

Creating a Simple Business Collaboration Scenario With a BPMS

Using BPMN 2.0 and Bonita Open Solution

Prof. Dr. Thomas Allweyer

University of Applied Sciences Kaiserslautern

June 2011

Contact:

Prof. Dr. Thomas Allweyer
Fachhochschule Kaiserslautern (University of Applied Sciences)
Department of Computer Science and Micro Systems Technology
Campus Zweibruecken
Amerikastr. 1
66482 Zweibruecken
Germany
Phone: +49 (0)631 3724 5324
E-Mail: thomas.allweyer@fh-kl.de

All brand names, company names, and product names in this paper are trademarks or registered trademarks or trade names of their respective holders.

Contents

1	Introduction	8
1.1	The collaboration scenario	8
1.2	Data structures and messages	10
1.3	Implementation alternatives	12
1.4	Necessary agreements between partners in a business collaboration.....	13
1.5	What is not covered in this example.....	14
1.6	Download and installation.....	14
2	The processes in Bonita	16
2.1	Customer processes	16
2.2	Supplier processes	18
3	The implementation in detail.....	20
3.1	XML schema for messages	20
3.2	Preparing the database	21
3.3	Process “Place Order”	22
3.3.1	Data in the Pool “Place Order”	22
3.3.2	Creating a unique order ID.....	24
3.3.3	Task “Create order”	26
3.3.4	Task “Send order”	27
3.3.5	Intermediate message event “Order response received”	28
3.3.6	The remainder of the process	29
3.4	Process “Send Order”	30
3.4.1	Variables in the process “Send Order”	31
3.4.2	Message start event “Order received for sending”	31
3.4.3	Task “Determine values for storing order”	31
3.4.4	Task “Write order into database”	32

3.5	Process “Retrieve Order Responses”	33
3.5.1	Variables in the process “Retrieve Order Responses”	33
3.5.2	Task “Read response message from database”	33
3.5.3	Second gateway – more messages?	34
3.5.4	Timer message event “Wait for polling interval”	34
3.5.5	Third gateway – message Type	35
3.5.6	Task “Get order response values”	35
3.5.7	Task “Forward order response”	35
3.5.8	Task “Mark message in database as received”	36
3.5.9	Processing a shipment notification	36
3.6	Process “Respond to Order”	37
3.6.1	Process variables	37
3.6.2	Start event “Order received”	37
3.6.3	Creating and sending an order response	37
3.6.4	Creating and sending a shipment notification	38
3.7	Process “Send Order Response”	38
3.8	Process “Retrieve Orders”	38
4	Running the processes	40
4.1	Preparations on the customer side	40
4.2	Preparations on the supplier side	41
4.3	Placing an order	42
4.4	Receiving an order and responding to it	43
4.5	Receiving an order response	45
4.6	Confirming a shipment	47
4.7	Receiving a shipment order	47
4.8	A shipment order is received before the order response has been processed	48

Figures

Figure 1: BPMN conversation diagram.....	8
Figure 2: Choreography diagram of the scenario.....	8
Figure 3: The collaboration diagram shows the message exchange within the scenario	9
Figure 4: The processes of the two partners	10
Figure 5: Logical data model	11
Figure 6: Contents of the messages	11
Figure 7: The implemented customer processes	17
Figure 8: The implemented supplier processes	19
Figure 9: Defining the XSD for an order in Eclipse	20
Figure 10: The data structures for order responses and shipment notifications in the Eclipse XSD editor	21
Figure 11: Customer's database access information in the MySQL connector	22
Figure 12: Process "Place Order" (detail from Figure 7).....	22
Figure 13: Data in pool "Place order"	23
Figure 14: Defining an XML variable.....	23
Figure 15: Skip the initial process dialog	24
Figure 16: Groovy script connector for setting a unique order ID	24
Figure 17: Specifying parameters for the Groovy script connector	25
Figure 18: Assigning the id to the order's id field	26
Figure 19: Dialog "Create order"	26
Figure 20: Outgoing message of task „Send message“	27
Figure 21: The order message.....	27
Figure 22: The content of the order message.....	28
Figure 23: Event "Order response received“	28
Figure 24: Assigning the message contents to a variable.....	29
Figure 25: Dialog "Show order response".....	30
Figure 26: Sequence flow "Order confirmed"	30
Figure 27: Process „Send Order“ (detail from Figure 7)	30
Figure 28: Event „Order received for sending“	31
Figure 29: Setting the values for the database	32
Figure 30: Process „Send Order“ (detail from Figure 7)	33
Figure 31: Assigning the retrieved data to process variables	34
Figure 32: Timer condition.....	35
Figure 33: Assigning the message content to process variables.....	35
Figure 34: Defining the forwarded message for a received order response	36
Figure 35: Process „Respond to Order“ (detail from Figure 8)	37
Figure 36: Process „Send Order Response“ (detail from Figure 8)	38
Figure 37: Process „Retrieve Orders“ (detail from Figure 8)	38
Figure 38: The customer's processes in the portal	40
Figure 39: Starting the Retrieve Order Responses process (customer)	40
Figure 40: Just a start button for starting a new process instance (customer)	41
Figure 41: The single process instance in the administration view (customer).....	41
Figure 42: The supplier's processes.....	41

Figure 43: Starting the process “Retrieve Orders” (supplier)	42
Figure 44: The single process instance in the administration view (supplier)	42
Figure 45: Starting the process “Place Order” (customer)	42
Figure 46: Creating an order (customer)	42
Figure 47: Creating another order (customer).....	43
Figure 48: Creating a third order (customer)	43
Figure 49: The order messages in the message queue (database)	43
Figure 50: The order messages have been marked as “received” by the supplier’s “Retrieve Order” process (database)	44
Figure 51: “Create order response” tasks in the inbox (supplier).....	44
Figure 52: Creating an order response (supplier)	44
Figure 53: The order response in the message queue (database)	44
Figure 54: The order response message has been marked as “received” by the customer’s “Retrieve Order Response” process (database)	45
Figure 55: New task “Confirm shipment” in the inbox (supplier)	45
Figure 56: Creating another order response (supplier)	45
Figure 57: Since the order has been rejected, no shipment confirmation task has been created (supplier).....	45
Figure 58: “Show order response” tasks in the inbox (customer)	46
Figure 59: Inspecting the first order response (customer).....	46
Figure 60: Inspecting the second order response (customer)	46
Figure 61: Current state of process instances in the administration view (customer)	47
Figure 62: Confirming a shipment (supplier)	47
Figure 63: Task “Show shipment notification” in the inbox (customer)	47
Figure 64: Inspecting the shipment notification (customer)	48
Figure 65: The last “Create order response” task in the inbox (supplier)	48
Figure 66: Confirming the last order (supplier).....	48
Figure 67: The “Confirm shipment” task for the last order (supplier)	49
Figure 68: Confirming the last order (supplier).....	49
Figure 69: Although a shipment notification has been received, there is still only the task “Show response” in the inbox (customer).....	49
Figure 70: Inspecting the last order response (customer).....	49
Figure 71: Now the shipment notification task is shown (customer)	50
Figure 72: Inspecting the last shipment notification task (customer).....	50

Tables

Table 1: Database columns	21
Table 2: Assigning the value of the order ID to a text variable.....	31
Table 3: Variables of the process „Retrieve Order Responses“	33
Table 4: Variables of the process „Retrieve Order Responses“	37
Table 5: Assigning the shipment date	38

Listings

Listing 1: XML Schema for an order	20
Listing 2: Creating the database table for the message queue.....	21
Listing 3: Groovy script for creating a unique order ID	25
Listing 4: Converting the order from a DOM object to an XML string.....	32
Listing 5: Wrting the order into the database.....	32
Listing 6: Retrieving order responses and shipment notifications from the database	33
Listing 7: SQL code for changing the message state to "received"	36

1 Introduction

1.1 The collaboration scenario

More and more companies are supporting their processes with Business Processes Management Systems (BPMS). Many processes interact with processes of other companies, e.g. customers or suppliers. If these interactions are to be automated, it is necessary to connect executable processes from different partners. Typically, this is achieved by exchanging messages between the processes.

BPMN (Business Process Model and Notation) provides different diagram types for modeling interactions between partners.

A conversation diagram gives an overview of the different partners and their communications. Figure 1 shows a communication “Order Products”. All BPMN diagrams in this paragraph have been created with the Signavio Process Editor (www.signavio.com).

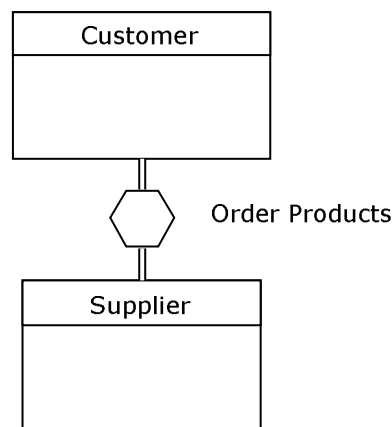


Figure 1: BPMN conversation diagram

This communication involves two partners: a customer and a supplier. More details of this communication can be seen in the choreography diagram in Figure 2. The entire scenario contains two choreography activities. The first one, “Send Order”, is initiated by the customer (shown in the white band) by sending a message “Order” to the supplier. In the same choreography activity, the supplier sends a message “Order response” back to the customer.

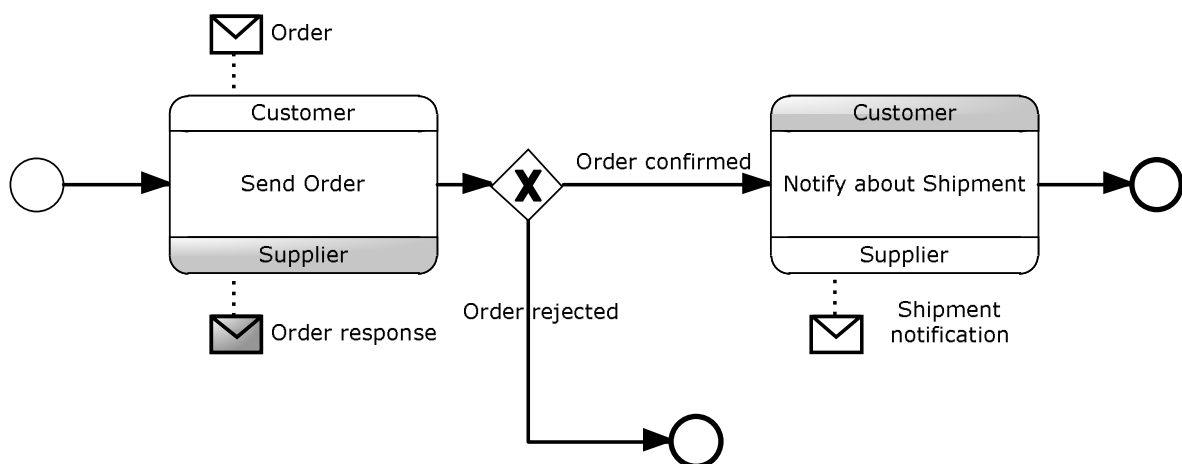


Figure 2: Choreography diagram of the scenario

If the order has been rejected, the scenario is finished. Otherwise, if the order is confirmed, the second choreography activity, “Notify about Shipment” is carried out. It is initiated by the supplier who sends a shipment notification to the customer. In this choreography activity, no message is sent back.

The customer is informed about the supplier’s decision by the order response message. Therefore he also knows if he needs to wait for a shipment notification, or if the scenario is finished.

We can also draw this scenario as a collaboration diagram with black box pools, i.e. the participants’ processes are hidden (Figure 3). Here, we can see the exchanged messages. However, we don’t see that a shipment notification is not always sent.

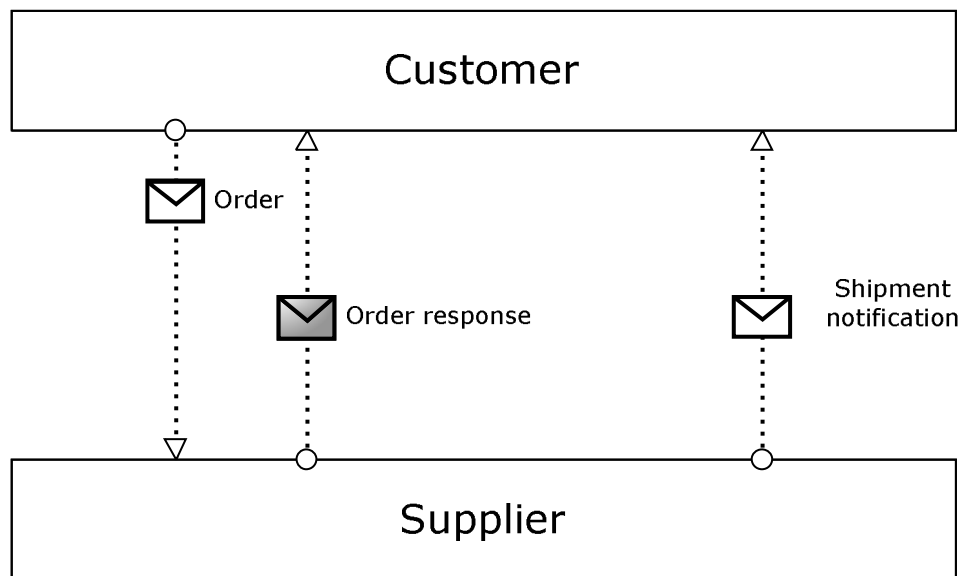


Figure 3: The collaboration diagram shows the message exchange within the scenario

Figure 4 shows the same collaboration, but with the partners’ processes displayed within the pools. After starting his process “Place Order”, the customer creates an order. “Create order” is a user task, i.e. it is carried out by a user of the BPMS. He enters the order details into a form and submits it to the BPMS which proceeds to the next task “Send order”. In this task a message containing the order information is sent to the supplier. After that, the process waits at the intermediate event for the reception of an order response.

On the supplier’s side, the arrival of a message “Order” triggers the process “Respond to order”. The first task “Create order response” is also carried out by a user. The order response is then sent back to the customer. If the response is negative, i.e. the order is rejected, the supplier’s process is finished. Otherwise, it proceeds to the task “Confirm shipment”.

Of course, the shipment can only be confirmed when the ordered goods actually have been shipped. We could have indicated this condition by a catching intermediate event, i.e. a condition event or a message event which waits for a message from the warehouse. However, since “Confirm shipment” is a user task, such an event is not required. A user task is only carried out when the user selects it from his task list. Of course, the user will do this only when he knows that the goods have been shipped.

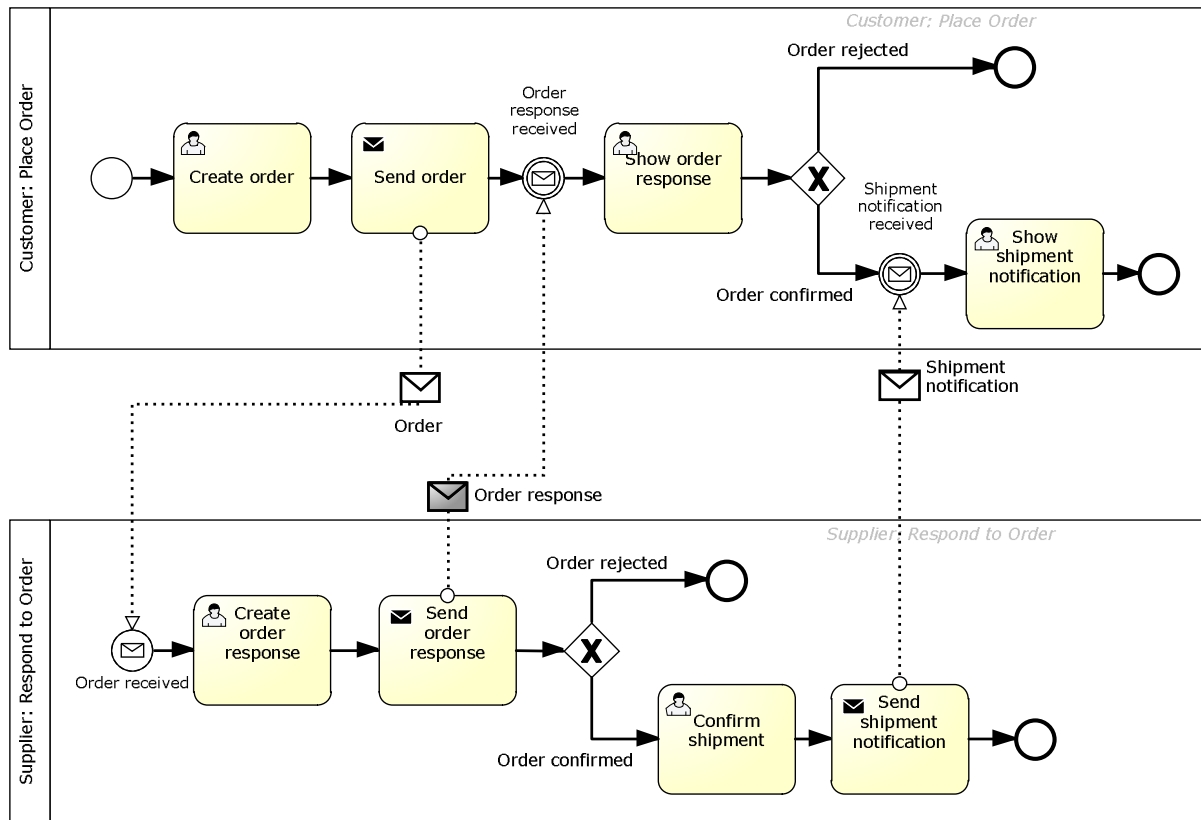


Figure 4: The processes of the two partners

When the order response has been received, the customer's process contains with "Show order response" in which the user inspects the received order response. If the order has been rejected, the customer process is also finished.

If the order has been confirmed, the customer process waits for the arrival of a shipment notification. This notification is created by the supplier in the user task "Confirm shipment" and finally sent to the customer process. Here, the shipment notification is inspected in the task "Show shipment notification" before this process finishes, too.

1.2 Data structures and messages

The data structure in this example is rather simple. It is shown in Figure 5 as a UML class diagram (created with Modelio, www.modeliosoft.com). Every order has a unique ID (identifier), and it contains the product to be ordered and the ordered amount of this product. There may or may not be an order response to an order. At the beginning of the above process, the order won't have an order response, since the supplier has to create one yet. Every order response is related to exactly one order. The response attribute is used for confirming or rejecting the order (value "yes" or "no").

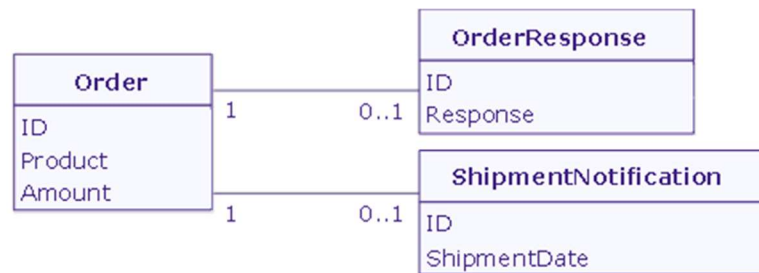


Figure 5: Logical data model

Every shipment notification also refers to exactly one order, and there may not be more than one shipment notification for any order. A shipment notification will only be created by the supplier when the order response's "response" attribute has the value "yes". A shipment notification has an attribute with the date on which the ordered product has been shipped. Both order response and shipment notification have their own unique IDs which are not identical to the referenced order's ID.

We don't implement this data structure within one system, but we need to exchange the different data objects by messages. Therefore we can't implement direct links, but we need to use attributes for the referenced order's ID. It is also useful to copy the other attributes of the order into the order response and into the shipping notification. This information can help detecting errors. For example, if the order IDs get mixed up in the supplier's system, the customer could detect this. In this case, the product and the amount of the order confirmation will be different from those provided by the customer in the referenced order. Besides this, an electronic order confirmation can also serve as a legally binding document. Therefore it should state what is actually being confirmed.



Figure 6: Contents of the messages

The messages' attributes are shown in Figure 6. Order response and shipment notification both have an attribute "OrderIDRef". This attribute contains the value of the referenced order's ID. We haven't defined the data types of the different attributes yet. Since messages may be processed by different kinds of systems, it is often advisable to use only strings in messages. Almost every other data type can be converted into a string.

This paper explains how to implement the above scenario with a BPMS. We have used the open source BPMS "Bonita Open Solution" (www.bonitasoft.com).

The general principles and methods explained in this paper are also valid for other BPMS, so that it can also be useful for readers who do not use Bonita Open Solution.

1.3 Implementation alternatives

How can we implement the message exchanges between the two processes?

It would be rather easy if we had to connect two processes that run in the same process engine. In this case, the process engine could forward the message between the processes. The Bonita engine, for example, can execute the two processes just as they are modeled in Figure 4.

However, customers and suppliers are independent of each other. They will use their own systems. Therefore, the two processes shown in Figure 4 will be deployed on separate process engines. It should also be possible to replace the Bonita-based implementation of one of the partners by another installation, e.g. a different BPMS or an individually developed software without a BPMS.

How can we implement the message flows between two independent processes running on different system?

There are different alternatives, such as:

1. **Directly connecting the two systems via proprietary interfaces.** Bonita, for example, provides an API that could be used by a partner for retrieving the required process instance and sending a message to it. However, this would require quite some coding, and the integration would only work with Bonita, and it could not be re-used for other partners that use a different system. Since every company has many partners, such an approach would require the development of individual connections for each partner.
2. **Directly connecting the two systems via standard interfaces, such as web services.** In fact, this is a common integration method. Some BPMS make heavy use of web service standards. With these systems it is very easy to integrate web service calls into processes. Based on the process definition standard BPEL (business process execution language), the interface of a process itself can be published as a Web Service. It is therefore very easy to integrate the call of another process into a process model. If you want to evaluate such a system, you may try the free version of Intalio|BPMS. It can be downloaded at community.intalio.com. Bonita, on the other hand, doesn't provide much web service support in version 5.5. A direct connection also has a disadvantage: The partner's system must be up and running, and it must be possible to reach it via the network at the time a message is sent. Web services require synchronous communication.
3. **Using a message queue (MQ).** Other than the direct connection, a message queue allows for asynchronous communication. It is based on the idea that each sender puts his messages into a queue. The receiver then can retrieve and process the messages that are addressed to him. This method is also working when the two partners are not online at the same time. The messages are stored in the queue, and the receiver can also retrieve them at a later time. There are special message queue systems available. A message queue can also be part of a middleware, such as an enterprise service bus (ESB).
In our small case it is easier to implement our own simple message queue. We can just define a table in a database as an intermediate storage for the exchanged messages. This will be explained in detail later on.
This approach is rather technology-independent. Older systems, for example, often do not support web service standards. Most middleware products provide connectors to different

systems and technologies. And for our suggested simple database implementation it is only necessary that the connected systems can access a relational database. Thus, it would be possible to replace one of the two Bonita processes with an entirely different system that sends and receives the same messages as defined in the choreography (Figure 2).

4. **Exchanging messages via e-mails.** The sender generates an e-mail containing the message contents in a pre-defined format that can be processed by the receiver's system without any human interaction. This is similar to the message queue. Each partner's mail inbox provides a queue for the incoming messages. So instead of one central message queue, there are several queues, one for each partner. With a central message queue it must be decided who host this queue: Either one of the partners, or a third party. Of course, separate incoming message queues do not require e-mail exchange, but they could also be implemented with another technology (such as a MQ system).

The exchange of e-mails would be another good option for connecting our two Bonita-based processes.

In this paper we will show how to connect the two processes via a simple message queue, implemented as a database table.

1.4 Necessary agreements between partners in a business collaboration

Our example shows that a business collaboration requires the involved partners to agree on several aspects, before a successful integration of the two processes can be achieved.

These aspects include:

- The choreography
- The contents of the exchanged messages
- The technical implementation, including standards (such as XML, web services etc.), exchange platforms (common infrastructure such as a message queue), technical interface descriptions, addressing, protocols, networks, security measures (such as encrypting the message), reliability (e.g. re-sending messages that have not been confirmed), etc.
- The implementation of the messages, e.g. by defining a common XML schema for the contents (see paragraph 3.1), but also for the meta data. In our example, the required meta data (message ID, message type, message state) are defined as columns in the database table (see paragraph 3.2)
- If a common infrastructure is used, it needs to be defined who is hosting this platform (one of the partners or a third party).
- Service levels, such as reaction times or availability of each partner's system.
- Responsibilities of each partner and contact persons. For example, if a problem occurs that cannot be handled by the implemented processes themselves (e.g. by defined exception flows), it needs to be defined how this will be handled.
- Distribution of costs, e.g. for using a hosted exchange platform or a cloud service.

1.5 What is not covered in this example

In order to keep the example easy to understand, it is extremely simplified compared to a real world business collaboration. Several aspects that are important in real implementations have not been covered, such as:

- Further variations in the processes, such as splitting an order or changing an order, the inclusion of several roles, etc.
- Extensibility towards more than two partners.
- More complex data structures.
- Service levels.
- Security aspects.
- Validation of user inputs, received messages etc.
- Exception handling. In practice, it would be necessary to handle exceptions in the process, e.g. if a message cannot be delivered, or if an expected message does not arrive within a certain time.

1.6 Download and installation

Bonita Open Solution can be downloaded for free from www.bonitasoft.com. We have used version 5.4, but our processes also work in version 5.5.

The processes developed in this paper can be downloaded from <http://www.kurze-prozesse.de/implementing-a-business-collaboration-with-bpmn-and-bonita/>

Ideally, the customer's processes and the supplier's processes are executed on different computers in a network. However, it is also possible to run both processes in one single Bonita installation. The user then receives both the tasks of the customer and those of the supplier in one inbox.

Prerequisites:

You need to set up a MySQL database and create a table for the messages to be exchanged. This is explained in chapter 3.2. You can also use another database system, if you replace the MySQL connectors in the processes by the connectors for your database system.

Installation:

- Uncheck "Drop database on startup" in Bonita Studio's preferences dialog (in the "Edit" menu). Further explanations for this can be found in paragraph 3.3.2.
- Import the downloaded bar-files into Bonita studio. "Place_Order-1.0.bar" has to be imported into the customer's Bonita installation, "Respond_to_Order-1.0.bar" into the supplier's installation.
- Enter the access data of your database into the database connectors. This is also explained in chapter 3.2.
- Deploy the processes.
- The only user is "admin" (password: "bpm")
- Start the processes "Retrieve order responses" (customer) and "Retrieve orders" (supplier). These two processes must be started exactly once, at the beginning.

- Now the customer can create orders by starting the process “Place order”.
- Wait for 10 seconds and refresh the view in order to see the tasks for received orders, etc. in the inbox.

Known problems:

The processes “Retrieve order responses” and “Retrieve orders” contain loops that are repeated every 10 seconds, i.e. they run endlessly. In order to avoid problems when re-deploying these processes, the process instances of these processes should be deleted by the administrator before closing Bonita. In some cases we had problems deleting these instances and we had to re-start the server before we could delete them.

2 The processes in Bonita

In this chapter, we will shortly explain the processes that have been modeled and implemented in Bonita. The various details, such as data structures, user interfaces, etc. will be explained in the next chapter.

2.1 Customer processes

Three processes have been implemented for the customer. They are shown in Figure 7. The model of the “Place Order” business process itself is identical to the model that has been developed during analysis (Figure 4). Since this process cannot communicate directly with the supplier, the supplier’s process is substituted by two utility processes: “Send order” receives an order message and writes its contents into the database. “Retrieve Order Responses” regularly queries the database for new messages. If it finds a message of the type “order response” or “shipment notification”, it forwards it to the “Place Order” process.

All technical details are handled by the two utility processes which are shown with a grey background. This has the advantage that the actual business process (with a yellow background) remains untouched by these implementation aspects. If the implementation of the message exchange was changed, the “Place Order” process wouldn’t be affected. We could, for example, change our message queue for e-mail or web service-based communication, and handle this entirely by the two utility processes. These two processes do not contain any user tasks, i.e. they are completely automated.

The “Send Order” process is rather simple. It is triggered when an order message is received from the “Place Order” process. The first task converts the contents of an order message into the format that is required for storing the order in the database. The second task then writes this message into the database.

The “Retrieve Order” process looks for new messages in the database. If it find messages of the type “order response” or “shipment notification”, it forwards them to the to the corresponding intermediate message event of the “Place Order” process. The “Retrieve Order” process must be started only once. The single process instance then regularly searches the database for new messages. If a new “order response” message has been found, the process continues with the task “Get order response values” which copies the contents into another format, before the task “Forward order response” sends the message to the “Place Order” process. The same things happen to a “shipment notification” message in the alternative path.

For both message types, the final task marks the retrieved message in the database as “received”, so that it won’t be retrieved a second time. After that, the process loops back to the beginning, so that the next new message can be retrieved. If the task “Read response message from database” doesn’t find a new message, the process continues with the timer event “Wait for polling time”. After waiting for the specified time, the process again looks for new messages. In practice, a waiting time of several minutes may be appropriate. For testing purposes we have set it to 10 seconds.

Note that this process doesn’t have an end event. It will continue looping until the process instance is deleted by an administrator.

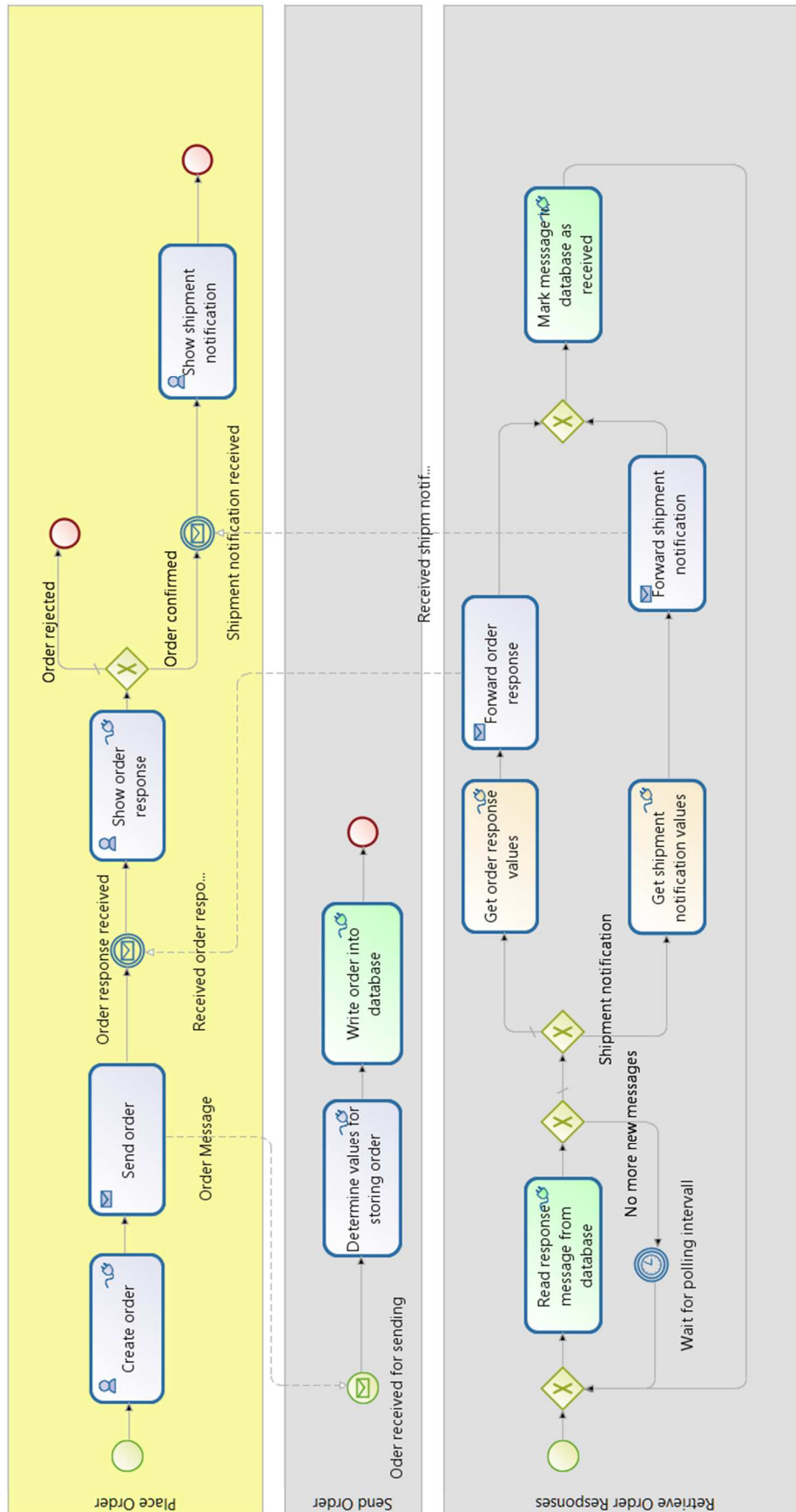


Figure 7: The implemented customer processes

2.2 Supplier processes

The supplier's processes are shown in Figure 8. Similar to the customer, the supplier has the actual business process (yellow background) and two utility processes for sending and retrieving messages.

The business process "Respond to Order" is again very similar to the analysis model from Figure 4. The only difference is a timer event after the decision, and an additional task "Initialize shipment notification". This task initializes the shipment notification by copying some values from the order response.

The timer event is necessary to make sure that "Confirm shipment" actually appears in the user's inbox. We have only one user in the example (the user "admin"). When two subsequent user tasks are performed by the same user, Bonita doesn't put the second task into the inbox, but it just starts the second task when the first task is finished. This is more convenient for the user, because he just can proceed with the next task without having to select it and to start it manually. However, in the case of the shipment notification, this task is usually carried out at a later time – when the shipment has actually taken place. Therefore we have inserted a little delay of 5 seconds, so that Bonita doesn't show the next task form, but it puts the task into the inbox after 5 seconds. The actual time between the order response and of the shipment notification may be much longer. The user just leaves the "Confirm shipment" task in his inbox until the shipment has actually taken place.

The other two processes are very similar the customer's utility process. The only significant difference is that "Send Order Response" needs to distinguish between the two message types "Order response" and "Shipment notification", while "Retrieve Orders" only needs to handle one message type ("Order").

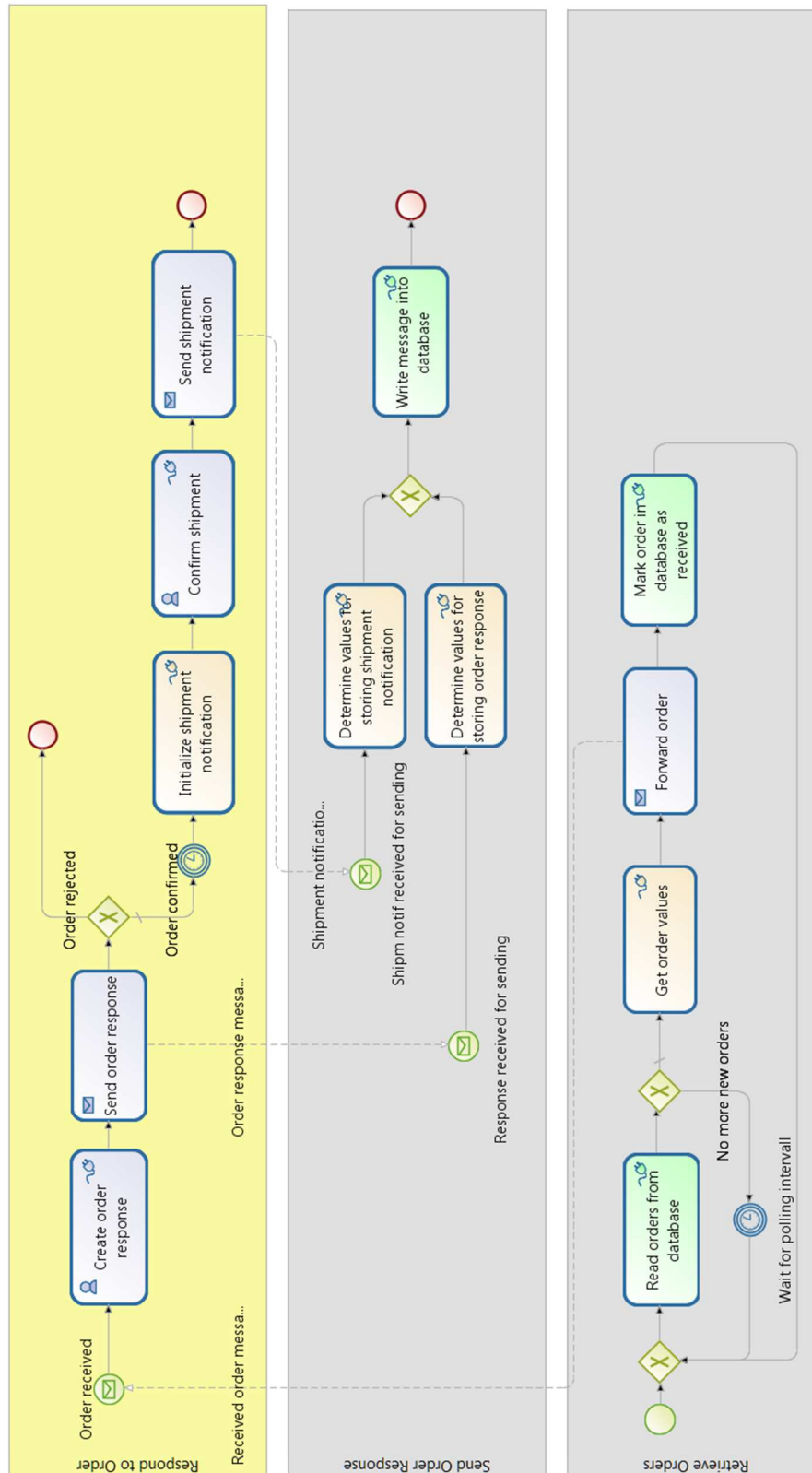


Figure 8: The implemented supplier processes

3 The implementation in detail

3.1 XML schema for messages

In chapter 1.2 we have described the contents of the messages to be exchanged. We use XML as a common standard for structuring messages. The data structures for XML files can be defined with XML Schema Definition (XSD). In order to use XML data structures in Bonita, you need to import an XSD file.

We have used the open source development tool Eclipse (with the Eclipse XML Editors and Tools plugin) for developing XSDs. Eclipse provides a graphical editor for XSDs. In Figure 9, the creation of an XSD of an order is shown. Here we have defined one element “Order” with the type “Order”. This “Order” type is shown in the right column of the Schema editor. When you click on this type, you get a view of this type (on the right). In our case, it contains the attributes “ID”, “Product” and “Amount”. This is the structure of an order message as defined in Figure 6.

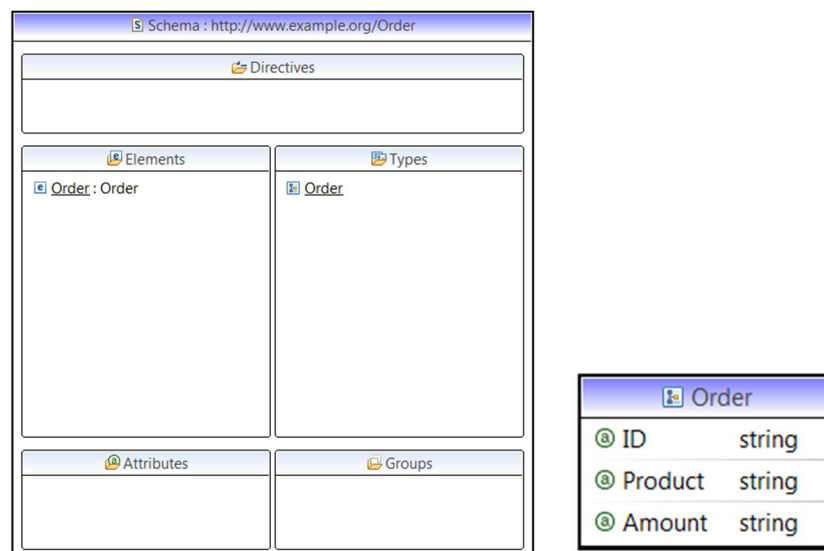


Figure 9: Defining the XSD for an order in Eclipse

We can also have a look at the actual XML code of this schema definition (Listing 1):

```
<?xml version="1.0" encoding="UTF-8"?>

<schema xmlns=http://www.w3.org/2001/XMLSchema
  targetNamespace=http://www.example.org/Order
  xmlns:tns="http://www.example.org/Order"
  elementFormDefault="qualified">

  <complexType name="Order">
    <attribute name="ID" type="string"></attribute>
    <attribute name="Product" type="string"></attribute>
    <attribute name="Amount" type="string"></attribute>
  </complexType>

  <element name="Order" type="tns:Order"></element>
</schema>
```

Listing 1: XML Schema for an order

The schemas for the other message types – order response and shipment notification – are defined in the same way. In Figure 10 the data type definitions of the two XSDs can be seen.

OrderResponse		ShipmentNotification	
a ID	string	a ID	string
a OrderIDRef	string	a OrderIDRef	string
a Product	string	a Product	string
a Amount	string	a Amount	string
a Response	string	a ShipmentDate	string

Figure 10: The data structures for order responses and shipment notifications in the Eclipse XSD editor

3.2 Preparing the database

We use a relational database for implementing our simple message queue. We only define one table for storing the messages. Since we want to use this table for any kind of message, we don't define any columns that are specific to a certain message type (such as product or shipment date). We only use a few columns for meta information, such as the message ID or the message type, and one column for the entire message content as an XML-formatted string (conforming to the schema defined in the previous paragraph). Here is the list of columns:

Column	Data Type	Size (characters)	Description
msg_id	varchar	40	Unique ID of the message
msg_type	varchar	40	In our scenario we have the types "order", "order response", "shipment notification"
msg_state	varchar	40	"open" for a new message, "received" for messages that have already been collected by the receiver
msg_content	varchar	5000	The actual message content in one string (XML, as defined in paragraph 3.1)

Table 1: Database columns

We use a MySQL 5.0 database. The database is called "mq4bonita".

The table can be created with the following SQL statement:

```
create table csmmessage(
    msg_id varchar(40),
    msg_type varchar(40),
    msg_state varchar(40),
    msg_content varchar(5000),
    primary key (msg_id)
)
```

Listing 2: Creating the database table for the message queue

This SQL statement can be entered directly in a database administration tool, such as phpMyAdmin.

Where should the database server be located? Customer and supplier must both access the same database in order to use it as a common message queue. Therefore, one of them can host them for both. As an alternative, a third party could host the database. Such a neutral platform could be also be used as an exchange for many senders and receivers. However, our message queue is only

designed for one customer and one supplier, since we do not store any information that could be used for selecting the correct receiver.

In our case, the customer is hosting the database. This means that the customer can use the standard access information in the MySQL connector (Figure 11). This information needs to be entered into the connectors of all tasks that access the database. They are colored green in Figure 7.

Figure 11: Customer's database access information in the MySQL connector

In the supplier's processes, "localhost" needs to be replaced by the customer's server name. The customer's database installation must allow for remote root access, and his firewall must allow the access to port 3306 (if a standard installation of MySQL is used). We have used the user "root" in all connectors, i.e. you only need to insert the root password in the MySQL connectors. Of course, you can also use different users that have read and write permission for the table "csmessage".

3.3 Process "Place Order"

3.3.1 Data in the Pool "Place Order"

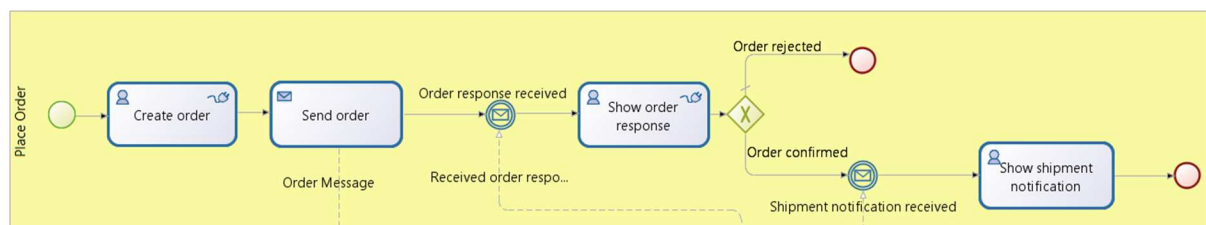


Figure 12: Process "Place Order" (detail from Figure 7)

Figure 12 shows the pool "Place Order" with its process. The variables defined in this pool are listed in Figure 13.

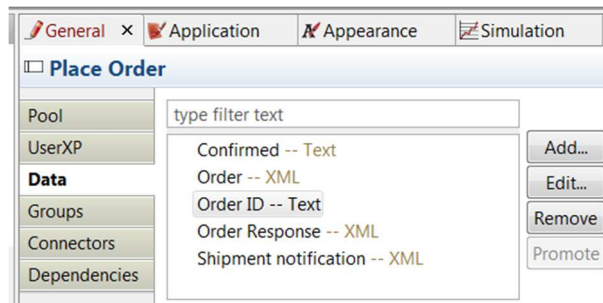


Figure 13: Data in pool "Place order"

We have two variables of type "Text". "Confirmed" is used for storing the suppliers answer to an order ("yes" or "no"). This information is required for the decision after "Show order response". "Order ID" stores the order's unique identifier. It is needed as a correlation key, i.e. for assigning the right order response messages to the right "Place Order" process instance. The information in these two variables is also contained in the XML data structures "Order" and "Order Response", but in many cases it's easier in Bonita to access a simple text variable than a part of a complex XML structure.

Sometimes complex data structures are required, i.e. data structures that contain many fields, lists etc. In Bonita, there are two ways for defining such complex variables. You can use Java objects or XML data structures. Since we have already defined XML schemas for our messages, we can use these schemas here, as well. We have defined three XML variables, one for each message type.

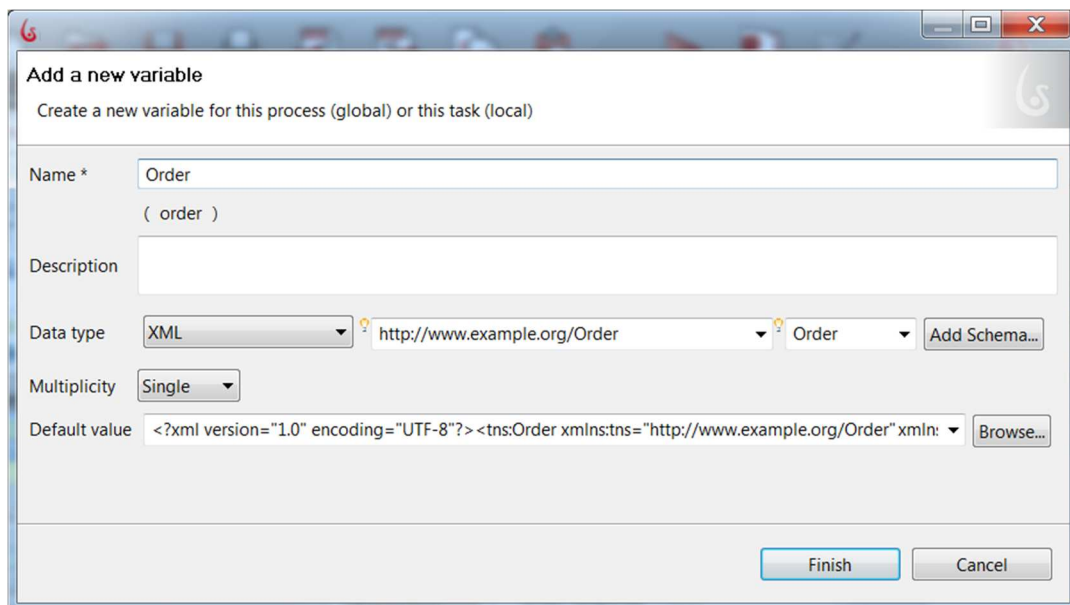


Figure 14: Defining an XML variable

In Figure 14, it can be seen how to define the XML variable "Order". The data type is "XML". With "Add Schema" we can import the XSD-File we have created in paragraph 3.1. After the schema has been imported, we can select the namespace and the element we want to use. We also need to provide a default value. This is necessary, because otherwise Bonita won't create the variable. In Eclipse we can create an XML file from an XSD and enter some sample data. This XML file can be imported via the "Browse..." button. You can also just copy the XML text into the field "Default value".

The other two variables, “Order response” and “Shipment notification” are created in the same way.

Note that we have selected to skip the initial process dialog (Figure 15). It is not possible to access an XML variable in the initial process dialog, because the process instance is only initialized after this dialog is completed. Since an XML variable needs to be initialized before it can be used, it does not yet exist in the initial process dialog.

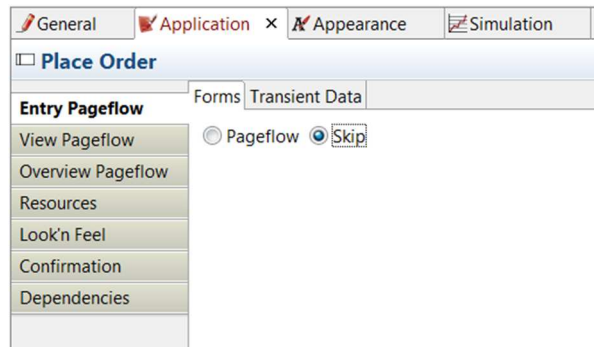


Figure 15: Skip the initial process dialog

3.3.2 Creating a unique order ID

Each order needs to have a unique order ID. Since the values of normal variables are only valid within a single process instance, we need to implement a counter that is outside of the processes and that can be accessed by all process instances.

Where can we implement such a counter? Since we already use a database, one possibility would be to store the counter in that database.

Here, we use another mechanism. Bonita provides a concept called “MetaData” that can be used for our purposes, since these meta data are independent from process instances. In order to read and write the counter value, we add a Groovy script connector to the pool “Place Order” (Figure 16).

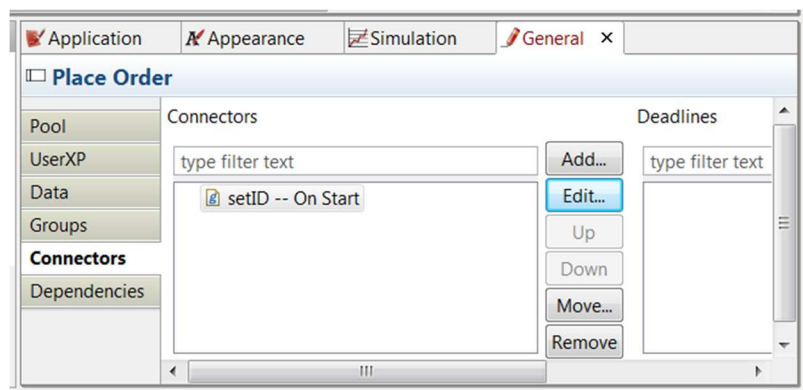


Figure 16: Groovy script connector for setting a unique order ID

Since the order ID has to be defined at the beginning of the process, we need to select the event “enter” for this connector (Figure 17).

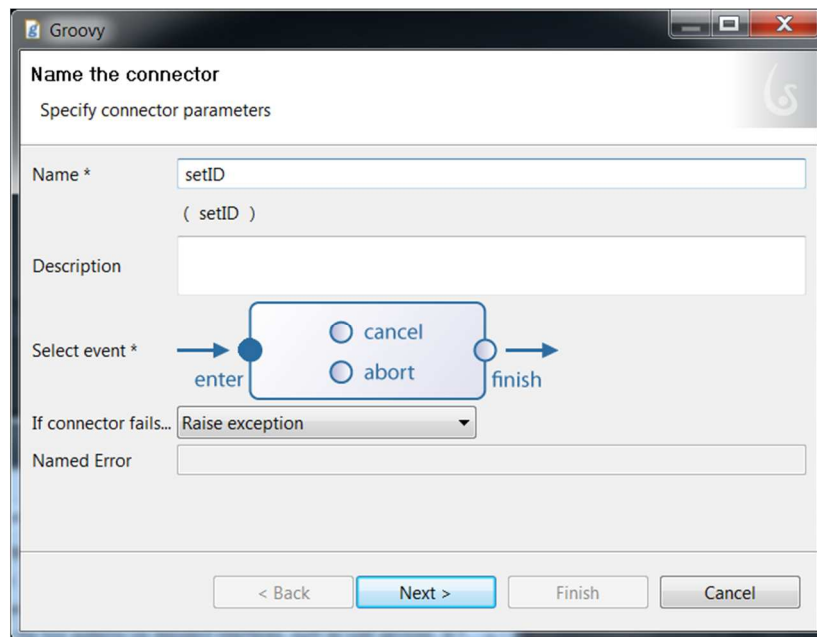


Figure 17: Specifying parameters for the Groovy script connector

```
import org.ow2.bonita.util.AccessorUtil;
import org.ow2.bonita.facade.ManagementAPI;

// Try to access a MetaData "idCounter"
String idCounter = AccessorUtil.getManagementAPI().getMetaData("idCounter");

// If it doesn't exist, add it and initialize it, else increment it
if(idCounter==null){
    AccessorUtil.getManagementAPI().addMetaData("idCounter", "1");
    idCounter = "1";
} else {
    int i = Integer.valueOf(idCounter);
    i++;
    idCounter = i.toString();
    AccessorUtil.getManagementAPI().addMetaData("idCounter", idCounter);
}
String idString = "C-" + idCounter
idString
```

Listing 3: Groovy script for creating a unique order ID

Listing 3 shows the Groovy script that creates the ID. It tries to read a MetaData called “idCounter”. If this counter doesn’t exist, it is created and its value is set to “1”. Otherwise the value is read and incremented by 1, and the new value is written back.

The script then creates an idString to which it adds a prefix “C-” (for “customer”). This prefix makes sure that the customer’s and the supplier’s IDs are different. The supplier’s IDs start with “S-”.

Finally, the idString is returned. In Figure 18, the result is assigned to the field “ID” of the XML variable “Order”.

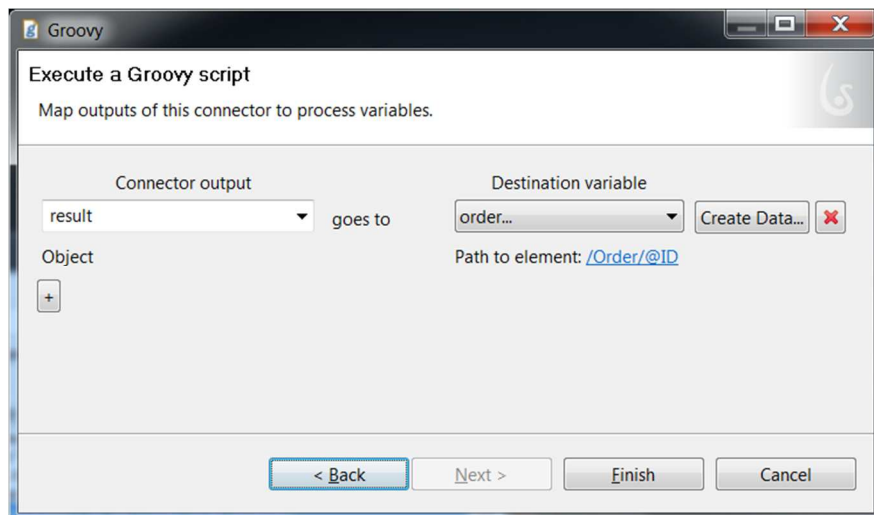


Figure 18: Assigning the id to the order's id field

Note that when you want to use the process with the above script you need to uncheck “Drop database on startup” in Bonita Studio’s preferences dialog (in the “Edit” menu). You find that checkbox by selecting “Bonita” and then “User Experience”. If this option remains active, the counter will be reset to “1” after every restart of Bonita. In this case our message queue database won’t store new messages, because the message IDs need to be unique.

3.3.3 Task “Create order”

“Create order” is a user task. The actor is the process initiator. Generally, the user tasks in our processes have the user “admin” as the actor. Therefore, for all processes, the user “admin” should be logged in the User XP portal.

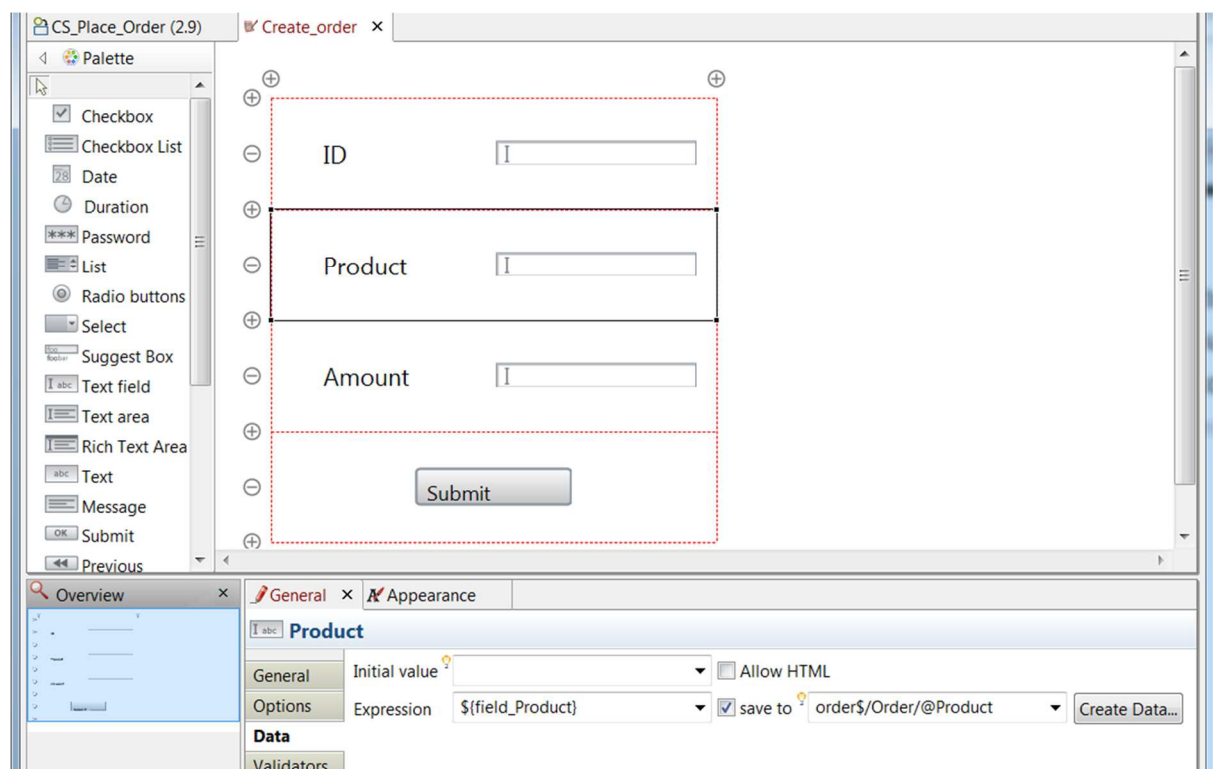


Figure 19: Dialog "Create order"

Figure 19 shows the dialog for this user task. The “ID” field is read-only. Its initial value is the order ID. “Product” and “Amount” have to be entered by the user. The values are stored in the respective field of the “Order” variable. This assignment is shown for the “Product” field.

This task also has a “Set variable” connector assigned that is used for copying the value of the order ID into the text variable “Order ID”. This variable is used later on for correlating the right response messages to the right process instance.

3.3.4 Task “Send order”

“Send order” is a send task. It has an outgoing message, called „Order_Message“ (Figure 20).

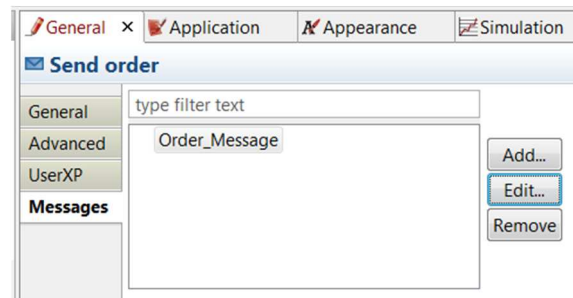


Figure 20: Outgoing message of task „Send message“

For this message we need to specify the target pool and the target task. The target task can also be an event. Since our target pool is in the same diagram, a message flow is shown from “Send order” to the event “Order received for sending” in the pool “Send order” (see Figure 7).

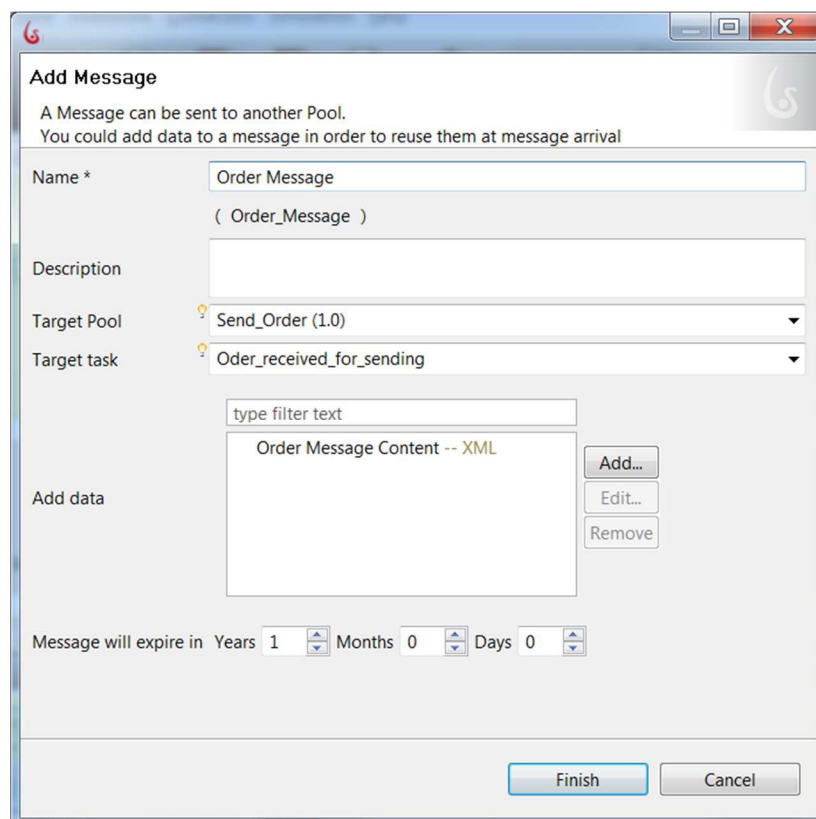


Figure 21: The order message

We also need to create a variable for carrying the message content. The specification of this variable is shown in Figure 22. We use the XML schema for the order that we have imported earlier. The default value is the XML variable “Order”. Thus, the content of the order is copied into the message variable.

Figure 22: The content of the order message

3.3.5 Intermediate message event “Order response received”

At this event, the process instance waits until the expected order response has been received. We define that this message catches the message “Received order response message” which is sent by the task “Forward order response” in the process “Retrieve order responses”. Note that you can select this message only after it has been defined in the “Forward order response” task (see paragraph 3.5.7).

Figure 23: Event “Order response received”

Several order responses may arrive at any time. Every order response refers to a certain order. We need to make sure that every process instance only catches that order response message that refers exactly to the order that has been sent by this instance. For this purpose we must compare the process instance’s order ID with the order ID that is referenced in the order response message. This is done by the expression “order_ID.equals(order_resp_order_ref_id)” in the field “Matching

condition”. Order responses with order reference IDs different from the order ID will be ignored by the process instance (and caught by another instance with the corresponding order ID).

After catching the message, we need to assign its contents to the process variable “Order Response”. This is done by a “Set Variable” connector. The value of “order_response_msg_content” can directly be assigned to “Order Reponse”, because both are XML variables with the same definition. “order_response_msg_content” is defined in the task “Forward order response” (see paragraph 3.5.7).

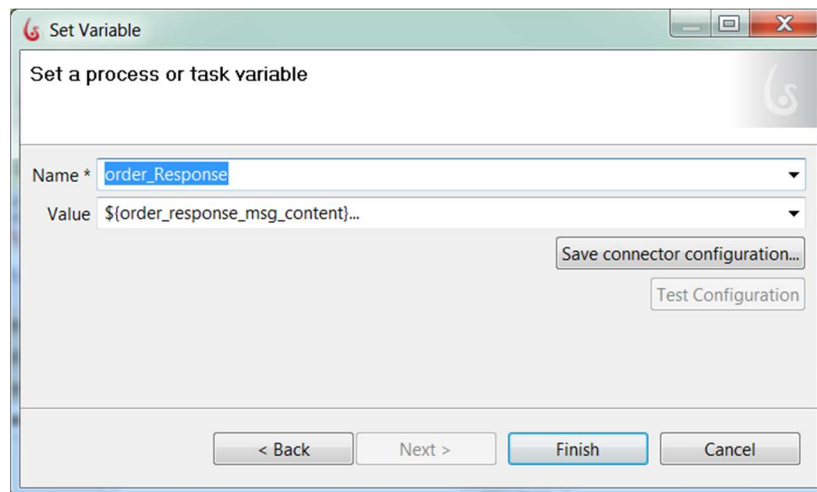


Figure 24: Assigning the message contents to a variable

3.3.6 The remainder of the process

In the user task “Show order response”, the contents of the original order and of the received order response are shown next to each order so that they can be compared (Figure 25).

This task also contains a “Set variable” connector that copies the value of the order response’s “response”-field to the text variable “confirmed”. This variable is used in the subsequent gateway.

The exclusive gateway has two outgoing sequence flows. The one labeled “Order rejected” is marked as a default flow. If the condition of the other sequence flow is not true, then this exit will be taken, and the process finishes with the end event.

The other sequence flow, “Order confirmed”, has a condition (Figure 26). It will be taken when the variable “Confirmed” contains the value “yes”.

The subsequent intermediate event, “Shipment notification received”, is another catching message event. Here, the process instance waits until the right shipment notification is received. This is very similar to the event “Order response received”. Therefore we don’t need to explain the details of this event.

The user task “Show shipment notification” shows both the original order and the shipment notification so that they can be compared. This task is similar to “Show order response”.

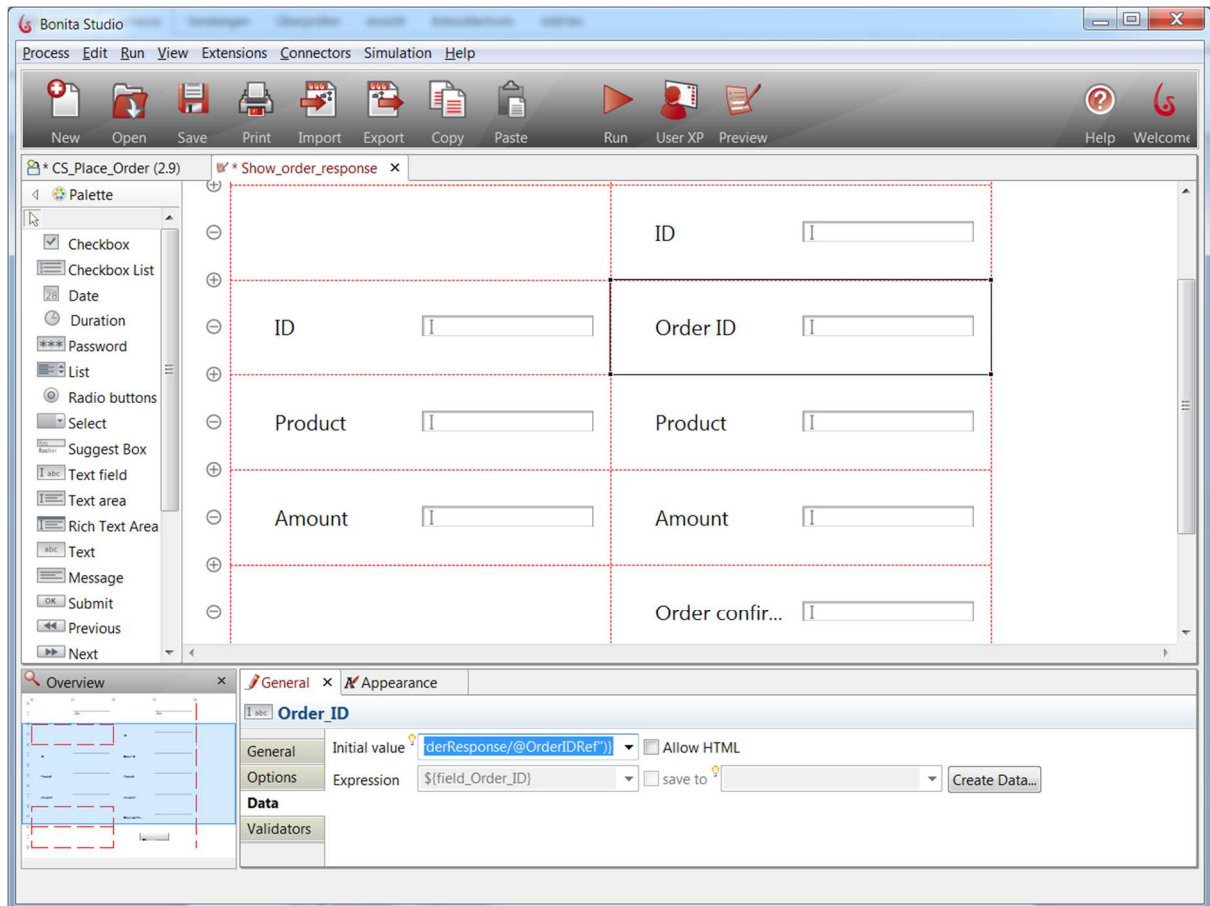


Figure 25: Dialog "Show order response"

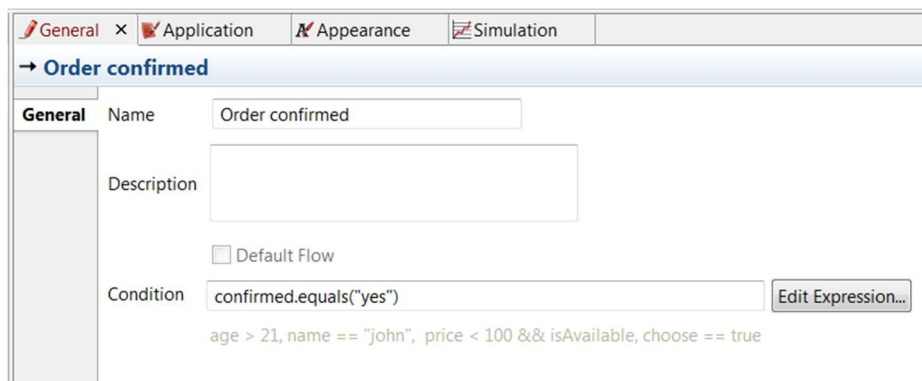


Figure 26: Sequence flow "Order confirmed"

3.4 Process "Send Order"

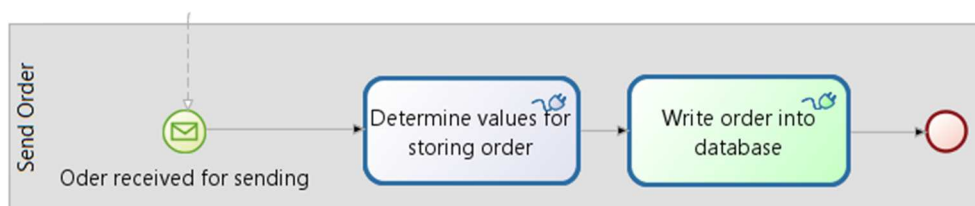


Figure 27: Process „Send Order“ (detail from Figure 7)

3.4.1 Variables in the process “Send Order”

This process has three variables: An XML variable “Order” and two text variables “ID” and “MsgContent”. “Order” is again based on the Order XSD. It is used for storing the contents of the received order message. The text variables are used for copying the ID and the message contents into the SQL statement for writing the message into the database.

3.4.2 Message start event “Order received for sending”

The process “Send Order” is triggered by the message start event “Order received for sending”. It catches an “Order Message” that is sent by the “Send order” task in the process “Place Order”. Since for every received order message a new instance of the “Send Order” process has to be started, no matching condition is required (Figure 28).

The screenshot shows the configuration for a message start event named "Order received for sending". The "General" tab is active. The "Name" field contains "Order received for sending" with a suffix "(Oder_received_for_sending)". The "Description" field is empty. The "Message Type" is set to "Start Message". The "Catch Event" is set to "Order_Message". The "Matching condition" field is empty, and there is an "Edit Expression..." button next to it.

Figure 28: Event „Order received for sending“

This event also has a “Set Variable” connector that copies the message content into the process variable “Order”.

3.4.3 Task “Determine values for storing order”

In this task we prepare the values that we want to write into the database. As described in paragraph 3.2, the database table has 4 columns: message ID, message type, message state and message content. In this process, the only message type is “order”. Since we write new messages in the database, the state is always “open”. Therefore we only need to extract the order ID and the message content from the XML variable “Order”, and to store them in the process’s two text variables “ID” and “MsgContent”.

A “Set Values” connector is used for that (Figure 29). Unfortunately, in this connector the widget for selecting variables doesn’t work properly. The correct format for assigning the order’s ID field to the text variable ID is:

Name	Value
iD	<code>\${providedscripts.BonitaXML.evaluateXPathOnVariable(order, "/Order/@ID")}</code>

Table 2: Assigning the value of the order ID to a text variable

The second variable is a bit more complicated. As stated in paragraph 3.2, we want to write the entire message contents as an XML-formatted string into the database. The variable “Order” contains this XML structure, but not as text, but as a DOM object. DOM (document object model) is a non-textual representation of an XML structure that can easily be modified by a program. Therefore we need to convert the DOM structure into a text representation again. This is achieved by the code in Listing 4. This code has been entered as an expression in the “Value field” in the

second row in Figure 29. Note that quote signs (") are replaced by escaped quote signs (\"). This is necessary, because the resulting string is inserted into an SQL statement. The quote signs within the string must not be interpreted as SQL delimiters.

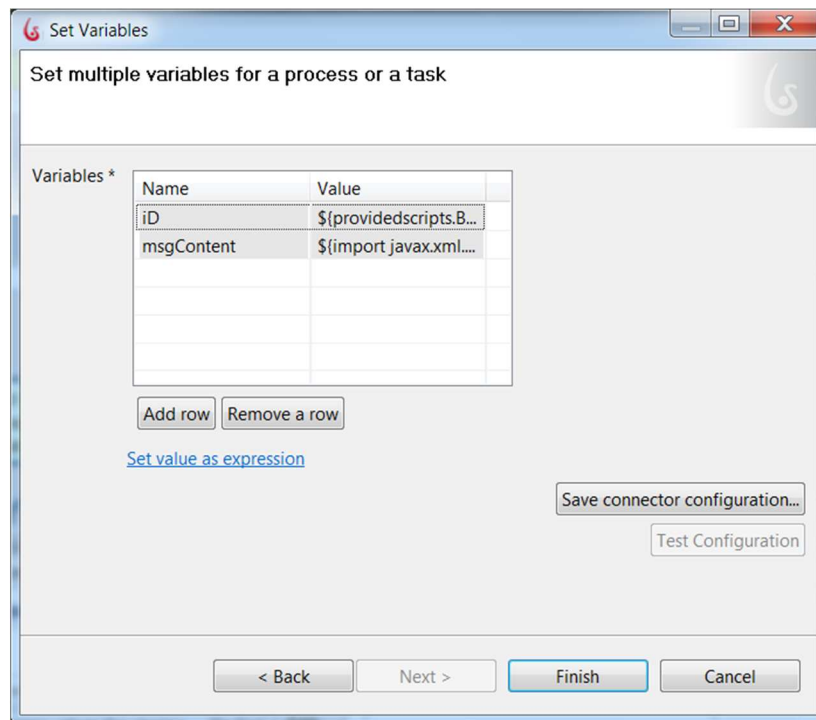


Figure 29: Setting the values for the database

```
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.Transformer;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import java.io.StringWriter;

String result = "empty"
try{
    TransformerFactory tf = TransformerFactory.newInstance();
    Transformer m = tf.newTransformer();
    DOMSource source = new DOMSource(order);
    StringWriter wr = new StringWriter();
    StreamResult target= new StreamResult(wr);
    m.transform(source, target);
    result = wr.toString();
    result = result.replace("\"", "\\\"");
} catch (Exception e){
}
result
```

Listing 4: Converting the order from a DOM object to an XML string

3.4.4 Task “Write order into database”

This task contains a MySQL connector. It contains the following SQL statement that writes the message into the database:

```
insert into csmessage (msg_id, msg_type, msg_state, msg_content)
values ("${iD}", "order", "open", "${msgContent}");
```

Listing 5: Wrting the order into the database

`${iD}` and `${msgContent}` are replaced by the actual values of these process variables.

3.5 Process “Retrieve Order Responses”

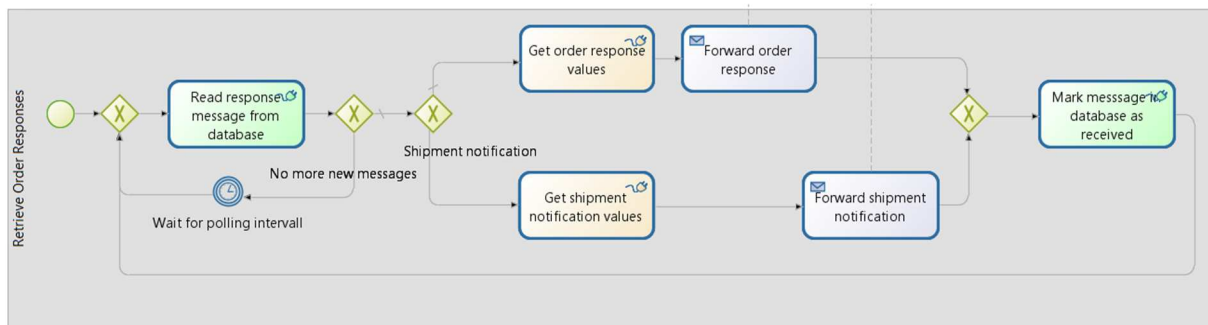


Figure 30: Process „Send Order“ (detail from Figure 7)

This process must be started exactly once, at the beginning. The only process instance then regularly checks the database for new messages.

3.5.1 Variables in the process “Retrieve Order Responses”

This process contains the following variables:

Name	Type	Description
currentID	Text	Stores the ID of the message that is currently being processed
IDs	java.util.List	A list containing the IDs of all retrieved messages.
types	java.util.List	A list containing the types of all retrieved messages.
contents	java.util.List	A list containing the message contents of all retrieved messages
Retrieved order response	XML	If the retrieved message is an order response, it is assigned to this variable which is based on the XML schema “OrderResponse” (see paragraph 3.1)
Retrieved shipment notification	XML	If the retrieved message is a shipment notification, it is assigned to this variable which is based on the XML schema “ShipmentNotification” (see paragraph 3.1)

Table 3: Variables of the process „Retrieve Order Responses“

3.5.2 Task “Read response message from database”

The database connector in this task reads order responses and shipment notifications which are in state “open”. This is achieved by the SQL statement in Listing 6.

```
select * from csmessgae
where msg_state="open"
and (msg_type="order_response" or msg_type="shipment_notification")
```

Listing 6: Retrieving order responses and shipment notifications from the database

The relevant columns of the retrieved data set are stored in the lists “IDs”, “types” and “contents” (Figure 31).

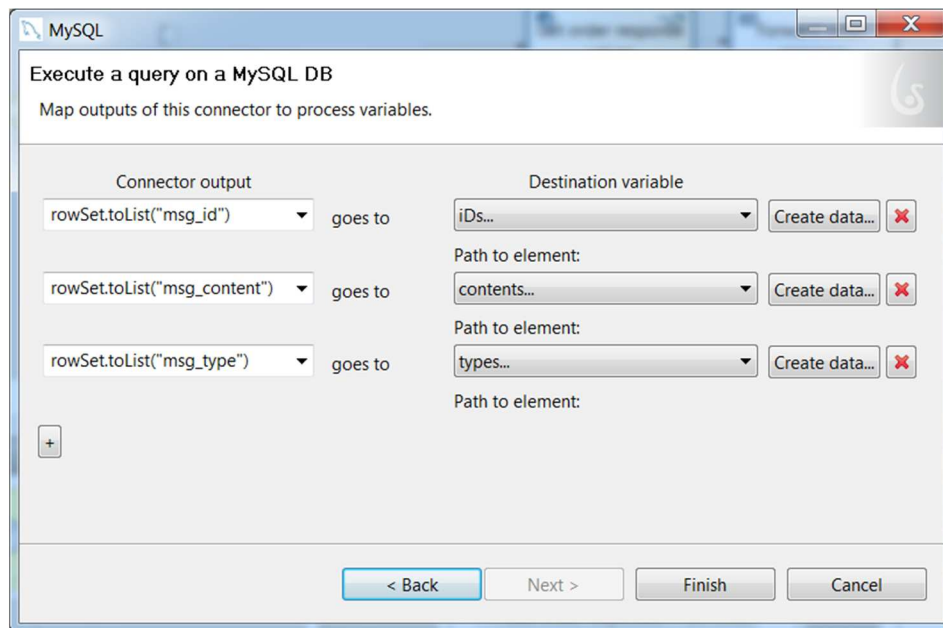


Figure 31: Assigning the retrieved data to process variables

3.5.3 Second gateway – more messages?

The first gateway right after the start event just joins three incoming sequence flows.

The second gateway needs more attention since it splits the sequence flow and therefore requires conditions at the outgoing sequence flows.

If one or more new messages have been retrieved, the default exit is taken. If there aren't any new messages, the process proceeds to the timer event. This is tested by the condition `"iDs.isEmpty()"` at the sequence flow going downwards.

3.5.4 Timer message event "Wait for polling interval"

Here, the process waits for a certain duration before it starts the next database query for new messages. For testing purposes we have used a rather short time, 10 seconds (Figure 32). This has the advantage that you needn't wait long until a sent message is retrieved. On the other hand, this short duration causes the single process instance to be active very often.

Sometimes, this causes update problems in Bonita, e.g. when the administrator tries to delete the process instance while it is active, or when the process is re-deployed via Bonita Studio. In some cases we had to re-start the server before we could delete the process instance and re-deploy the process. We also experienced the problem, that the dashboard could not be displayed while this process instance was running. These problems may be avoided by using a longer duration, e.g. several minutes.

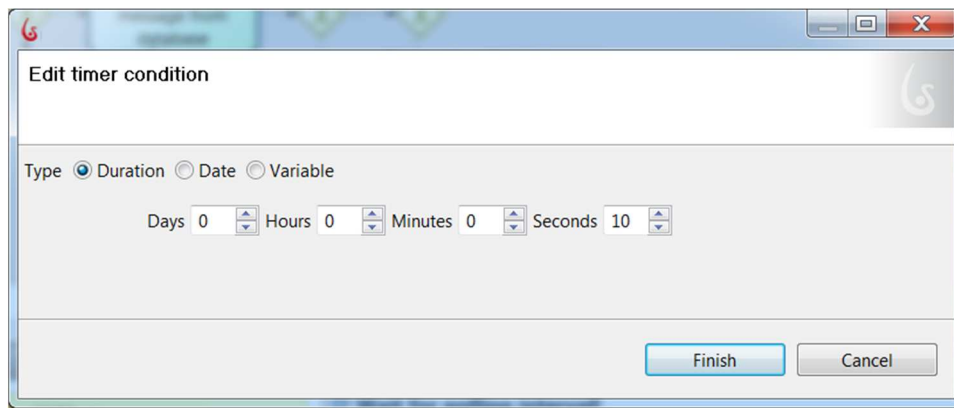


Figure 32: Timer condition

3.5.5 Third gateway – message Type

This gateway distinguishes between the two received messages types. We handle only one message a time, i.e. we just use the first retrieved message. If more than one new message has been retrieved, the other messages will be handled in the following loops.

For order responses the upper path is used (default flow), while shipment notifications take the lower path. The condition at the lower outgoing sequence flow is

```
"types.get(0).equals("shipment_notification")".
```

3.5.6 Task “Get order response values”

Here, a “Set Variables” connector is used for copying the first entries of the lists “iDs” and “contents” to the process variables “currentID” and “Retrieved order response” (Figure 33). As already mentioned, we only process the first retrieved message.

Variables *	
Name	Value
currentID	<code>\${iDs.get(0)}</code>
retrieved_order_res...	<code>\${contents.get(0)}</code>

Figure 33: Assigning the message content to process variables

Although “contents” contains texts, such a text can be directly assigned to the XML variable “Retrieved order response”, since it contains valid XML according to the order response schema definition.

3.5.7 Task “Forward order response”

This send task uses a message flow to forward the received order response message to the “Place Order” process where it is received by the intermediate message event “Order response received” (see paragraph 3.3.5). Figure 34 shows the message that is sent. It contains two variables: “order resp order ref id” and the XML variable “order response message content”. The latter also conforms to the XML schema “OrderResponse”. The default value is “`${retrieved_order_response}...`”, i. e. the content of the process variable “Retrieved order response” is just copied into this message variable.

“order resp order ref id” contains the ID of the original order that is referenced in the order response. This is required in order to assign it to the correct process instance of “Place order” (see paragraph 3.3.5). The ID of the referenced order is retrieved with the following expression:

```
${providedscripts.BonitaXML.evaluateXPathOnVariable(
  retrieved_order_response, "/OrderResponse/@OrderIDRef")}
```

This expression is provided as default value for the message variable “order resp order ref id”.

Figure 34: Defining the forwarded message for a received order response

3.5.8 Task “Mark message in database as received”

The last task in this process changes the status of the message in the database from “open” to “received”, so that it won’t be retrieved a second time. This is achieved with the SQL code in Listing 7.

```
update csmessage
set msg_state="received"
where (msg_id="${currentID}")
```

Listing 7: SQL code for changing the message state to “received”

After this task, the process loops back to the first task that looks for the next new message in the database.

3.5.9 Processing a shipment notification

If the retrieved task isn’t an order response, but a shipment notification, the process will take the lower sequence flow in the third gateway.

The two tasks in the lower path are just the equivalents to the two tasks in the upper path, only dealing with a shipment notification instead of an order response. The two paths are joined at the fourth gateway, so that for a shipment notification the corresponding entry in the database is also marked as “received”.

3.6 Process “Respond to Order”

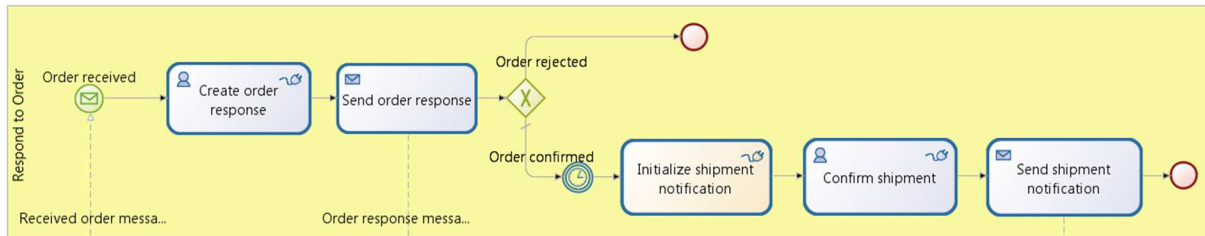


Figure 35: Process „Respond to Order“ (detail from Figure 8)

3.6.1 Process variables

This process is the supplier’s counterpart of the customer’s “Place Order” process. It has the following variables:

Name	Type	Description
Order ID	Text	The ID of the received order
response	Text	The result of the decision whether the order is accepted (“yes” or “no”).
Order response	XML	The created order response.
Shipment notification	XML	The created shipment notification

Table 4: Variables of the process „Retrieve Order Responses“

3.6.2 Start event “Order received”

The process is started when a “Received order message” arrives. This message is sent by the task “Forward order” in the process “Retrieve Orders”. The start event contains a “Set Variables” connector that copies the values of the received order into the Order response. The order ID is copied into the attribute “OrderIDRef” that is used to reference the original order. The order response receives a unique ID which is determined like the order ID in the customer system (see paragraph 3.3.2). The suppliers’ message IDs get the prefix “S-”.

3.6.3 Creating and sending an order response

“Create order response” is a user task which displays the order attributes that have been copied into the order response. The user can confirm or reject the order by selecting “yes” or “no”. There is also a set variables connector in this task that sets the values of the text variables “Order ID” and “response”.

The task “Send order response” forwards the order response to the process “Send Order Response”. This is very similar to the “Send order” task in the customer’s “Place Order” process (see paragraph 3.3.4). If the value of “response” is “no”, the process is finished after the gateway. If it is “yes”, it proceeds with the timer intermediate event. At this event, the process waits for 5 seconds. This short delay causes Bonita to put the next user task into the task list rather than opening the connected form right away. It is more appropriate to have that task in the task list, because usually the goods are shipped at a later time.

3.6.4 Creating and sending a shipment notification

The “Set variables” connector of “Initialize shipment notification” copies the required values from the order response into the shipment notification. The shipment notification also receives a unique ID which is determined in the same way as the order response’s ID.

“Confirm shipment” is a user task that shows the order’s details and allows the user to confirm that the product has been shipped. There is also a “Set Variable” connector that assigns the current date to the shipment notification’s attribute “ShipmentDate”. The parameters of this connector are shown in Table 5.

Name	shipment_notification\$/ShipmentNotification/@ShipmentDate
Value	<pre> import java.util.Date; import java.text.DateFormat; import java.text.SimpleDateFormat; DateFormat dateFormat = new SimpleDateFormat("dd.MM.yyyy"); dateFormat.format(new Date()); </pre>

Table 5: Assigning the shipment date

The last task, “Send shipment notification” is similar to “Send order response”.

3.7 Process “Send Order Response”

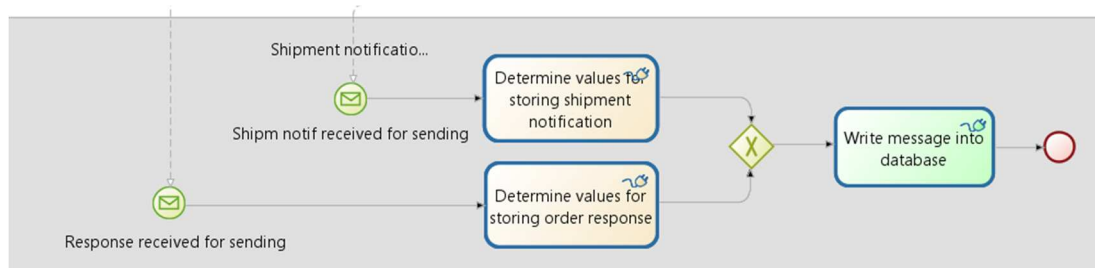


Figure 36: Process „Send Order Response“ (detail from Figure 8)

This process is very similar to the customer process “Send Order” (see paragraph 3.4). The main difference is that here we have two start events for the two different message types. The first task in each path is specific for each message type. After that, the two paths are joined, and the created message is written into the database.

3.8 Process “Retrieve Orders”

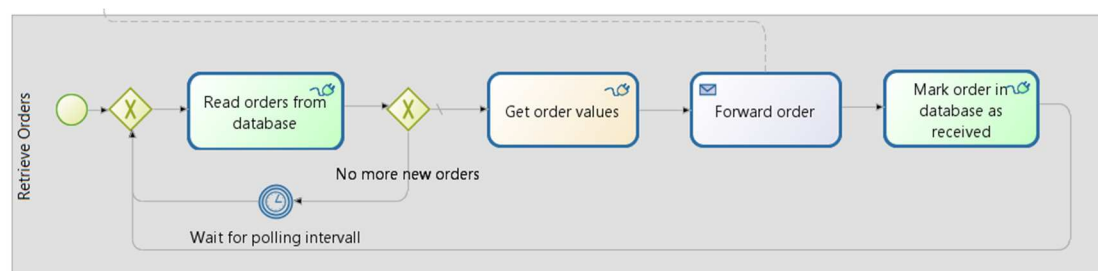


Figure 37: Process „Retrieve Orders“ (detail from Figure 8)

This process is very similar to the customer's process "Retrieve Order Responses" (see paragraph 3.5). However, in this case we only have one message type "Order". Therefore the distinction between two different message types in the "Retrieve Order Responses" process is not required here.

4 Running the processes

In this paragraph we show how you can actually work with the processes we have developed. We have deployed the processes on two different computers so that we actually have separate portals for customer and supplier. If you run both processes on one Bonita server, you will receive both the customer's and the supplier's tasks in the inbox.

As an alternative, you could change one of the models so that different users perform the two roles, and you can switch between the roles by logging in as a different user.

In our models we have always defined "admin" as the user, so that both customer and supplier need to login as admin into their User XP portals. This has the advantage that it is easy to switch into the "Administration" view if we want to inspect the current states of the different process instances.

4.1 Preparations on the customer side

After deploying the customer's processes, all three processes are shown on the left side of the customer's portal under "start a case".

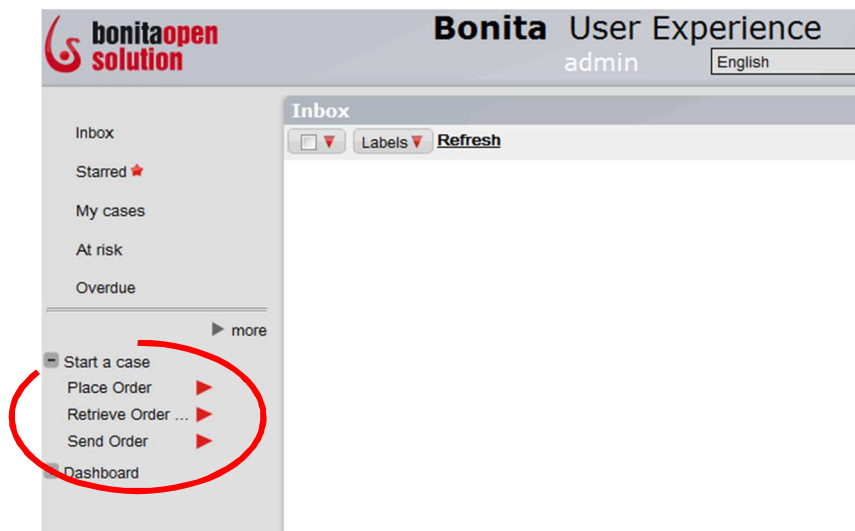


Figure 38: The customer's processes in the portal

Unfortunately, Bonita does not distinguish between those processes that should be started by a specific user and those that shouldn't be started manually. "Send Order" should not be started manually, since it will be triggered automatically by a message from the process "Place Order".

Before we can place orders and receive order responses, we need to start the process "Retrieve Order Responses":



Figure 39: Starting the Retrieve Order Responses process (customer)

Note that process instances are called “cases” in Bonita.

Since this process doesn’t contain any user interface, there is just a button for starting the case:

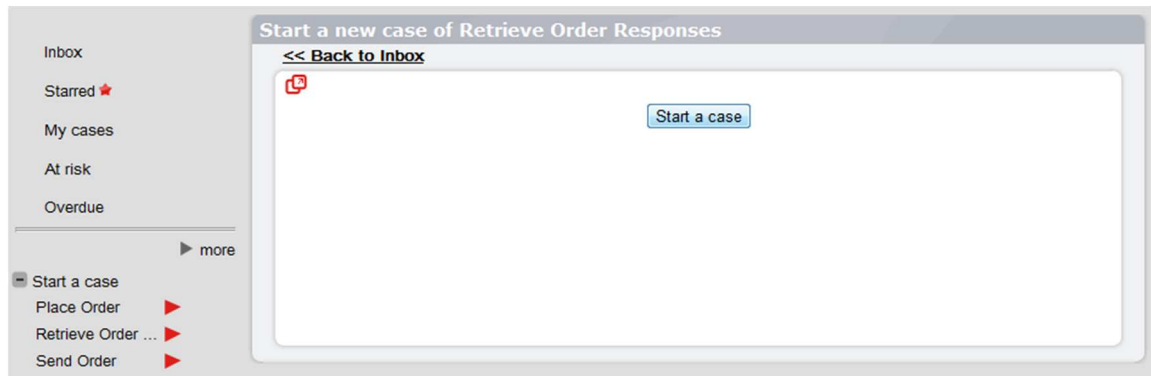


Figure 40: Just a start button for starting a new process instance (customer)

This process must be started only once, at the beginning. The only process instance regularly looks for new order responses and shipment notifications in the database. If there were several instances of this process, these instances could interfere with each other, since we haven’t implemented a mechanism for dealing with concurrent access of several instances. Therefore, it could happen that the first process instance reads an order response with state “open”, and the second process instance would retrieve the same order response before the first process instance has marked the order instance as “received”. In this case, two messages would be sent to the process “Place Order”. Here, this wouldn’t be a big problem, since there is only one instance of the “Place Order” process awaiting such a message, i.e. the second message would be ignored. However, if the same thing happened on the supplier side, the process “Respond to Order” would be started twice, since every message with an order starts a new process instance.

The problem could be solved by using a database transaction that includes both retrieving the message and marking it as “received”. Only after this transaction has completed successfully, the process would send the message to the “Place Order” process.

When we switch into the administration view, the single process instance can be seen:

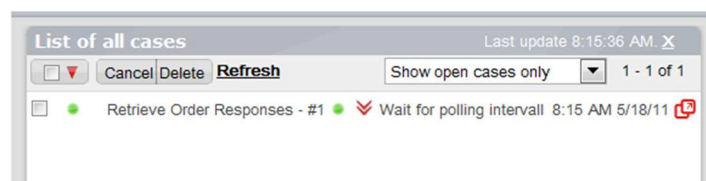


Figure 41: The single process instance in the administration view (customer)

4.2 Preparations on the supplier side

On the supplier side, we also have deployed three processes which are shown in the portal:

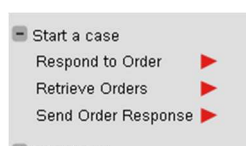


Figure 42: The supplier’s processes

Here, we need to start the process “Retrieve Orders”. This is the supplier’s counterpart to “Retrieve Order Responses”. It also must be started only once, at the beginning.

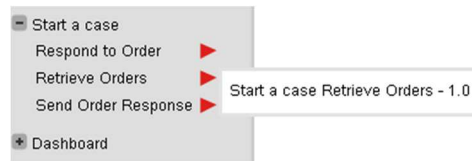


Figure 43: Starting the process “Retrieve Orders” (supplier)

Again, we can check the correct startup of the process in the administration view:



Figure 44: The single process instance in the administration view (supplier)

4.3 Placing an order

Now the preparations of our collaborative scenario have been finished, and the customer can start placing orders. This is simply done by starting a case of the process “Place Order”:



Figure 45: Starting the process “Place Order” (customer)

We start the process three times and create three orders. The automatically assigned order IDs are “C-12”, “C-13”, and “C-14”.

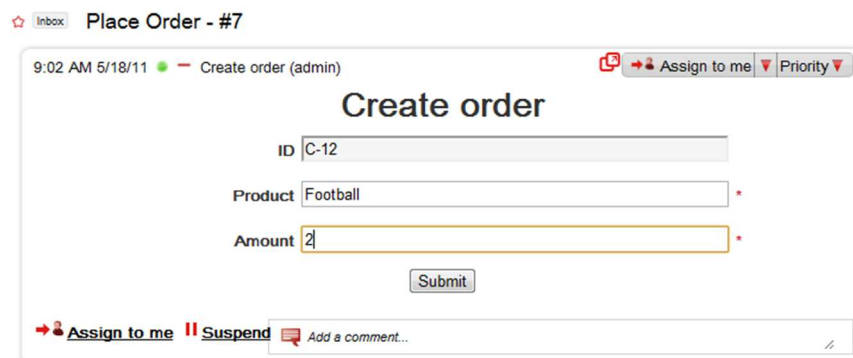


Figure 46: Creating an order (customer)

☆ Inbox Place Order - #8

9:02 AM 5/18/11 ● Create order (admin) Assign to me Priority

Create order

ID

Product

Amount

Assign to me Suspend Add a comment...

Figure 47: Creating another order (customer)

<< Back to Inbox Labels Refresh

☆ Inbox Place Order - #9

9:02 AM 5/18/11 ● Create order (admin) Assign to me Priority

Create order

ID

Product

Amount

Assign to me Suspend Add a comment...

Figure 48: Creating a third order (customer)

Note that Bonita has assigned IDs to the three process instances (#7, #8, #9). Since we are using our own numbering schemes for identifying orders, the order IDs are entirely independent from the process instance IDs.

If we want to see what's happening in the database, we can use a database inspection tool. We have used DbVisualizer Free by DbVis Software AB.

Here we find the three messages C-12, C-13, C-14. At first, their status is "open":

	msg_id	msg_type	msg_status	msg_content
1	C-12	order	open	<?xml version="1.0" encoding="UTF-8" standalone="no"?><tns:Order xmlns:tns="http://www.example.org/Order" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.example.org/Order http://www.w3.org/2001/XMLSchema-instance" />
2	C-13	order	open	<?xml version="1.0" encoding="UTF-8" standalone="no"?><tns:Order xmlns:tns="http://www.example.org/Order" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.example.org/Order http://www.w3.org/2001/XMLSchema-instance" />
3	C-14	order	open	<?xml version="1.0" encoding="UTF-8" standalone="no"?><tns:Order xmlns:tns="http://www.example.org/Order" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.example.org/Order http://www.w3.org/2001/XMLSchema-instance" />
4	C-10	order	received	<?xml version="1.0" encoding="UTF-8" standalone="no"?><tns:Order xmlns:tns="http://www.example.org/Order" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.example.org/Order http://www.w3.org/2001/XMLSchema-instance" />
5	C-11	order	received	<?xml version="1.0" encoding="UTF-8" standalone="no"?><tns:Order xmlns:tns="http://www.example.org/Order" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.example.org/Order http://www.w3.org/2001/XMLSchema-instance" />
6	C-7	order	received	<?xml version="1.0" encoding="UTF-8" standalone="no"?><tns:Order xmlns:tns="http://www.example.org/Order" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.example.org/Order http://www.w3.org/2001/XMLSchema-instance" />
7	C-8	order	received	<?xml version="1.0" encoding="UTF-8" standalone="no"?><tns:Order xmlns:tns="http://www.example.org/Order" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.example.org/Order http://www.w3.org/2001/XMLSchema-instance" />
8	C-9	order	received	<?xml version="1.0" encoding="UTF-8" standalone="no"?><tns:Order xmlns:tns="http://www.example.org/Order" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.example.org/Order http://www.w3.org/2001/XMLSchema-instance" />
9	S-13	order_response	received	<?xml version="1.0" encoding="UTF-8" standalone="no"?><tns:OrderResponse xmlns:tns="http://www.example.org/OrderResponse" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.example.org/OrderResponse http://www.w3.org/2001/XMLSchema-instance" />
10	S-14	order_response	received	<?xml version="1.0" encoding="UTF-8" standalone="no"?><tns:OrderResponse xmlns:tns="http://www.example.org/OrderResponse" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.example.org/OrderResponse http://www.w3.org/2001/XMLSchema-instance" />
11	S-16	shipment_notification	received	<?xml version="1.0" encoding="UTF-8" standalone="no"?><tns:ShipmentNotification xmlns:tns="http://www.example.org/ShipmentNotification" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.example.org/ShipmentNotification http://www.w3.org/2001/XMLSchema-instance" />

Figure 49: The order messages in the message queue (database)

4.4 Receiving an order and responding to it

Since the supplier's polling process "Retrieve Orders" is already running, the message states change to "received" after a few moments:

	ms	msg_type	msg_state	
1	C-10	order	received	<?xml version='1.0' encoding='UTF-8' standalone='no'?><tns:Order xmlns:tns='http://www.example.org/Order' xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance' xsi:schemaLocation='http://www.example.org/Order http://www.example.org/Order.xsd'>
2	C-14	order	received	</tns:Order></tns:Order xmlns:tns='http://www.example.org/Order' xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance' xsi:schemaLocation='http://www.example.org/Order http://www.example.org/Order.xsd'/>
3	C-12	order	received	<?xml version='1.0' encoding='UTF-8' standalone='no'?><tns:Order xmlns:tns='http://www.example.org/Order' xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance' xsi:schemaLocation='http://www.example.org/Order http://www.example.org/Order.xsd'>
4	C-13	order	received	<?xml version='1.0' encoding='UTF-8' standalone='no'?><tns:Order xmlns:tns='http://www.example.org/Order' xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance' xsi:schemaLocation='http://www.example.org/Order http://www.example.org/Order.xsd'>
5	C-14	order	received	<?xml version='1.0' encoding='UTF-8' standalone='no'?><tns:Order xmlns:tns='http://www.example.org/Order' xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance' xsi:schemaLocation='http://www.example.org/Order http://www.example.org/Order.xsd'>
6	C-7	order	received	<?xml version='1.0' encoding='UTF-8' standalone='no'?><tns:Order xmlns:tns='http://www.example.org/Order' xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance' xsi:schemaLocation='http://www.example.org/Order http://www.example.org/Order.xsd'>
7	C-8	order	received	<?xml version='1.0' encoding='UTF-8' standalone='no'?><tns:Order xmlns:tns='http://www.example.org/Order' xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance' xsi:schemaLocation='http://www.example.org/Order http://www.example.org/Order.xsd'>
8	C-9	order	received	<?xml version='1.0' encoding='UTF-8' standalone='no'?><tns:Order xmlns:tns='http://www.example.org/Order' xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance' xsi:schemaLocation='http://www.example.org/Order http://www.example.org/Order.xsd'>
9	S-13	order_response	received	<?xml version='1.0' encoding='UTF-8' standalone='no'?><tns:OrderResponse xmlns:tns='http://www.example.org/OrderResponse' xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance' xsi:schemaLocation='http://www.example.org/OrderResponse http://www.example.org/OrderResponse.xsd'>
10	S-14	order_response	received	<?xml version='1.0' encoding='UTF-8' standalone='no'?><tns:OrderResponse xmlns:tns='http://www.example.org/OrderResponse' xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance' xsi:schemaLocation='http://www.example.org/OrderResponse http://www.example.org/OrderResponse.xsd'>
11	S-16	shipment_notification	received	<?xml version='1.0' encoding='UTF-8' standalone='no'?><tns:ShipmentNotification xmlns:tns='http://www.example.org/ShipmentNotification' xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance' xsi:schemaLocation='http://www.example.org/ShipmentNotification http://www.example.org/ShipmentNotification.xsd'>

Figure 50: The order messages have been marked as “received” by the supplier’s “Retrieve Order” process (database)


After pressing “refresh”, the supplier finds three tasks “Create order response” in his inbox, one for each retrieved order:

Inbox				Last update 9:04:18 AM	X
<input type="checkbox"/>	▼	Labels ▼	Refresh	1 - 3 of 3	
<input type="checkbox"/>	☆	Inbox	Respond to Order - #9   Create order response	9:04 AM 5/18/11	
<input type="checkbox"/>	☆	Inbox	Respond to Order - #8   Create order response	9:04 AM 5/18/11	
<input type="checkbox"/>	☆	Inbox	Respond to Order - #7   Create order response	9:04 AM 5/18/11	

Figure 51: "Create order response" tasks in the inbox (supplier)

He opens the task in the middle. Here, he can inspect the order details and state whether he accepts this order. In this case, he selects “yes”. The supplier’s system has assigned the ID “S-19”, but the order response also contains the order id “C-13” as a reference to the original order.

 [Inbox](#) **Respond to Order - #8**

9:04 AM 5/18/11  [Create order response \(admin\)](#)  [Assign to me](#)  [Priority](#) 

Create order response

ID

Order ID

Product

Amount

Confirm order? ☒ yes ☐ no

Figure 52: Creating an order response (supplier)

After pressing the “Submit” button, the order response is sent to the process “Send Order Response”. This process writes the order into the database.

	ms	msg_type	msg_state	
1	C-10	order	received	<?xml version="1.0" encoding="UTF-8" standalone="no"?><tns:Order xmlns:tns="http://www.example.org/Order" xmlns:xsi="http://www.w3.org/2001/XMLSchema"...
2	C-11	order	received	<?xml version="1.0" encoding="UTF-8" standalone="no"?><tns:Order xmlns:tns="http://www.example.org/Order" xmlns:xsi="http://www.w3.org/2001/XMLSchema"...
3	C-12	order	received	<?xml version="1.0" encoding="UTF-8" standalone="no"?><tns:Order xmlns:tns="http://www.example.org/Order" xmlns:xsi="http://www.w3.org/2001/XMLSchema"...
4	C-13	order	received	<?xml version="1.0" encoding="UTF-8" standalone="no"?><tns:Order xmlns:tns="http://www.example.org/Order" xmlns:xsi="http://www.w3.org/2001/XMLSchema"...
5	C-14	order	received	<?xml version="1.0" encoding="UTF-8" standalone="no"?><tns:Order xmlns:tns="http://www.example.org/Order" xmlns:xsi="http://www.w3.org/2001/XMLSchema"...
6	C-7	order	received	<?xml version="1.0" encoding="UTF-8" standalone="no"?><tns:Order xmlns:tns="http://www.example.org/Order" xmlns:xsi="http://www.w3.org/2001/XMLSchema"...
7	C-8	order	received	<?xml version="1.0" encoding="UTF-8" standalone="no"?><tns:Order xmlns:tns="http://www.example.org/Order" xmlns:xsi="http://www.w3.org/2001/XMLSchema"...
8	C-9	order	received	<?xml version="1.0" encoding="UTF-8" standalone="no"?><tns:Order xmlns:tns="http://www.example.org/Order" xmlns:xsi="http://www.w3.org/2001/XMLSchema"...
9	S-13	order_response	received	<?xml version="1.0" encoding="UTF-8" standalone="no"?><tns:OrderResponse xmlns:tns="http://www.example.org/OrderResponse" xmlns:xsi="http://www.w3.org/2001/XMLSchema"...
10	S-14	order_response	received	<?xml version="1.0" encoding="UTF-8" standalone="no"?><tns:OrderResponse xmlns:tns="http://www.example.org/OrderResponse" xmlns:xsi="http://www.w3.org/2001/XMLSchema"...
11	S-19	order_response	open	<?xml version="1.0" encoding="UTF-8" standalone="no"?><tns:OrderResponse xmlns:tns="http://www.example.org/OrderResponse" xmlns:xsi="http://www.w3.org/2001/XMLSchema"...

Figure 53: The order response in the message queue (database)

After a few moments, the customer's process "Retrieve Order Responses" marks the new order response as received:

ms	1	msg_type	msg_state	
1	C-10	order	received	<?xml version="1.0" encoding="UTF-8" standalone="no"?><tns:Order xmlns:tns="http://www.example.org/Order" xmlns:xsi="http://www.w3.org/2001/XMLSchema"...
2	C-11	order	received	<?xml version="1.0" encoding="UTF-8" standalone="no"?><tns:Order xmlns:tns="http://www.example.org/Order" xmlns:xsi="http://www.w3.org/2001/XMLSchema"...
3	C-12	order	received	<?xml version="1.0" encoding="UTF-8" standalone="no"?><tns:Order xmlns:tns="http://www.example.org/Order" xmlns:xsi="http://www.w3.org/2001/XMLSchema"...
4	C-13	order	received	<?xml version="1.0" encoding="UTF-8" standalone="no"?><tns:Order xmlns:tns="http://www.example.org/Order" xmlns:xsi="http://www.w3.org/2001/XMLSchema"...
5	C-14	order	received	<?xml version="1.0" encoding="UTF-8" standalone="no"?><tns:Order xmlns:tns="http://www.example.org/Order" xmlns:xsi="http://www.w3.org/2001/XMLSchema"...
6	C-7	order	received	<?xml version="1.0" encoding="UTF-8" standalone="no"?><tns:Order xmlns:tns="http://www.example.org/Order" xmlns:xsi="http://www.w3.org/2001/XMLSchema"...
7	C-8	order	received	<?xml version="1.0" encoding="UTF-8" standalone="no"?><tns:Order xmlns:tns="http://www.example.org/Order" xmlns:xsi="http://www.w3.org/2001/XMLSchema"...
8	C-9	order	received	<?xml version="1.0" encoding="UTF-8" standalone="no"?><tns:Order xmlns:tns="http://www.example.org/Order" xmlns:xsi="http://www.w3.org/2001/XMLSchema"...
9	S-13	order_response	received	<?xml version="1.0" encoding="UTF-8" standalone="no"?><tns:OrderResponse xmlns:tns="http://www.example.org/OrderResponse" xmlns:xsi="http://www.w3.org/2001/XMLSchema"...
10	S-14	order_response	received	<?xml version="1.0" encoding="UTF-8" standalone="no"?><tns:OrderResponse xmlns:tns="http://www.example.org/OrderResponse" xmlns:xsi="http://www.w3.org/2001/XMLSchema"...
11	S-15	order_response	received	<?xml version="1.0" encoding="UTF-8" standalone="no"?><tns:OrderResponse xmlns:tns="http://www.example.org/OrderResponse" xmlns:xsi="http://www.w3.org/2001/XMLSchema"...
12	S-19	order_response	received	<?xml version="1.0" encoding="UTF-8" standalone="no"?><tns:OrderResponse xmlns:tns="http://www.example.org/OrderResponse" xmlns:xsi="http://www.w3.org/2001/XMLSchema"...

Figure 54: The order response message has been marked as “received” by the customer’s “Retrieve Order Response” process (database)

Since the supplier has accepted the order, he receives a new task in his inbox: “Confirm shipment for order no. C-13”.

Inbox			Last update 9:06:32 AM
<input type="checkbox"/> Labels Refresh			1 - 3 of 3
<input type="checkbox"/>	Inbox: Respond to Order - #8	<input type="checkbox"/> Confirm shipment for order no. C-13	9:04 AM 5/18/11
<input type="checkbox"/>	Inbox: Respond to Order - #9	<input type="checkbox"/> Create order response	9:04 AM 5/18/11
<input type="checkbox"/>	Inbox: Respond to Order - #7	<input type="checkbox"/> Create order response	9:04 AM 5/18/11

Figure 55: New task “Confirm shipment” in the inbox (supplier)

Since he hasn’t shipped this order yet, he selects one of the other tasks instead and creates another order response:

Inbox: Respond to Order - #9

9:04 AM 5/18/11 ☐ Create order response (admin)

☐ Assign to me
 ☐ Priority

Create order response

ID

Order ID

Product

Amount

Confirm order? ☐ yes ☒ no

Submit

Figure 56: Creating another order response (supplier)

In this case, he has selected “no”. Therefore, he doesn’t receive another “Confirm shipment” task in his inbox:

Inbox			1 - 2 of 2
<input type="checkbox"/> Labels Refresh			
<input type="checkbox"/>	Inbox: Respond to Order - #8	<input type="checkbox"/> Confirm shipment for order no. C-13	9:04 AM 5/18/11
<input type="checkbox"/>	Inbox: Respond to Order - #7	<input type="checkbox"/> Create order response	9:04 AM 5/18/11

Figure 57: Since the order has been rejected, no shipment confirmation task has been created (supplier)

4.5 Receiving an order response

In the customer’s portal, there are now two tasks for showing the received order responses:

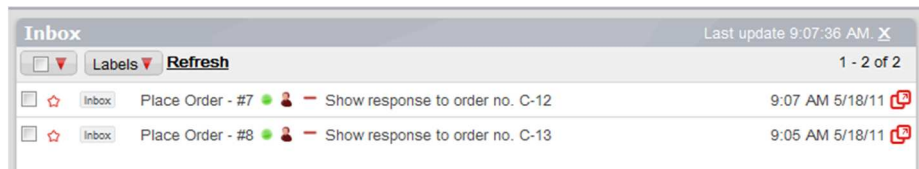


Figure 58: "Show order response" tasks in the inbox (customer)

When we select each, we can see our original order details, and compare them to the contents of the respective order response.

Figure 59: Inspecting the first order response (customer)

Here is the same dialog for order no. C-13:

Figure 60: Inspecting the second order response (customer)

It can be seen that both order responses have been correctly assigned to the original orders.

When we change to the administration view, we can see the current states of our process instances:



Figure 61: Current state of process instances in the administration view (customer)

There is still the single process instance of “Retrieve Order Responses” which continues to look into the database for new order responses and shipment notifications. Since one of our orders, order no. C-12 has been rejected; the respective process instance (“Place Order #7”) has been finished after showing the order response. Therefore this instance is not in the list of open cases anymore.

The instance “Place Order #8” is now waiting for the shipment notification. The other process instance is still expecting an order response.

4.6 Confirming a shipment

When the supplier has finally shipped order no. C-13, he confirms this shipment:

9:05 AM 5/18/11 • Confirm shipment for order no. C-13 (admin) Assign to me Priority

Confirm shipment

ID: S-21

Order ID: C-13

Product: Skateboard

Amount: 1

Confirm shipment

Figure 62: Confirming a shipment (supplier)

After pressing the button “Confirm Shipment”, the shipment date is automatically added to the shipment notification. Then, the process “Send Order Response” writes the notification in the database.

4.7 Receiving a shipment order

On the customer’s side, the new shipment notification is again collected by the process “Retrieve Order Responses”, and forwarded to the corresponding “Place Order” instance. This instance creates the task “Show shipment notification to order no. C-13” in the customer’s inbox::

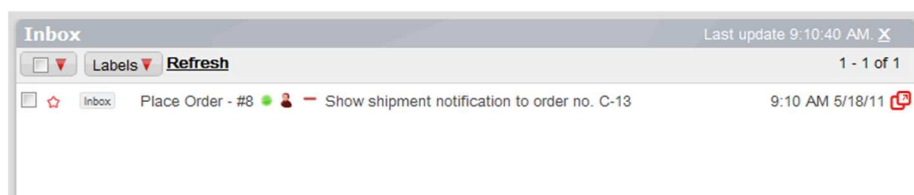


Figure 63: Task “Show shipment notification” in the inbox (customer)

After opening this task, we can compare the contents of our order and that of the shipment notification, and we see the date on which the order was shipped.

Place Order - #8

9:10 AM 5/18/11 ● - Show shipment notification to order no. C-13 (admin) Assign to me Priority ▼

Show shipment notification

Your order	Shipment notification
ID <input type="text" value="C-13"/>	ID <input type="text" value="S-21"/>
Product <input type="text" value="Skateboard"/>	Order ID <input type="text" value="C-13"/>
Amount <input type="text" value="1"/>	Product <input type="text" value="Skateboard"/>
	Amount <input type="text" value="1"/>
	Shipment date <input type="text" value="18.05.2011"/>

Assign to me Suspend Add a comment...

Figure 64: Inspecting the shipment notification (customer)

Now, this process instance is also finished.

4.8 A shipment order is received before the order response has been processed

The supplier still needs to send an answer to order no. C-14:

Inbox

Labels ▼ Refresh 1 - 1 of 1

Inbox Respond to Order - #7 ● - Create order response 9:04 AM 5/18/11 Assign to me

Figure 65: The last "Create order response" task in the inbox (supplier)

He confirms that order:

Inbox

Back to Inbox Labels ▼ Refresh

Inbox Respond to Order - #7 ● - Create order response (admin) Assign to me Priority ▼

Create order response

ID

Order ID

Product

Amount

Confirm order? ☒ yes ☐ no

Figure 66: Confirming the last order (supplier)

His final task is to confirm the shipment of this order:



Figure 67: The “Confirm shipment” task for the last order (supplier)

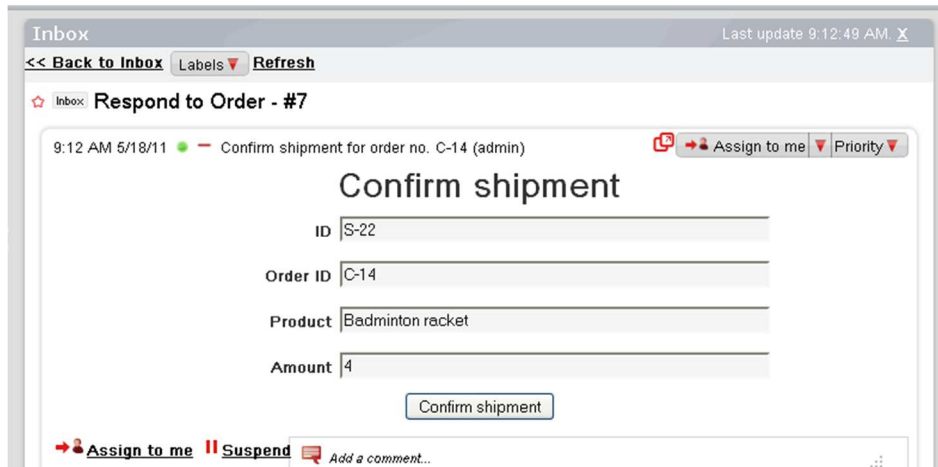


Figure 68: Confirming the last order (supplier)

In this case, the supplier has sent the shipment notification before the customer has read the order response. Still, the customer only has one task in his inbox, “Show response to order no. C-14”:



Figure 69: Although a shipment notification has been received, there is still only the task “Show response” in the inbox (customer)

This happens, because in the process model, the catching message event “Shipment notification received” comes after the task “Show order response”. This means that the process does not detect the arrival of a shipment notification until the user has completed the task “Show order response”.

Luckily, Bonita buffers this event so that it doesn’t get lost and we still get the shipment notification.

But first we need to open the order response:

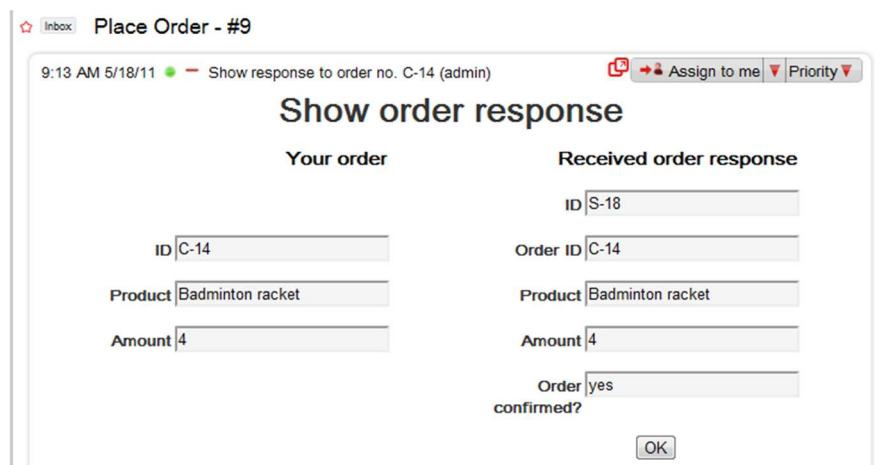


Figure 70: Inspecting the last order response (customer)

After completing this task with the “OK” button, the process instance detects the waiting message event and creates the task for showing the shipment notification:



Figure 71: Now the shipment notification task is shown (customer)

 A screenshot of a web application showing a task titled 'Show shipment notification'. The task is assigned to 'me' and has a priority dropdown. The form is divided into two columns: 'Your order' and 'Shipment notification'.

Your order		Shipment notification	
ID	C-14	ID	S-22
Product	Badminton racket	Order ID	C-14
Amount	4	Product	Badminton racket
		Amount	4
		Shipment date	18.05.2011

 An 'OK' button is at the bottom right.

Figure 72: Inspecting the last shipment notification task (customer)

With this task, processing of the third order also finished.

Note that the customer's process would not be able to handle the reception of a shipment notification without a prior order response. In order to handle this situation we would need to change our model so that we wait for both messages in parallel.

In our scenario, however, this is not a problem because the supplier's process always sends an order response.