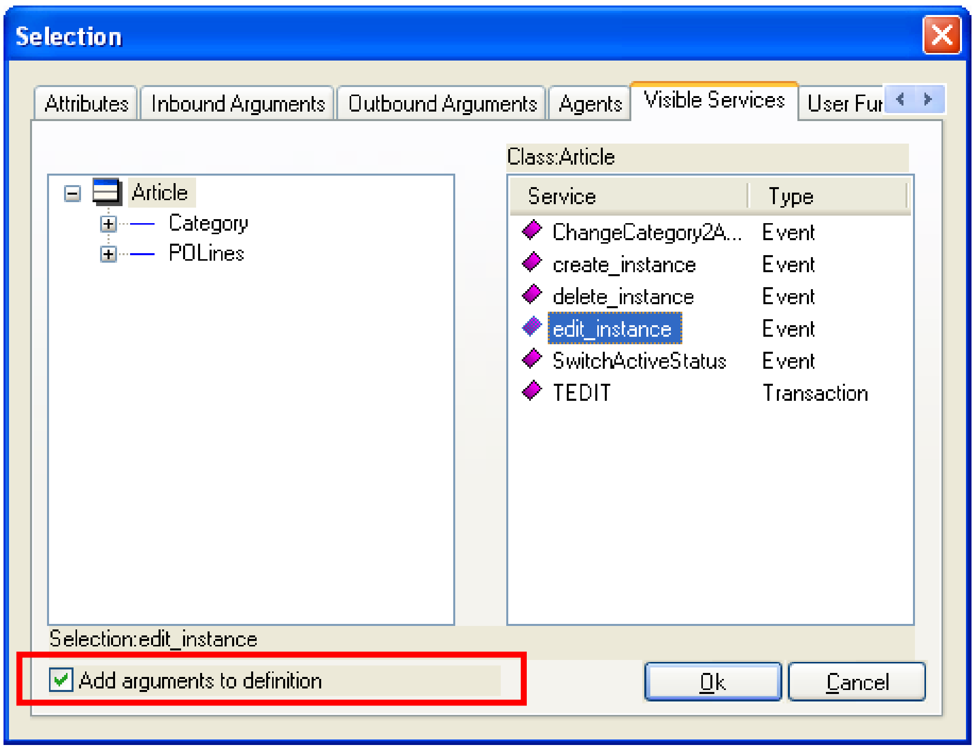
First of all, let's create the *TEDIT* transaction. Fulfill all the required data and press the *Add* button.

After that, go to the *Transactions* tab and click on *Help* button to access the *Help Navigation Window (*press F2 Key as alternative*).*

Next, go to the *Visible Services* tab and choose the *edit_instance* event.

In this form, we have to check the *"Add arguments to definition"* checkbox. This option will automatically create as many arguments as arguments were defined in the *edit_instance* event in the *TEDIT* transaction.

Note: *Add arguments to definition* checkbox adds to the definition of any transaction (or operation), as many arguments as the selected service has. The arguments will be of the same type, with the same alias, etc… The only change is the name, because "pt_" is prefixed.



Article
Category
POLines

Class:Article

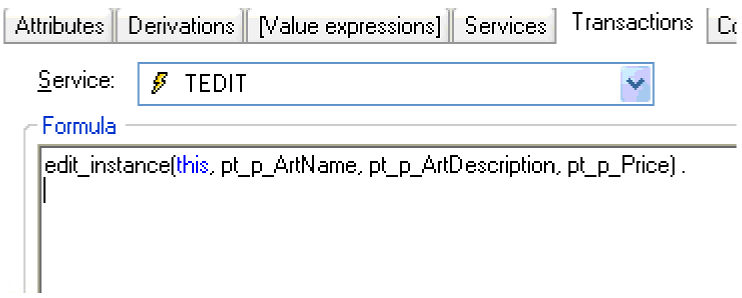
Service	Type
ChangeCategory2A...	Event
create_instance	Event
delete_instance	Event
edit_instance	Event
SwitchActiveStatus	Event
TEDIT	Transaction

Selection:edit_instance

[✓] Add arguments to definition

Figure 14 Add arguments to definition

The result will be the following:



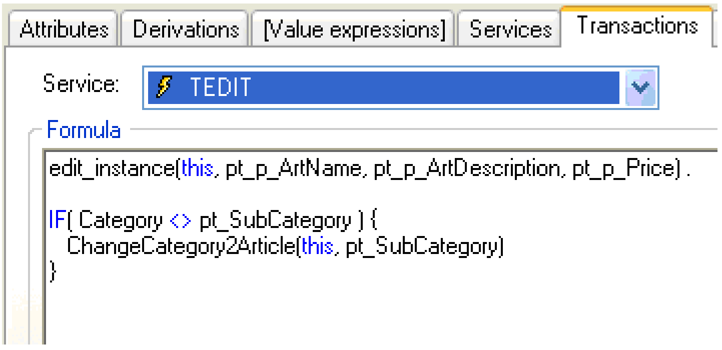
Service: TEDIT

Formula

```
edit_instance(this, pt_p_ArtName, pt_p_ArtDescription, pt_p_Price) .
```

Figure 15 filling the formula

Then, you can fulfill the formula in the way you have learned to get this:



Service: TEDIT

Formula

```
edit_instance(this, pt_p_ArtName, pt_p_ArtDescription, pt_p_Price) .

IF( Category <> pt_SubCategory ) {
  ChangeCategory2Article(this, pt_SubCategory)
}
```

Figure 16 Final TEDIT Transaction Formula

In the formula, we have to remark that the *ChangeCategory2Article* event is a *change event* (see Tutorial 3 in the series), and it can only change a subcategory by another one. Then, we have to verify that the category has changed in order to replace it by another one. If this service would be directly called with the same category, it would fail.

3 Defining internal services

In this point of our model, we are going to introduce the concept of *Internal Service*. Services must be marked as 'Internal' to ensure that they cannot be directly executed by an application user but remaining accessible from transactions (or operations) formulas. These services will never appear in the User Interface.

The **Integranova Modeler** offers the possibility of defining either internal or non-internal services depending on what the analyst wants to offer in the application's user interface.

✓ What will happen when a service is set as *Internal*?

- It is not as simple as it seems to be. When a service is set as internal, it loses all its defined agents permissions and will not be part of the interface. Then it loses all properties that the analyst could have been defined for this service as interaction in the final interface.

✓ What would happen if we set an internal service as *non-Internal*?

- If you change an internal service back to a non-internal service, the above mentioned properties need to be re-defined.

Now, how can I define a service as *Internal*? It is very easy, let's see how to do it.

Previously in this tutorial, we have created the *TDELETE* transaction for deleting purchase orders. Now we are going to mark the *delete_instance* event as internal service (because we do not have to offer it to the user).

Then, Double click on 'PurchaseOrder' class and select the *Service* tab. Next, double click on the 'delete_instance' event and check the *Internal use* check box.

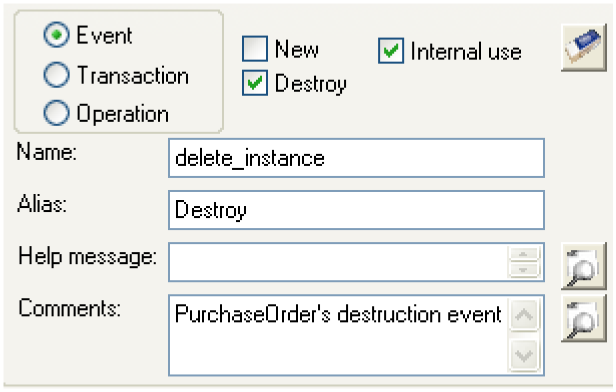

[✓ Event] Transaction Operation
☐ New
☑ Destroy
☑ Internal use
Name: delete_instance
Alias: Destroy
Help message:
Comments: PurchaseOrder's destruction event

Figure 17 Mark as internal

Finally press the *Modify* button.

Once the event is internal, you can see that its related icon has changed from to .

Note: If we set a transaction as internal, its icon changes from to .

Now, it is time to mark as *Internal use* all the events that we have replaced by transactions because it would be desirable that they will never appear in the user interface. They are:

✓ In 'Article' class, the *edit_instance* and *ChangeCategory2Article* events

✓ In 'PaymentType' class the events: *InsPaymentType2PurchaseOrder* and *DelPaymentType2PurchaseOrder*.

- ✓ In ‘PurchaseOrder’ class, the same events (*InsPaymentType2PurchaseOrder* and *DelPaymentType2PurchaseOrder*) and also *delete_instance* and *pay*.

- ✓ In ‘Category’ class, the events to be checked as internal will be: *ChangeCategory2Article*, *InsSubCategory2Category*, *DelSubCategory2Category*.

4 Default user interface

Integranova Modeler offers the possibility of creating a customized user interface using the features associated to the *Presentation Model*. All these features will be presented from Tutorial 9 in the series tutorials to the last one (Tutorial 15).

Meanwhile, **Integranova Modeler** provides a default user interface where all the functionality defined in the model can be accessed. It will be possible to access to the application using any of the agents defined in the model.

The main menu will have as many menu entries as classes defined in the model (the name will be the alias of the class). Each menu entry will have the following subentries: one scenario to show **one instance** of the class, another scenario to **list all the instances of the class** and as many scenarios as **non-internal services** we have defined in the class.

When an agent is connected to the application, he only will have access to the services that he can execute and the roles and attributes over which he has visibility. The menu entries which the agent has no permission (no execution, no visibility, no navigation) will be hidden.