Object selection, roles and nodes

Overview of generated code

Each dialog is drawn on the screen by a *<dialog name>DefaultView* class (<dialog name> is substituted with the actual name of the dialog), and is controlled by a *<dialog name>DefaultController* class. For each node in the object selection, a *<node name>Methods* class is generated (<node name> is substituted with the actual name of the nodes). These classes are used by the controller to interact with the domain objects, and to traverse the object selection tree. The controller's reference to a <node name>Methods class is named *the<node name>*.

The methods classes represent a single instance of a domain object, thus a more-related node needs a second class to represent the collection of domain objects. The java target in Genova 8.1 only supports list blocks as a representation of a more-related node, and the edit fields represent the a single instance (controlled by the <node name>Methods class). Each line in a list block is controlled by a *LineListblock<node name>_LISTBLOCK* class, and the inner class *ListblockMethods<node name>_LISTBLOCK*, residing in the default controller, is used to reference the list block. The reference to the list block class is named *listblockMethods<node name>LISTBLOCK*.

The classes mentioned so far are always generated, and the programmer should not make any changes to these classes as they will be lost the next time the dialog is generated. Both the controller and the view class are subclassed by respectively a *<dialog name>Controller* and a *<dialog name>View* class. When generating, these classes will not be overwritten and can be safely modified by the application programmer. Figures 1 and 2 below, illustrates an object selection and the corresponding generated classes.

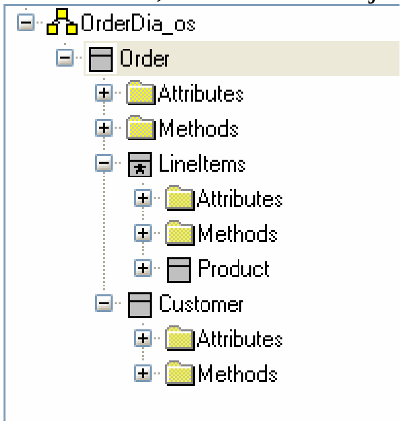


Figure 1: An object selection consisting of four nodes.

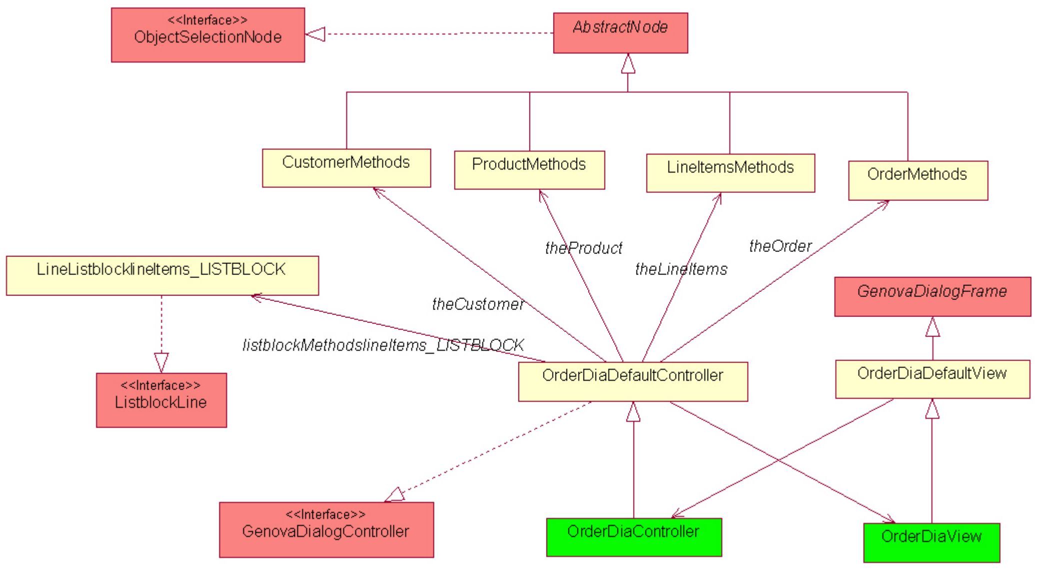


Figure 2: Generated classes based on the object selection in figure 1.
The classes marked in green are not replaced when generating and is intended for customization by the application programmer. The classes in yellow are always generated. The classes and interfaces in red are part of the Genova Runtime library.

.

Obtain and display

This section describes how domain objects are displayed in a dialog, and how they are updated (obtained) with data from the dialog. The controller uses the corresponding methods-class in order to achieve this. E.g. in order to display a customer object, the controller invokes

```
theCustomer.display(aCustomerObject);
```

To update an order object, the controller invokes

```
theOrder.obtain();
```

Displaying an attribute

Six display methods can be used in order to display a single attribute. All take the name of the attribute as the first parameter. The second parameter can either be a value object, a string representation of the attribute value, or a domain object. The third, optional parameter (named resetState) is a boolean value, used to control the changed status of a field. If it is “false” the display is interpreted as if the user had typed into the field. If it is true, the changed status of the field is reset, and the field is considered unchanged. The latter is equivalent to the two-parameter variant.

Displaying an order’s date using a Date-object:

```
Date today = Calendar.getInstance().getTime();
theOrder.display(“orderDate”, today);
```

Displaying a customer’s name using a domain object, simulating that the user typed the name (the three parameter variant):

```
Customer customer = ...;
theCustomer.display(“name”, customer, false);
```

Displaying a customer’s balance using a string:

```
theCustomer.display(“balance”, “250.75”);
```

Note that when displaying an attribute using a string, it must be possible to parse the string into the correct value type. Dates displayed in this manner must adhere to the internal date format, which is: “dd.mm.yyyy”.

Automatic modification checks

Genova comes with a set of predefined checks that are performed on actions that alter the state of the dialog. The automatic checks run through all involved nodes depending on the action and issues a warning if any unsaved changes (made by the user) will be lost. The check methods reside in the AbstractNode class, which is the super class of the <nodeName>Methods classes.

Check find and check find all

A find (and find all) action checks that at most the key-field of the target node is changed. If any other field is changed (on any node including and below the target node), a warning is issued to the user, providing a chance to cancel the ongoing action.

CheckFind traverses the object selection as follows:

1. Invoke check find on the target node
2. Invoke “check change as child” on child nodes.

CheckFind can be aborted by overriding hookCheckFind<node name>.

Check delete

Check delete checks that no fields are changed. If any field is changed, a warning is issued to the user, providing a change to cancel the ongoing action.

CheckDelete invokes checkChange on the target node.

CheckDelete can be aborted by overriding hookCheckDelete<node name>.

Check save

Check save, (invoked as part of a save, insert or update action) does not perform any check by it self, but simply invokes hookCheckSave<node name> for all nodes involved in the save action.

Check close

Check close is performed when the user closes the dialog. Check close checks that no unsaved data will be lost by invoking check change on each root node. Check close can be aborted by overriding the hookCheckClose<node name> method.

Check row select

Check row select is invoked when the user selects (or unselects) a row in a list block. Check row select invokes check change on the list block's object selection role, and can be aborted by invoking hookCheckRowSelect<node name>.

Check change

Check change is invoked as part of the other check methods. Check change starts by checking that no fields are changed, and then invokes checkChangeAsChild on each child node. Check change can be aborted by overriding hookCheckChange<node name>.

Check change as child

Check change as child invokes check change. In addition, if the child node is up-related, it checks that the relation between the parent domain object and the child domain object is intact. If the parent node is either *cleared* or *edited* (the un-displayed super state, see chapter about node state), the relation is always ok. Otherwise, the domain child object (as seen from the parent) is compared with the displayed domain child object. The comparison is based on the two object's hash code.

Obtaining an attribute

When obtaining an attribute, the displayed string is returned. The supplied parameter specifies the name of the attribute. E.g. obtaining the displayed name of a customer:

```
String name = theCustomer.obtain("name");
```

Server actions, recursive obtain and display

Genova comes with a set of pre-defined CRUD actions. On the client side, each action consists of five steps:

1. Check dialog for modifications
2. Recursive obtain object selection
3. Invoke server action

4. Merge result with displayed objects (only applies to save, update and insert actions)
5. Recursively clear the dialog from target node
6. Recursively display the result

The first step is described in detail above, the rest is described below.

Each server action is generated in a method named action<actionName><targetName>. E.g. a find action on a customer will generate the method actionFindCustomer, which will again be called from the method defined by the application programmer in Genova's event dialog. It may be advisable to look at a generate server action (they involve some thread code which can be ignored for the sake of this chapter) while reading the next sections.

Recursive obtain

The second step of any server action is a recursive obtain, where all domain objects are updated with information taken from the dialog. The recursive obtain checks each node's state before obtain. If a node is CLEARED and the node's children are also CLEARED, the node is not obtained. Node state is described thoroughly in the section titled "Node state" below.

Invoke server action

After the recursive obtain, the ObjectSelelction object initialized, and a hookAction<actionName><targetName>Obtain is invoked. This hook can be used by the application programmer to modify the ObjectSelection, e.g. setting up FindData etc.

After the obtain-hook, hookAction<actionName><targetName>ClientContext is invoked, giving the application programmer the opportunity to supply a custom client context. If the return value is null, the default application client context is used. Finally the server action is invoked, passing the ObjectSelection and the clientContext as parameters, and the result is stored.

Merge the result with displayed objects

Only one instance of more-related roles that lies above the target node are transferred between the server and the client. E.g. if the target is "Product" only one instance of LineItem is obtained. (This saves network traffic and allows the server part of the action to navigate to the target object). Thus the client must now merge the sets returned from the server (they only contain one instance) with the original displayed sets. E.g. the previously displayed order's set of LineItems is merged with the returned order's set of LineItems.

Clear the dialog

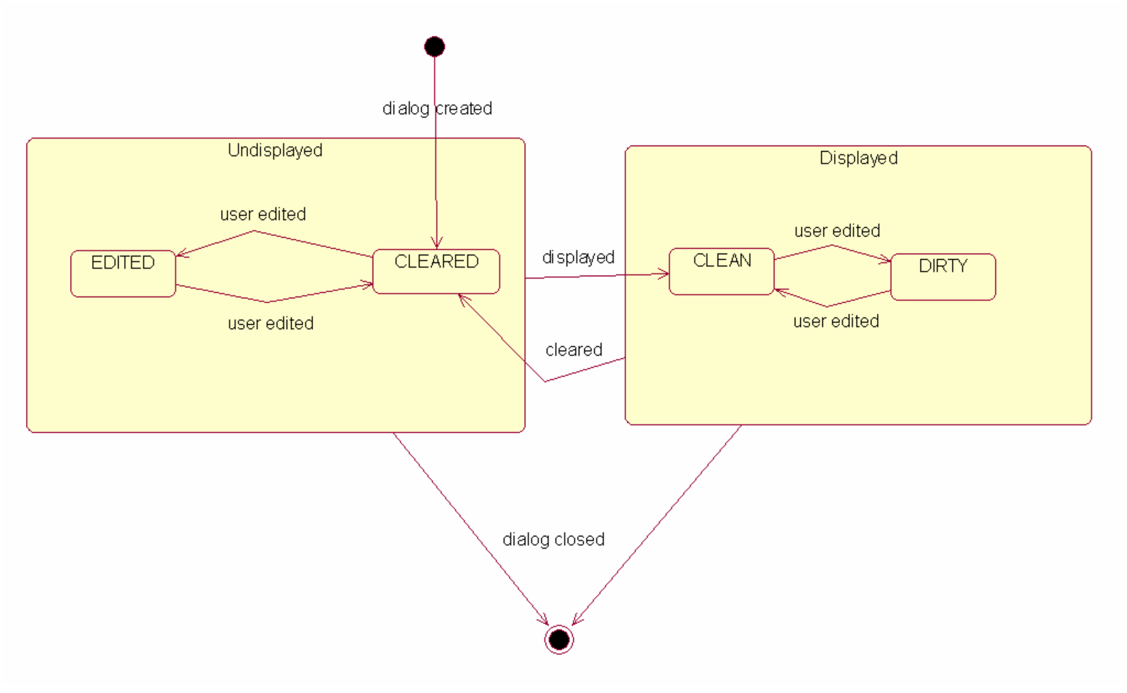
After the server call returns, the result is passed to hookAction<actionName><targetName>Display, before the dialog is cleared recursively from the target node and down. A find and find all action does not clear the key-fields on the target node. Thus if nothing was found on the server, the key-fields are preserved.

Recursive display

The last and final step of a server action is to recursively display the result.

Node state

A node has four states, two “un-displayed” states, and two “displayed” states.



The initial state of the node is the *cleared* state. In this state the node is not displaying any domain object, and the user has not edited any fields belonging to that node. If the user edits a cleared node, it changes to the edited state (if the users editing actually changed the contents of any fields). If a domain object is being displayed by the node, the node is either *clean* or *dirty*, depending on whether the user has edited any fields.

, A nodes state is derived by checking each data field of the particular node. Depending on the super state of the node, a data field reporting that it is changed set’s the node to either *edited* or *dirty*. If all data fields are unchanged, the node is either *cleared* or *clean*.

Recursive obtain, expressions and node state

First of all, it is important to understand that expressions do not affect a nodes super state. In order to fully understand this, we must first look at how a nodes state is derived, and how expressions work.

When inquiring a node's state, the following takes place: If the nodes state has been overridden by the application programmer (by invoking the *setState* method) the overridden state value is returned. Otherwise, the state is calculated by checking each data field belonging to the node. If any fields are changed, either *edited* or *dirty* is returned depending on the node's super state, otherwise either *cleared* or *clean* is returned (again depending on the node's super state).

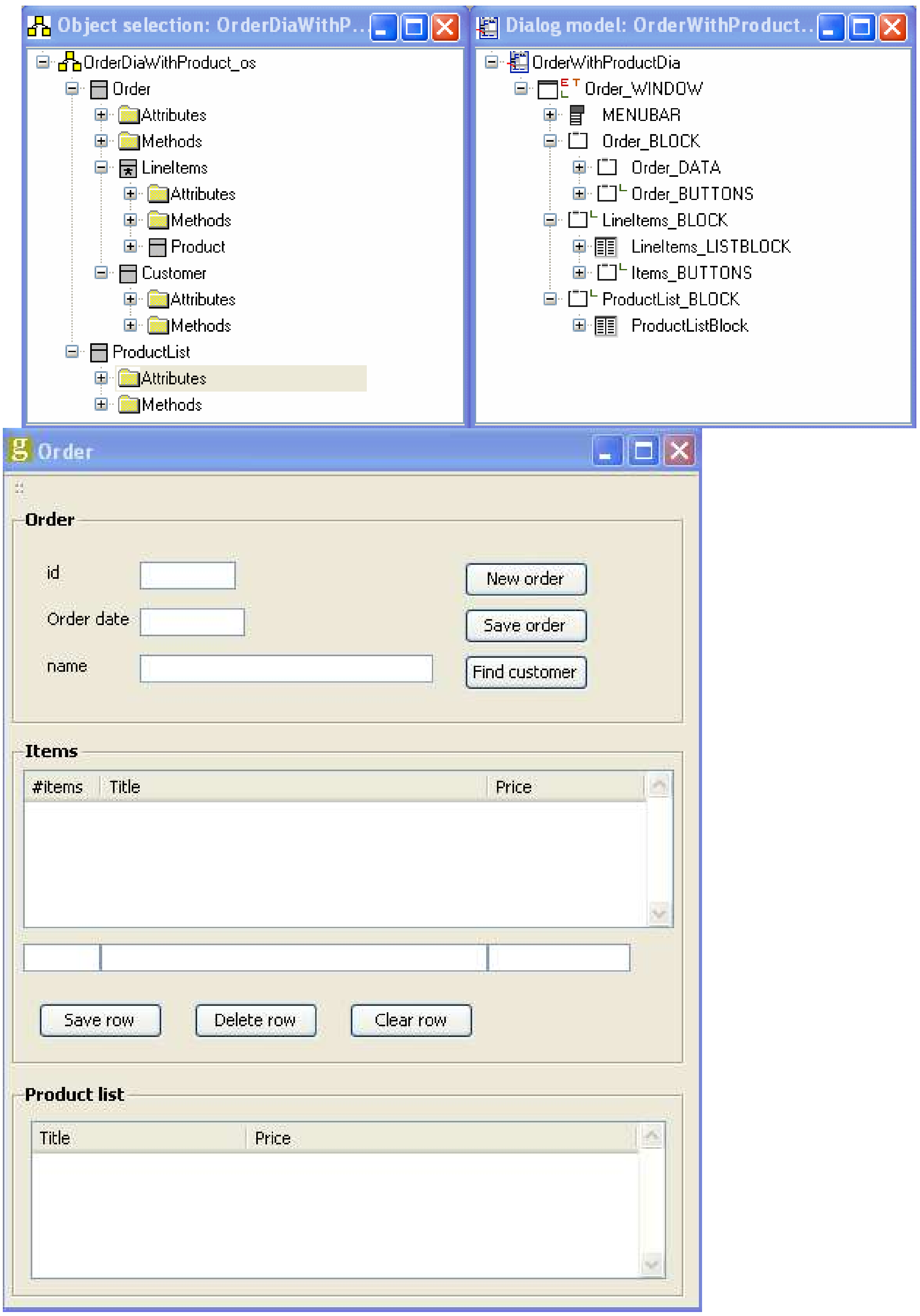
When setting up an expression between two fields, the field's changed status is transferred along with the characters of that field. Thus, if fields A and B are connected with expressions, and the A field is considered unchanged – the B-field is also unchanged. So if A's node is *clean*, the B node stays either *cleared* or *clean* depending on its super state (and if it is *cleared*, it will not be obtained as part of an action). If, however, the user edited the A field, then the B field would be considered changed, and the B node would be either *edited* or *dirty* (again depending on the super state).

If we extend the object selection from figure 1 with a fifth node "ProductList", we can use this node in a list block to display all products, and let the user add new line items by selecting a product from the list of all products. The object selection, the dialog designer and preview of the dialog are shown in figure 3 below.

In order to display the selected record (from the record list) in the edit fields of the line items list, we use expressions to transfer field data from the product list node to the product node. Apparently, this works fine. When the user selects a record in the record list, the line items' edit fields display the record data. But attempting to save a line item fails – a record object is never obtained from the record node, and the line item can not be identified (assuming that line items are identified by order and product). Here the ProductList node is the 'A' node, and the Product node is the 'B' node (from above). The Product node is initially *cleared*, and passing data from the *clean* ProductList node does not alter its state. The solution is to override the Product nodes state, typically at the beginning of the save method. So, if the save action is part of a method named saveLineItems, the application programmer should override the method before hook, ensuring that the state of the Product node is correct:

```
public boolean hooksaveProductBefore(GenovaEventHolder e) {
    theProduct.setState(theProductList.getState());
    return true;
}
```

Although this hook needs some fine tuning (setting the product node's state only if the product list block has a selected line etc.), it illustrates how to override a node's state in order to make the recursive obtain work.


Object selection: OrderDiaWithP...
OrderDiaWithProduct_os
Order
Attributes
Methods
LineItems
Attributes
Methods
Product
Customer
Attributes
Methods
ProductList
Attributes
Methods
Dialog model: OrderWithProduct...
OrderWithProductDia
Order_WINDOW
MENUBAR
Order_BLOCK
Order_DATA
Order_BUTTONS
LineItems_BLOCK
LineItems_LISTBLOCK
Items_BUTTONS
ProductList_BLOCK
ProductListBlock
Order
Order
id
Order date
name
New order
Save order
Find customer
Items
#items
Title
Price
Save row
Delete row
Clear row
Product list
Title
Price
Figure 3

Writing customized code

Customization of generated actions is achieved by overriding various hook methods in the controller class. In this chapter we will use the dialog above, and solve some of the problems we encounter.

When the dialog opens, the user is supposed to enter an order number and click the find button to bring the order up. But what if the user has already made a selection from the product list and entered a number of items? Unless we do something, the action will display a warning, but since no order is displayed, there is very little the user can do about it, thus we want to avoid this warning. The same issue arises if the user clicks the new button (which basically clears the dialog), or if the user starts typing in a customer name before an order is found.

The thing to realize is that this situation occurs when the order node is in one of the un-displayed states, thus we can stop check change on the line item node and the customer node when the order node is in either the cleared or edited state:

```
public boolean hookCheckChangeLineItems() {
    int orderState = theOrder.getState();

    return orderState != ObjectSelectionNode.CLEARED
            && orderState != ObjectSelectionNode.EDITED;
}

public boolean hookCheckChangeCustomer() {
    int orderState = theOrder.getState();

    return orderState != ObjectSelectionNode.CLEARED
            && orderState != ObjectSelectionNode.EDITED;
}
```

When a dialog is displayed, the user might pick a new product from the product list, or edit an already existing item. But we need to make sure that only either a product is selected or an item is selected – not both at the same time (they are both displayed at the same edit-fields, so if both are selected it is unclear which line is actually displayed). The ambiguity is solved by listening for a selected event on both list blocks specifying an activate action. In it, we simply clear the other list block selection.

```
public void unselectInLineItems(GenovaEventHolder e)
        throws RuntimeException {

    listblockMethodsLineItems_LISTBLOCK.clearSelectedLines();
}

public void unselectInProductList(GenovaEventHolder e)
        throws RuntimeException {
```

```
    listblockMethodsProductListBlock.clearSelectedLines();
}
```

But we are not quite finished. First selecting a line from the items list, then selecting a line from the product list before again selecting a line from the items list causes a warning. The warning is issued because the relation between the line item object and the product object is changed. One might consider this to be ok, but it makes the dialog quite obnoxious. It is probably enough that it gives a warning when the user either press the new button or want to find a new order. Thus we want to stop the check when the user selects a row (unless the item field is edited). The list block we want to stop checking is attached to the role LineItems, so we override the hookCheckRowSelectLineItems method:

```
public boolean hookCheckRowSelectLineItems
        (int currentSelection, int newSelection) {

    return lineItemsChanged();
}

private boolean lineItemsChanged() {
    int lineItemsState = theLineItems.getState();
    return lineItemsState != ObjectSelectionNode.CLEARED
            && lineItemsState != ObjectSelectionNode.CLEAN;
}
```

Displaying messages

The last thing to consider is that when the user selects a line from the product list block, no warning is issued even if the user has changed the number of items field. The reason is that there is no expression between the number of items field and a field in the product list block, so changes to the number of items field are never detected by the product list block. So the generated check that occurs when selecting a row in a list block is of no use when it comes to the product list – we must supply our own check.

When the user selects a line from a listblock, two things happens: First a vetoable selection event is triggered, giving listeners a chance to veto the selection change. Then, if the selection change is not vetoed, the list block goes a head an makes the selection change. Thus, we should add a vetoableSelectionChange listener to the product list block. We do this on the opened event on the dialog, in an activate action named “initSelectionListener”. We start by defining an inner class that implements the desired interface. In it, we define a helper method that will display a message to the user if needed.

```
class ProductSelectVetoer implements
        GenovaVetoableSelectionListener {

    public void vetoableSelectionChange
            (GenovaVetoableSelectionEvent e) {

        if (lineItemsChanged()) {
            int oldRow, newRow;
            oldRow = e.getOldSelection();
            newRow = e.getNewSelection();
            boolean ok = displayMessage(oldRow, newRow);
            e.veto(!ok);
        }
    }

    private boolean displayMessage(int oldRow, int newRow) {
        long msgID;
        if (oldRow == newRow) {
            msgID = CRuntimeMsg.CC_CHECK_UNSELECTION_MSG;
        } else {
            msgID = CRuntimeMsg.CC_CHECK_SELECTION_MSG;
        }
        Message msg = RuntimeMsg.get(msgID, null);
        RuntimeInteractor.interact(msg, getWindow());
        Object reply = msg.getReply();
        return MessageReplyType.REPLY_OK.equals(reply);

    }
}
```

Next, we override the activate method and add the listener to the product list block.

```
public void initSelectionListener(GenovaEventHolder e)
        throws RuntimeException {

    view.ProductListBlock_table
            .addVetoableSelectionListener
                    (new ProductSelectVetoer());
}
```

Since we are performing the check our self, the generated check method should not run, thus we override the start hook on the selected event and returns false:

```
public boolean hookCheckRowSelectProductList
        (int currentSelection, int newSelection) {

    return false;
}
```

Finally, we need to decide on some default value that should be displayed in the number of items field when the user selects a product from the product list. We define an activate action on the selected event in the product list and display a default value of one from there.

```
public void displayDefaultLineItemsValue(GenovaEventHolder e) {
    theLineItems.display("numItems", "1");
}
```

Transfer data between dialogs

When creating a new order, the user must supply a customer, and we want the user to pick a customer from a dialog showing all customers. In it, the user selects a customer, and we transfer the selected customer object back to the order dialog.

Transferring data between two dialogs involves two objects: A StartDialog object that keeps track of the *caller* and *callee* (caller is the dialog controller that opens the other dialog – the callee). The other object is a ReturnToDialog object that keeps the transferred data and invokes the caller's return method.

The first part consist of writing an activate action method. In it, we use StartDialog to open the Customer dialog.

The caller (the order dialog)

```
ReturnToDialog rtd = new ReturnToDialog();

public void openCustomerDia() {
    rtd.setCaller(this);
    StartDialog sd = new StartDialog();
    sd.instantiateDialog(getApplication(), "CustomerDia", rtd);
}
```

When we return from the customer dialog, the controller's returnToDialog method will be invoked. Here we display the returned data. Note that the supplied parameter will be the same instance as our previously declared ReturnToDialog, thus we do not need that parameter since the customer dialog can only be opened from the order dialog:

```
public void returnToCaller(Object returnObject) {
    Customer customer = (Customer) rtd.getReturnData();
    theCustomer.display(customer);
}
```

The callee (the customer dialog)

After the customer dialog is created (by invoking StartDialog.instantiateDialog(...)), the initDialog(...) method is called (on the Customer dialog controller). In it, we must keep the reference to the return object:

```
ReturnToDialog rtd;
public void startInit(Object returnObject, Object startObject){
    rtd = (ReturnToDialog) returnObject;
}
```

We define an activate action when the user double clicks a line in the customer list. In it, we obtain the selected customer, set it at the return data, and invokes ReturnToDialog.returnToDialog(). In addition we set up another action hiding the customer dialog (not shown here).

```
public void returnToOrder(GenovaEventHolder e) {
    rtd.setReturnData(theCustomer.obtain());
    rtd.returnToCaller();
}
```