Checklist: Making Correct Use of an Integration Process

Using the considerations below, you can check whether integration processes are an appropriate solution to your particular use case.

Prerequisites

SAP only recommends that you use integration processes if the following prerequisites are fulfilled:

- Message-based communication

The business systems or applications involved can communicate by means of XML messages.

- Semantic relations between messages (correlations)

The messages that are to be processed in the process are related to each other in some way, for example, a purchase order and the relevant purchase order response.

If this is not the case, check whether you are able to realize the process simply by using just the Integration Server (without integration processes) instead.

- Correlations can be defined uniquely

Correlations can be defined in such a way that they determine messages that belong together **uniquely**.

- Defined end condition

The process has a clear end condition. No instances of the process should usually last longer than a few days.

To avoid processes that run infinitely, always define a deadline as a unique stopping criterion.

- No user action required

The Process Engine supports the message choreography; however, it is **not** intended to act as a central workflow engine. For this reason, only use integration processes for processes that do not require any user action.

If user action is required, check whether you can realize the process by using Business Workflow or Guided Procedures instead.

Using the Integration Server Efficiently

Integration processes are executed on the Integration Server at runtime by the Business Process Engine. Since the Integration Server is **the** central resource for message exchange, you must ensure that it is not overloaded; otherwise, this can lead to bottlenecks or performance problems.

Resource Consumption

Every step of an integration process uses Integration Server resources.

- Every message that is sent to the Process Engine is duplicated.
- Every message that is sent from the Process Engine is duplicated.
- A work item is created for the process itself and for every step the process contains.

This means that for a process that just receives one message that is sent without being processed further, four messages and three work items are created.

For this reason, you must ensure that you take Integration Server resources into account when you define integration processes.

Do Not Transfer Application Logic

Do **not** use integration processes to transfer application logic from the application systems to the Integration Server.

No Replacement for Mass Interfaces

Check whether it would not be better to execute particular processing steps, for example, collecting messages, on the sender or receiver system.

If, for example, you only want to collect the messages from one business system to forward them together to a second business system, you should do so by using a mass interface and not an integration process.

In particular, if you want to split a message up into lots of individual messages, use a mass interface instead of an integration process. A mass interface requires only a fraction of the backend-system and Integration-Server resources that an integration process would require to carry out the same task.

Typical Usage Cases

You would normally use an integration process when you need to save the processing status for message processing within the process. If this kind of stateful processing is not required, check whether you are able to realize the process simply by using the Integration Server (without integration processes) instead.

The following table shows typical examples of where integration processes can be used:

Application Case	See also:
Send a message to multiple receivers and wait for a response message from each of the receivers. The number of receivers is determined at runtime.	Example: Multicast – Multiple Receivers (with Response Message)
Define the order in which messages from the integration process are sent.	Example: Serialization – Defining the Send Sequence
Collect multiple messages from one interface or multiple interfaces, bundle them into a single message and then forward this bundled message.	Example: Collecting and Bundling Messages from One Interface Example: Collecting and Bundling Messages from Multiple Interfaces
A synchronously calling business system and an asynchronously called business system are to communicate with each other.	Example: Sync/Async Communication
Send one message from an integration process synchronously to multiple receivers. The first response message to arrive is to be processed further.	Example: Sending to Multiple Receivers Synchronously
Define deadline monitoring for the receipt of a response message.	Example: Deadline Monitoring for Receipt of a Response Message

-

Further Checklists

Checklist: Making Correct Use of Correlations

Checklist: Making Correct Use of a ParForEach

Checklist: Making Correct Use of Mappings

Example: Multicast – Multiple Receivers (with Response Message)

You have the option of sending a message to multiple receivers and waiting for a response message from each of the receivers. The procedure of sending a message to multiple receivers and waiting for a response message is also known as ‘multicast’.

The receivers are determined at runtime from the receiver determination that is configured in the Integration Directory. The number of receivers therefore does not need to be known at design time.

You can define a multicast in different ways. The following table shows the various options. You can find the examples in the Integration Repository under *SAP Basis* → *SAP Basis 6.40*, namespace *http://sap.com/xi/XI/System/Patterns*.

Multicast	Description	Example
Send one after the other (block in *ForEach* mode)	Multiple messages are sent to individual receivers one after the other.	BpmPatternMulticastSequential
Send simultaneously (block in *ParForEach* mode)	Multiple messages are sent to individual receivers simultaneously.	BpmPatternMulticastParallel

Example Processes

The example process is started when a message in the `Message`container element is received in the first receive step. A subsequent receiver determination step calls the receiver determination that you configured in the Integration Directory and gets the receiver list in the multiline container element `Receivers`. In a dynamic block, the messages are sent to the receivers either in parallel or one after the other.

The container element `Receivers` is defined as the multiline element for dynamic processing. The container element `Receiver` is defined for the current line. This gets the individual receivers for which the block will be executed.

Each receiver sends a response message, which is received in the `Response` container element. The sent message and the response message are linked by means of a correlation.

The sent message might be a purchase order and the response message the response to the purchase order, for example. The purchase order and the purchase order response might be correlated by a purchase order number, for example.

One of the abstract interfaces is used as an inbound interface and the other as an outbound interface (see also: Process Signature).

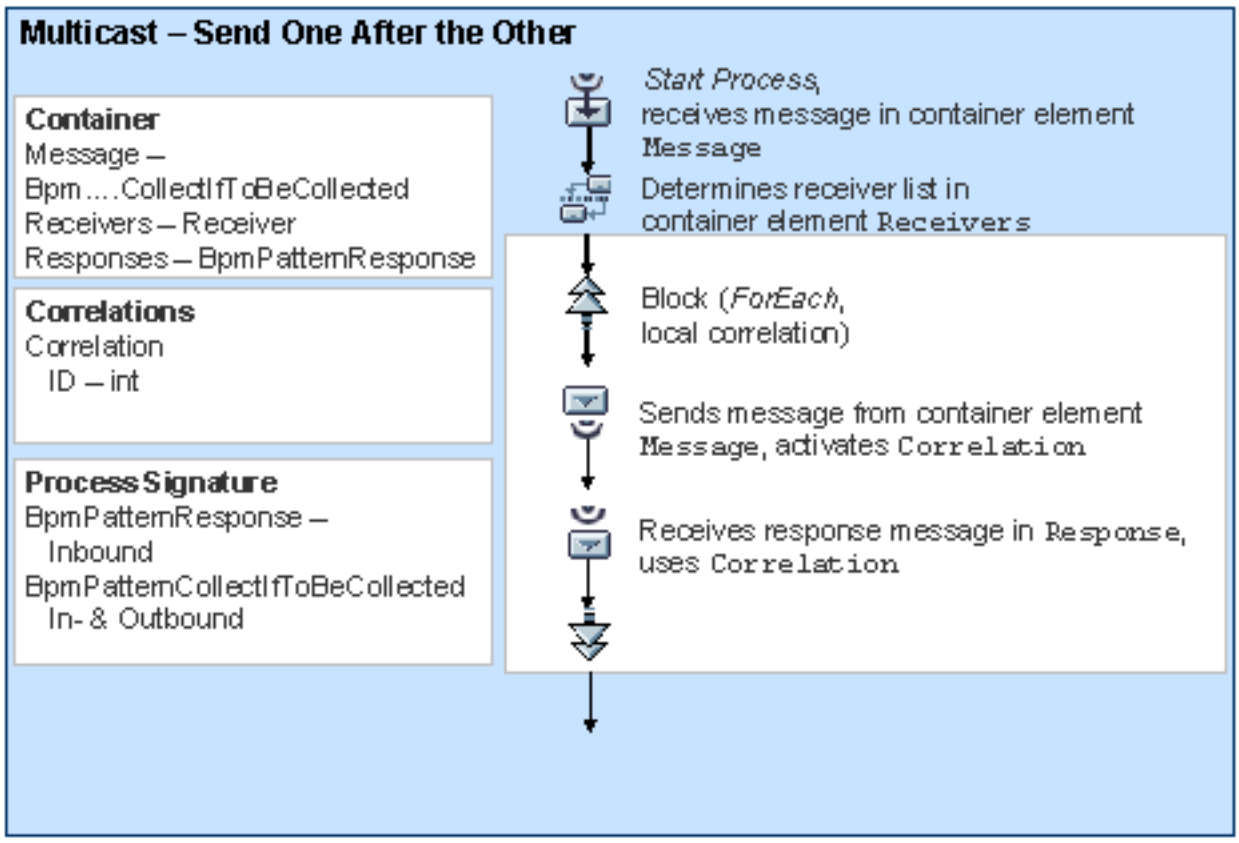
Sending One After the Other (*ForEach*)

If what is most important is that the message **is actually sent** to the various different receivers and not whether this is done simultaneously, define the block as a ForEach. Within the block, a send step sends the message to the first receiver in the receiver list and creates the correlation `Correlation`. A receive step uses `Correlation` and receives the response message from the first receiver. If this

receive step is complete, the message is sent to the next receiver in the receiver list. The whole receiver list is processed in this way.

So that a separate instance of the correlation can be processed for each receiver, the `Correlation` is defined as a local correlation.

The following graphic illustrates the process definition:



Send Simultaneously (Block with ParForEach)

The block is defined as a ParForEach. A block instance is generated for each receiver in the receiver list in which the following steps are executed:

- A send step sends the message to the receiver and activates the correlation `Correlation`.

  So that a separate instance of the correlation can be processed for block instance, the `Correlation` is defined as a local correlation.

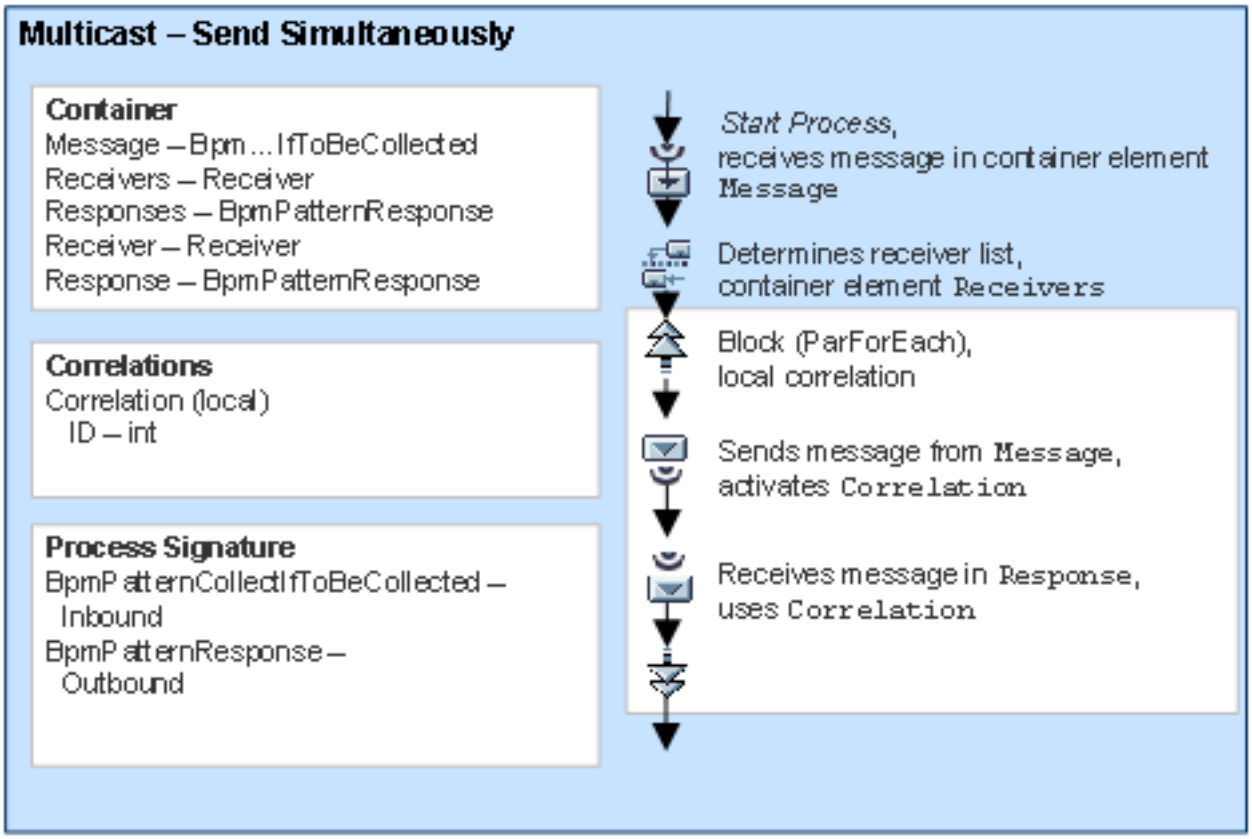
- A receiver step uses this correlation and receives the response message from this receiver.

The block is complete once all block instances that were generated in parallel are complete. In other words, the response message has been received from all receivers in the receiver list.

Note that using a ParForEach will **not** have any effect on performance.

See also: Checklist: Making Correct Use of a ParForEach

The following graphic illustrates the process definition:


Multicast – Send Simultaneously
Container
Message – Bpm...IfToBeCollected
Receivers – Receiver
Responses – BpmPatternResponse
Receiver – Receiver
Response – BpmPatternResponse
Correlations
Correlation (local)
ID – int
Process Signature
BpmPatternCollectIfToBeCollected – Inbound
BpmPatternResponse – Outbound
Start Process, receives message in container element Message
Determines receiver list, container element Receivers
Block (ParForEach), local correlation
Sends message from Message, activates Correlation
Receives message in Response, uses Correlation

Example: Serialization – Defining the Send Sequence

You have the option of defining the sequence in which a process sends received messages. In doing so, you can specify that the process must wait for an acknowledgement from the receiver each time that it sends a message.

If the messages are sent from different business systems, it is only possible to serialize them by using an integration process. However, if the messages all come from just **one** business system, check whether it would not be better if this business system executes the serialization.

You can define a serialization in different ways. The following table shows the various options. You can find the examples in the Integration Repository under *SAP Basis* → *SAP Basis 6.40*, namespace *http://sap.com/xi/XI/System/Patterns*.

Serialization	Description	Example
Start Message	The process is started when a particular message is received	BpmPatternSerializeOneTrigger
Multiple Start Messages	Different messages can start the process	BpmPatternSerializeMultipleTrigger

Start Message

Three receive steps are defined in this example process. The corresponding messages are received in the container elements `FirstMessage`, `SecondMessage`, or `ThirdMessage`.

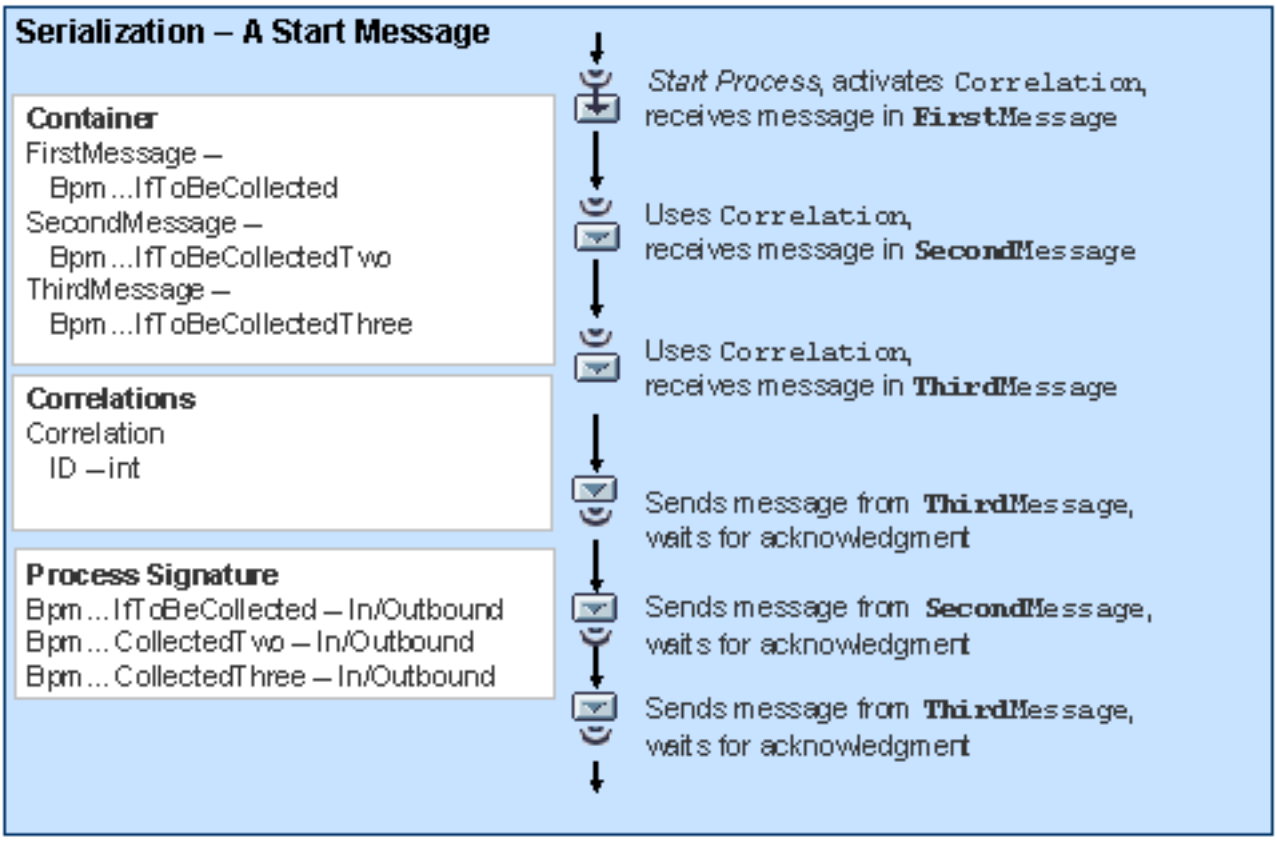
The process starts once the first receive step receives the message. For this reason, the *Start Process* indicator has been set for the first receive step. When a message is received, the receive step activates the correlation `Correlation`. This correlation is used by both subsequent receive steps and correlates the messages by means of an ID.

Note that you must ensure that the message of the first receive step – `FirstMessage` - really is the first message to arrive. If, for example, `SecondMessage` arrives first, it cannot be assigned to the process.

Once all three messages have been received, the process sends the messages in reverse order. Once it has sent a message, the process waits for the acknowledgment from the receiver to arrive before sending the next message.

All three abstract interfaces are used as inbound and outbound interfaces (see also: Process Signature).

The following graphic illustrates the process definition:



Different Start Messages

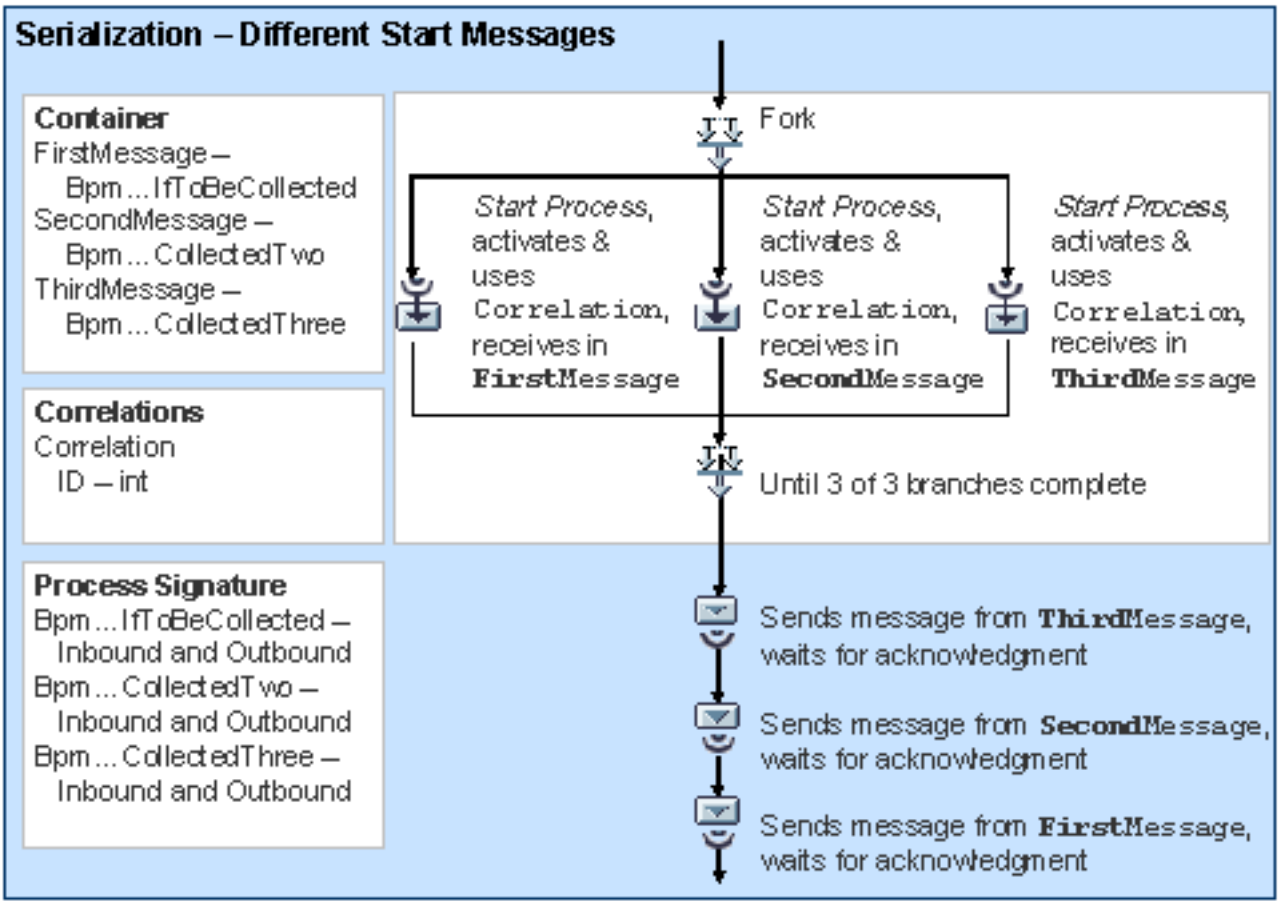
Three receive steps are defined in this example process. The corresponding messages are received in the container elements `FirstMessage`, `SecondMessage`, or `ThirdMessage`.

Any of the three messages can start the process. For this reason, the *Start Process* indicator has been set for each of the receive steps. At design time it is not known which of the messages will arrive first. Therefore, the receive steps are arranged in a fork. If one of the receive step receives its message, it starts the process and activates the correlation `Correlation`. This correlation is used by both other receive steps. The fork should be complete once all three messages have been received. Therefore, the number of required branches is set to three.

Finally, the process sends the received messages in the specified sequence. Each send step waits for the corresponding acknowledgment once it has sent its message.

All three abstract interfaces are used as inbound and outbound interfaces (see also: Process Signature).

The following graphic illustrates the process definition:


Serialization – Different Start Messages
Container
FirstMessage –
Bpm...IfToBeCollected
SecondMessage –
Bpm...CollectedTwo
ThirdMessage –
Bpm...CollectedThree
Correlations
Correlation
ID – int
Process Signature
Bpm...IfToBeCollected –
Inbound and Outbound
Bpm...CollectedTwo –
Inbound and Outbound
Bpm...CollectedThree –
Inbound and Outbound
Fork
Start Process, activates & uses Correlation, receives in FirstMessage
Start Process, activates & uses Correlation, receives in SecondMessage
Start Process, activates & uses Correlation, receives in ThirdMessage
Until 3 of 3 branches complete
Sends message from ThirdMessage, waits for acknowledgment
Sends message from SecondMessage, waits for acknowledgment
Sends message from FirstMessage, waits for acknowledgment

Example: Collecting and Bundling Messages - One Interface

You have the option of collecting multiple messages for an interface and bundling them into one message, for example, individual purchase order items into one purchase order. To do so, you need to define a receiver step within a loop. The loop can finish in a variety of ways.

The following table shows the various options. You can find the examples in the Integration Repository under *SAP Basis* → *SAP Basis 6.40*, namespace *http://sap.com/xi/XI/System/Patterns*.

Collect and Bundle	Description	Example
Payload-Dependent	The loop finishes when the number of messages received matches the number specified in the payload of the messages.	BpmPatternCollectPayload
Time-Dependent	The loop finishes at a specified point in time.	BpmPatternCollectTime
Message-Dependent	The loop finishes when a specific message is received.	BpmPatternCollectMessage

Example Processes

The example processes receive messages in a loop. The first message received starts the process and activates the correlation `Correlation` by using an ID. Each subsequent message uses this correlation. The messages are received in the container element `CollectMessage`.In the loop the received messages are attached to the multiline container element `CollectMessageList`.The following examples show the different options you have for defining the loop.

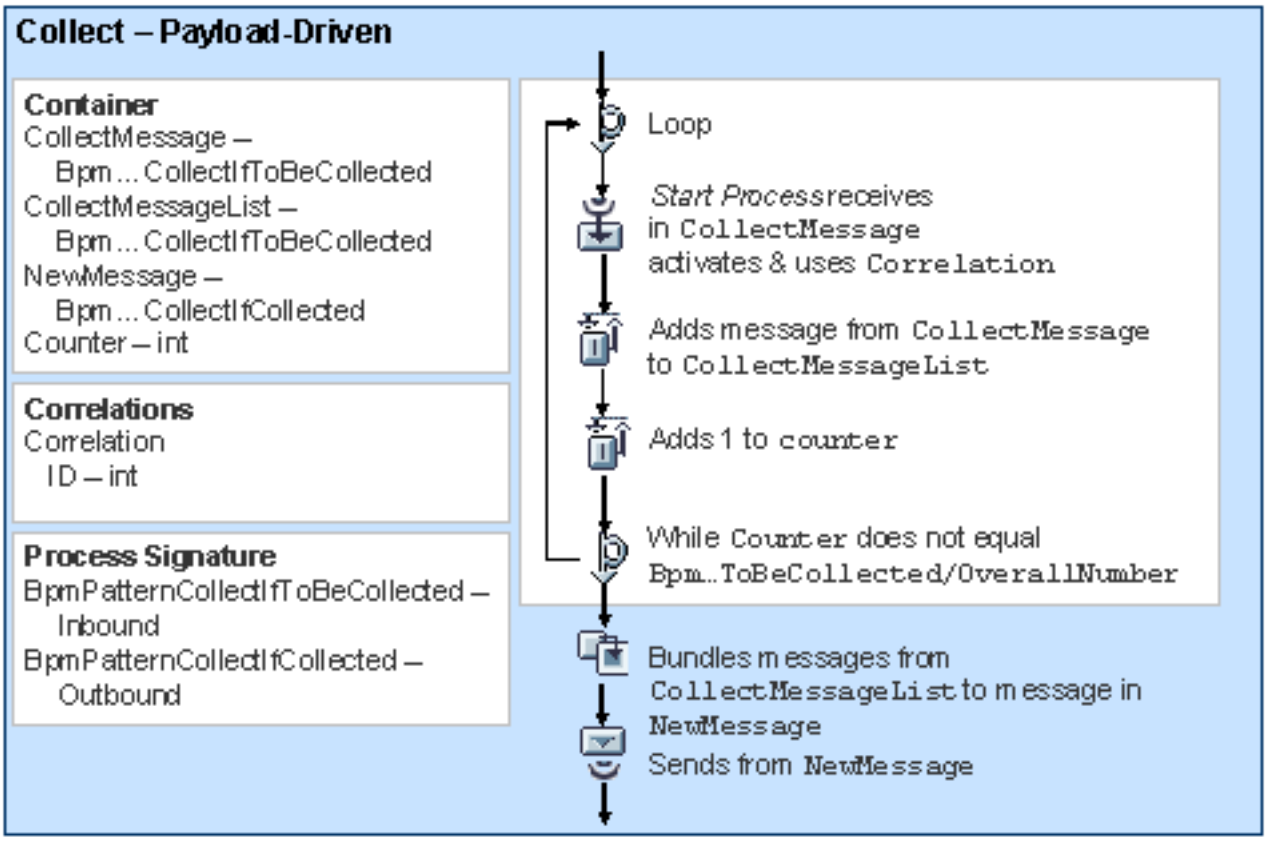
Once all the messages have been received, a transformation step bundles the messages collected in the container element `CollectMessageList` and from them creates a message in the container element `NewMessage`.This message is then sent by a subsequent send step.

The processes use one of the abstract interfaces as an inbound interface, and the other as an outbound interface (see also: Process Signature).

Payload-Dependent Loop

Each message that is received in the loop contains a number in the payload, which corresponds to the total number of messages to be received.In the loop, a container operation counts the counter `Counter` for the number of messages received. The loop continues to run while the number of messages received is not equal to the number of messages to be received.

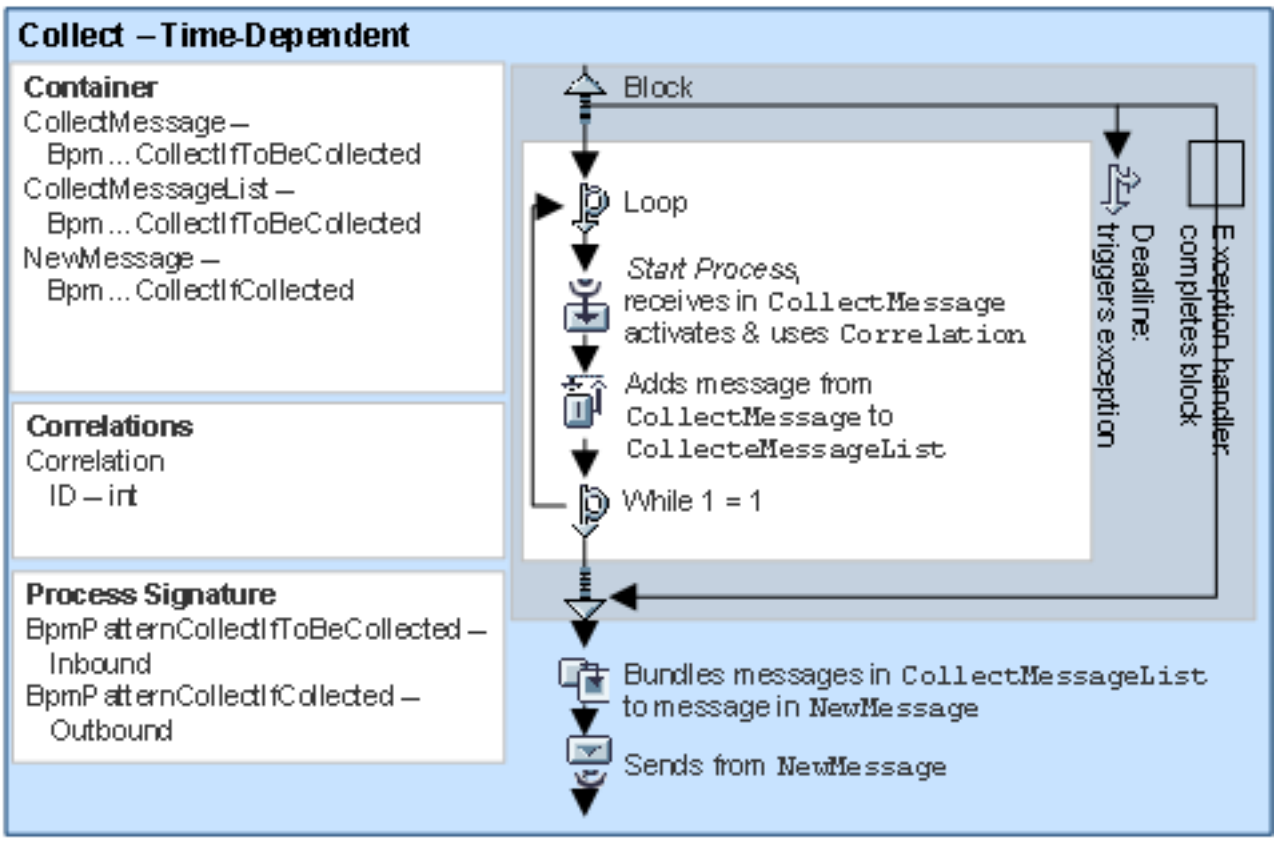
The following graphic illustrates the process definition:



Time-Dependent Loop

The loop for receiving the messages is defined as an infinite loop.The infinite loop forms a branch within a block. A deadline has been defined for the block. When the deadline is reached, the process is diverted into the designated branch.A control step then triggers an exception in this branch. The relevant exception handler ends the block (normal completion, no error status).

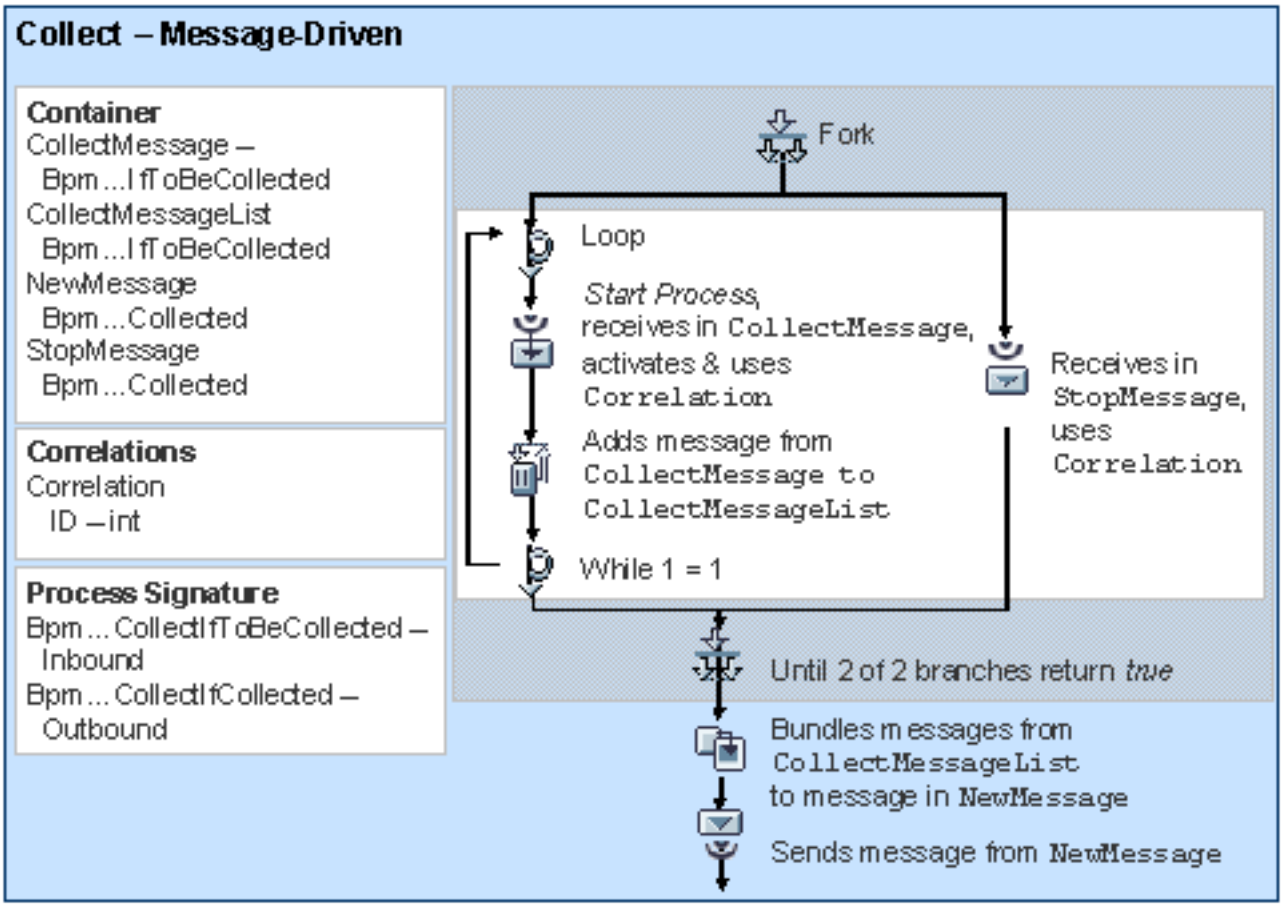
The following graphic illustrates the process definition:



Message-Dependent Loop

The loop for receiving the messages is defined as an infinite loop.The infinite loop forms a branch within a fork. A receive step is defined in a parallel branch to receive the message that ends the process.The fork is complete when both branches return *true*. However, since the infinite loop always returns *true*, the fork is only complete when the message that ends the process is received.

The following graphic illustrates the process definition:


Collect – Message-Driven
Container
CollectMessage –
Bpm...IfToBeCollected
CollectMessageList
Bpm...IfToBeCollected
NewMessage
Bpm...Collected
StopMessage
Bpm...Collected
Correlations
Correlation
ID – int
Process Signature
Bpm...CollectIfToBeCollected – Inbound
Bpm...CollectIfCollected – Outbound
Fork
Loop
Start Process, receives in CollectMessage, activates & uses Correlation
Adds message from CollectMessage to CollectMessageList
While 1 = 1
Receives in StopMessage, uses Correlation
Until 2 of 2 branches return true
Bundles messages from CollectMessageList to message in NewMessage
Sends message from NewMessage

Example: Collecting and Bundling Messages - Multiple Interfaces

You have the option of collecting and bundling messages from different interfaces. To do so, define the corresponding receive step in a fork.

You can define the collecting of messages in different ways. The following table shows the various options. You can find the examples in the Integration Repository under *SAP Basis* → *SAP Basis 6.40*, namespace *http://sap.com/xi/XI/System/Patterns*.

Collect	Description	Example
Collect all messages	Collects all messages	BpmPatternCollectMultiIf
Collect particular messages only	Ends collecting when a specified condition is fulfilled	BpmPatternCollectMultiIfCondition

Example Processes

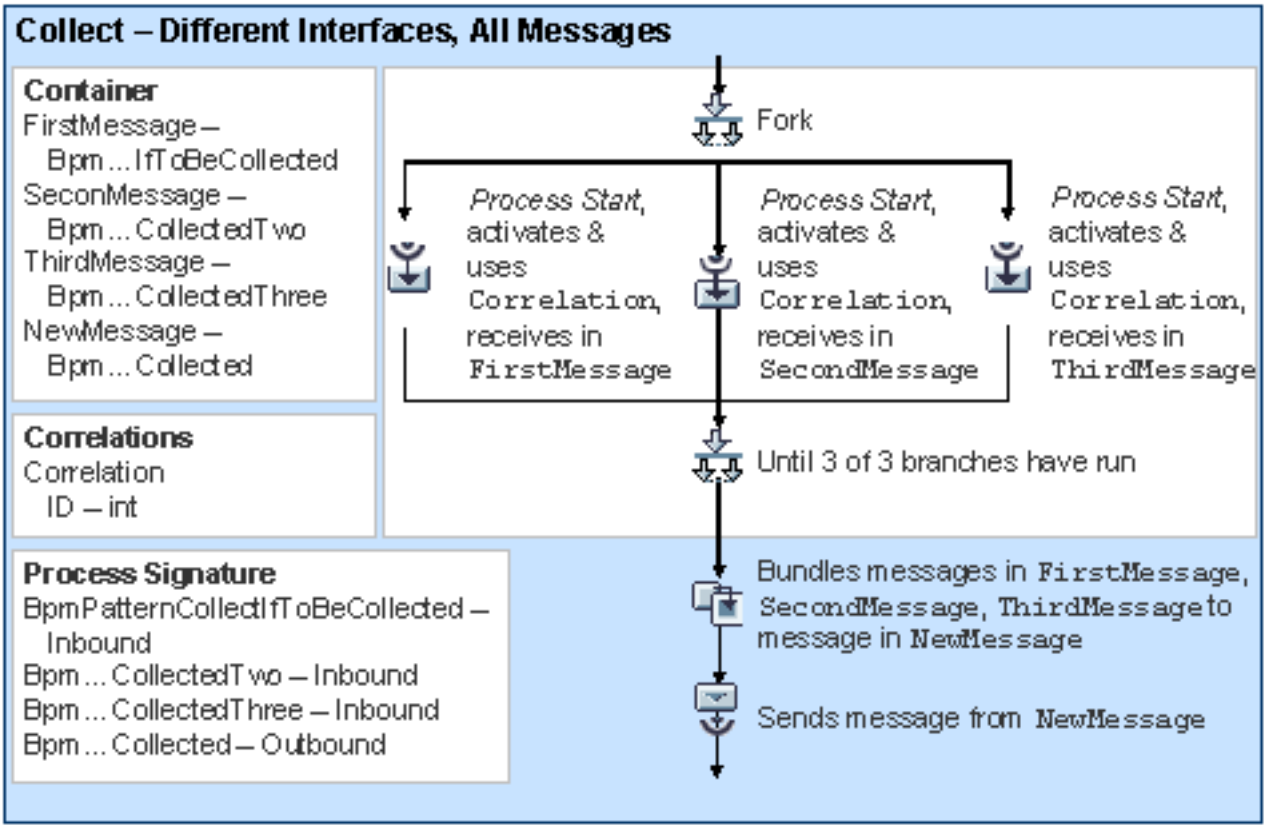
The example processes receive three messages from different interfaces in the container elements `FirstMessage`, `SecondMessage` and `ThirdMessage`.Three receive steps have been defined correspondingly within a fork. Any of the receive steps can start the process. For this reason, the *Start Process* indicator has been set for each of the receive steps. The first message received starts the process and activates the correlation `Correlation`. Each receive step uses this correlation. Once the messages have been received, a transformation step bundles the messages together and from them creates a message in the container element `NewMessage`. This message is then sent by a subsequent send step.

The processes use three of the abstract interfaces as inbound interfaces, and one as an outbound interface (see also: Process Signature).

Collect all messages

To collect all messages, define that the fork is complete once all branches have been processed.

The following graphic illustrates the process definition:



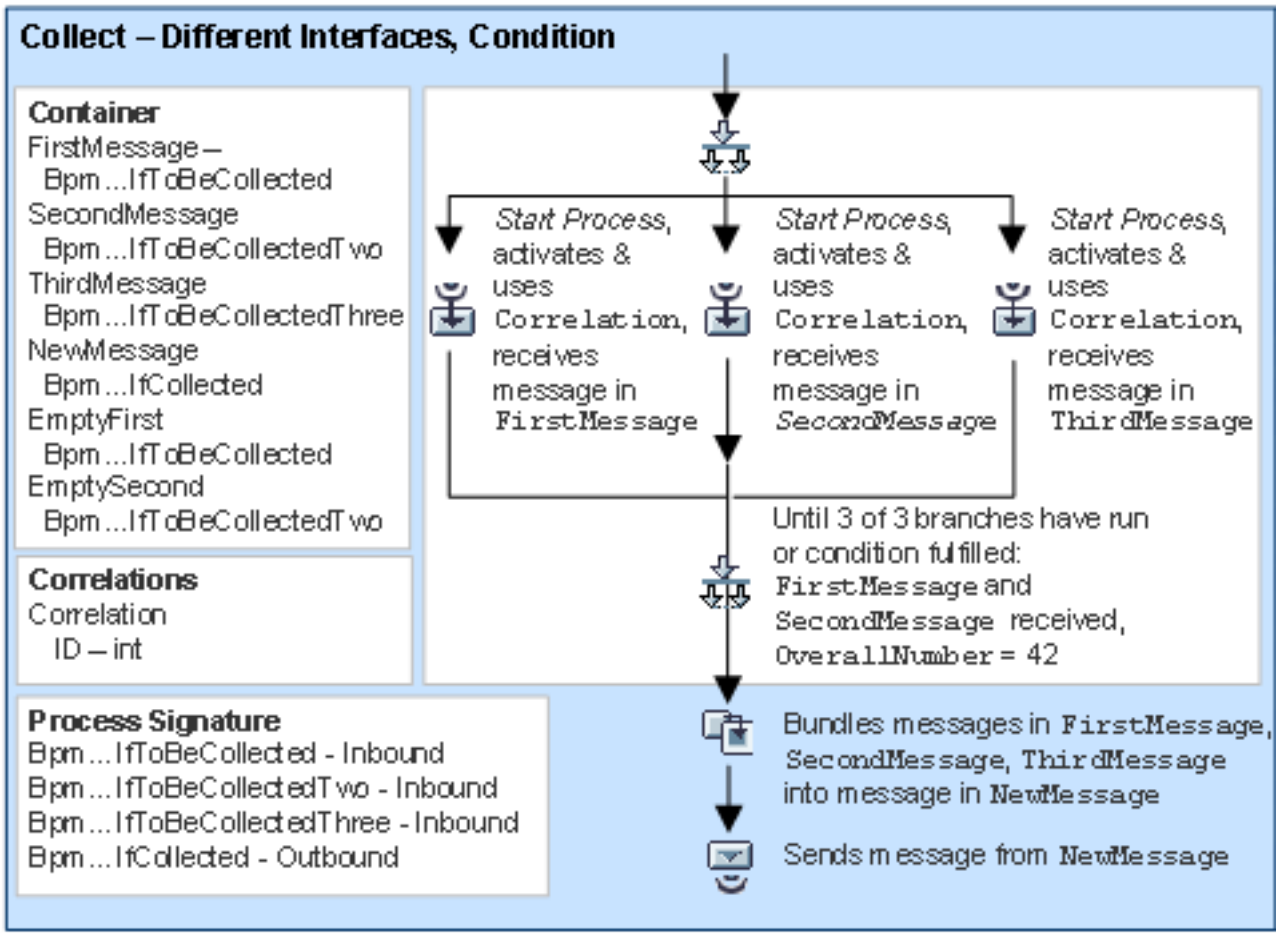
Collect Messages By Using a Condition

You can also specify that the collecting of messages stops when a particular condition is fulfilled. Define a relevant end condition for the fork for this purpose. In the following example, the fork is complete once one of the following conditions returns *true*:

- All required branches are complete. Three messages were received
- The end condition has been fulfilled. The relevant messages have been received in the container elements `FirstMessage` and `SecondMessage` and the `OverallNumber` element in message `FirstMessage` has the value `42`. To check whether the messages were received, both container elements are compared with an empty container element (`EmptyFirst`, `EmptySecond`).

The system checks the conditions in the sequence that you specified.

The following graphic illustrates the process definition:


Collect – Different Interfaces, Condition
Container
FirstMessage –
Bpm...IfToBeCollected
SecondMessage
Bpm...IfToBeCollectedTwo
ThirdMessage
Bpm...IfToBeCollectedThree
NewMessage
Bpm...IfCollected
EmptyFirst
Bpm...IfToBeCollected
EmptySecond
Bpm...IfToBeCollectedTwo
Correlations
Correlation
ID – int
Process Signature
Bpm...IfToBeCollected - Inbound
Bpm...IfToBeCollectedTwo - Inbound
Bpm...IfToBeCollectedThree - Inbound
Bpm...IfCollected - Outbound
Start Process, activates & uses Correlation, receives message in FirstMessage
Start Process, activates & uses Correlation, receives message in SecondMessage
Start Process, activates & uses Correlation, receives message in ThirdMessage
Until 3 of 3 branches have run or condition fulfilled: FirstMessage and SecondMessage received, OverallNumber = 42
Bundles messages in FirstMessage, SecondMessage, ThirdMessage into message in NewMessage
Sends message from NewMessage

Example: Sync/Async Communication

You want a synchronously calling business system BS-S and an asynchronously called business system BS-AS to communicate with each other. For this purpose you must define a sync/async bridge.

You can find the examples in the Integration Repository under *SAP Basis* → *SAP Basis 6.40*, namespace *http://sap.com/xi/XI/System/Patterns* under *BpmPatternSyncAsyncBridge*.

You can only define **one** sync/async bridge for each integration process. This comprises at least the following steps:

- Receive Step *SyncReceive*

  Receives the *Request* message from the synchronously calling business system BS-S and opens the sync/async bridge.

  The receive step is the first step in the integration process. In the receive step you specify the synchronous interface *BpmPatternBridgeSyncIf* for receiving the message from the synchronously calling business system. The integration process is started when the message is received. The message type of the message to be received and the request message from the synchronous interface must be identical.

- Send Step *AsyncSend*

  Sends the received *Request* message asynchronously to the asynchronously called business system BS-AS.

- Receive Step *AsyncReceive*

  Receives the *Response* message from the asynchronously called business system BS-AS.

- Send Step *SyncSend*

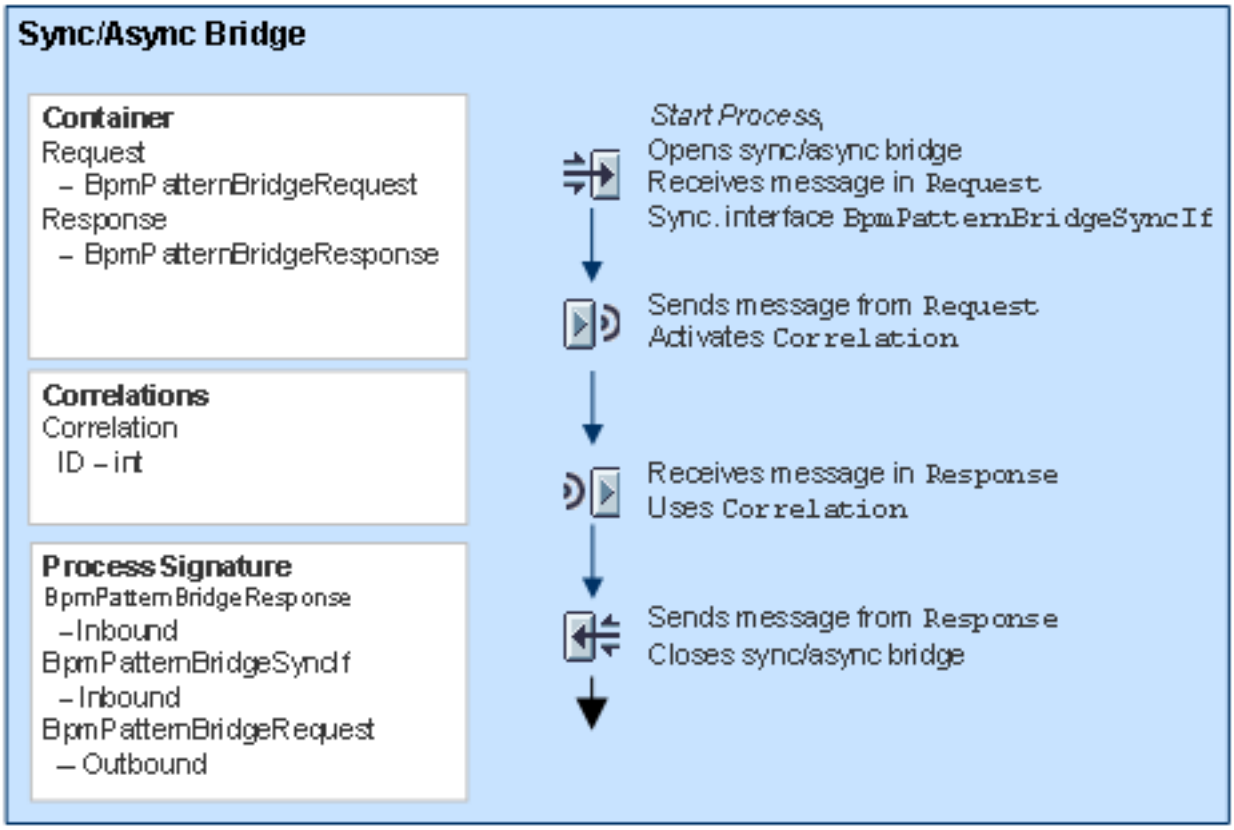
  Sends the *Response* message from the asynchronously called business system BS-AS synchronously to the synchronously calling business system BS-S and closes the sync/async bridge.

  The message type of the message to be sent and that of the reply message from the synchronous interface in the opening receive step *SyncReceive* must be identical. In the send step, enter the name of the receive step that opened the sync/async bridge (in this example *SyncReceive*).

  If you insert additional steps in the sync/async bridge, the synchronous time is increased correspondingly. If the time period until the sync/async bridge is closed is too long, this can lead to problems.

The following graphic illustrates the definition of the integration process:



Message Interfaces

The integration process requires the following message interfaces:

- `BpmPatternBridgeSyncIf`

  The integration process uses this synchronous, abstract message interface to receive the message from the synchronous business system. This synchronous message interface is mapped to an asynchronous, abstract message interface by using the sync/async bridge.

- `BpmPatternBridgeRequest`

  The integration process uses this asynchronous, abstract message interface to send the message from the asynchronous business system.

- `BpmPatternBridgeResponse`

  The integration process uses this asynchronous, abstract message interface to receive the message from the asynchronous business system.

Example: Sending Synchronously to Multiple Receivers

You want to send a request message synchronously to multiple receivers. The **first** message received is to be used as the reply message. All messages that are received after this message are ignored.

For this purpose, define a fork and insert one branch for each receiver. For each branch, insert a synchronous send step and define a send context. You use this send context in the receiver determination in the Integration Directory. Define the end condition for the fork in such a way that one branch must have been executed for the fork to be completed.

Example: Deadline Monitoring for Receipt of a Response Message

It is often the case that a business system sends a request message (for example, a purchase order) and waits for a response message (for example, a response to a purchase order) from another business system.

If you want to monitor whether the response message is received within the predefined deadline, you must define an integration process for this particular message exchange. To monitor the deadline, you define a block with a deadline branch in the integration process.

You can define how the integration process reacts to deadline that is not met in different ways. The following table shows the various options. You can find the examples in the Integration Repository under *SAP Basis* → *SAP Basis 6.40*, namespace *http://sap.com/xi/XI/System/Patterns*.

Reaction	Description	Example
Alert	When a deadline is not met, an alert is triggered and the process continues to wait for the response message to be received.	BpmPatternReqRespAlert
Terminate with Error Message	When a deadline is not met, an exception is thrown. An error message is created in the relevant exception handler and then sent.	BpmPatternReqRespTimeOut

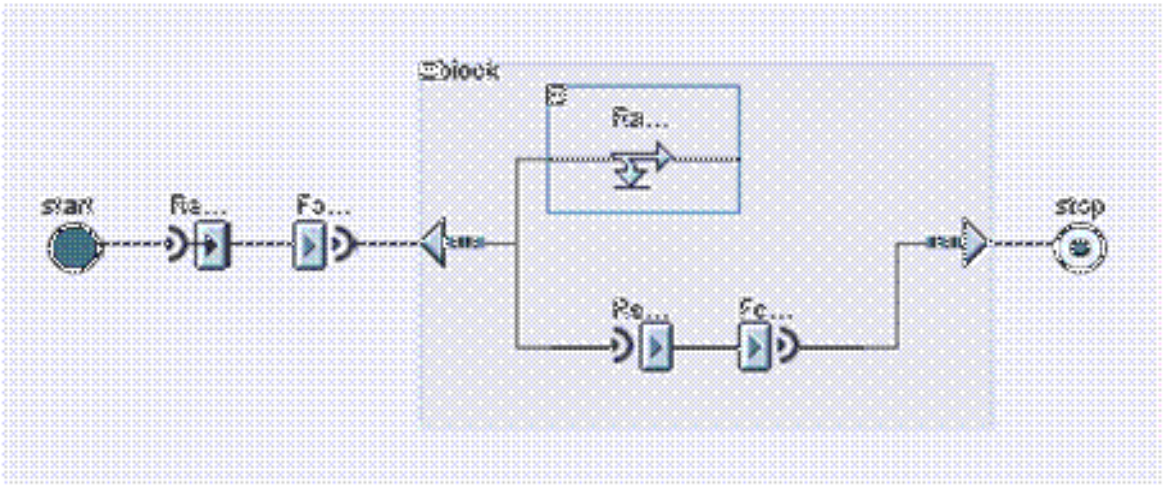
Example Process

The first receive step receives the request message, starts the process, and activates the correlation *Correlation*. The correlation links the request and response messages by means of an ID (for example, a purchase order number). The subsequent send step sends the message according to the receiver determination configured in the Integration Directory.

The receive step for receiving the response message uses the correlation *Correlation*. A subsequent send step then sends the response message according to the receiver determination configured in the Integration Directory. To be able to define deadline monitoring for both steps, they were defined within a block with a deadline branch.

Create Alert when Deadline is Not Met

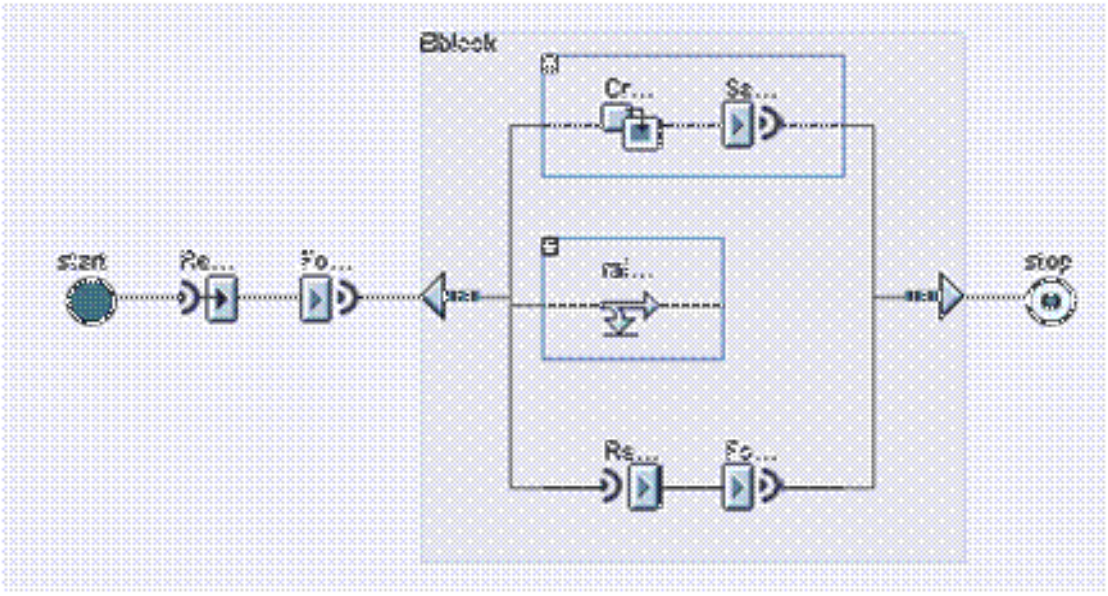
The following graphic illustrates the process definition:



If the block is not finished 24 hours after it was created, the deadline branch is executed. In the example, the control step in the deadline branch triggers an alert. The receive step continues to wait for the response message to be received.

Terminate with Error Message when Deadline is Not Met

The following graphic illustrates the process definition:



If the block is not finished 24 hours after it was created, the deadline branch is executed. In the example, the control step in the deadline branch triggers an exception. Processing continues in the relevant exception handler branch. In the exception handler branch, a transformation step creates an error message. The error message contains the ID from the request message with which the correlation *Correlation* was activated. If the ID is a purchase order number, for example, you can see which purchase order number has not yet received a purchase order response.

A subsequent send step then sends the error message according to the receiver determination configured in the Integration Directory. The block is now fully processed.

Checklist: Making Correct Use of Correlations

Correlations Using Payload Fields

The fields that you define by using a correlation must be available in the **payload** of the messages involved.

Uniqueness

Ensure that you always define correlations uniquely. If, for example, multiple factories use the same material number, a correlation that uses the material number alone is not sufficient to uniquely determine all the messages that belong together. In this case you need to define the correlation by using the material number and the factory.

Validity

Ensure that you always define the validity of correlations correctly. For each correlation, check whether it can be defined as a local correlation. A local correlation is only valid for the block for which it is defined. Once the block has been processed, the correlation is no longer active.

By using a local correlation you can stop messages from being assigned further by a correlation that should actually no longer be active. Moreover, local correlations enable you to make it easier for others to follow how messages are processed.

Checklist: Making Correct Use of a ParForEach

You have the option of choosing the mode *ParForEach* (Parallel For Each) for a block. The following considerations will help you to decide whether this mode is appropriate for your particular use case.

A ParForEach is **not** a means of improving performance. A ParForEach will **not** result in a split into threads.

ParForEach Recommended

A ParForEach is recommended if:

- The messages are processed further in the parallel processing branches.
- The individual processing branches are not dependent on each other on a business level.
- There are less than 999 parallel processing branches.

  This means that the multiline table element that is processed in the ParForEach does not contain more than 999 lines.

A ParForEach would be beneficial in the following example:

You want to send a message to multiple receivers, wait for the respective answers, and then edit each one independently of the others in a separate context. Once the messages have been sent asynchronously in the ParForEach, the step can wait for an acknowledgment or some kind of business confirmation, for example.

ParForEach Not Recommended

A ParForEach is **not** recommended if:

- The messages are not processed further in the processing branches.

  If, for example, you want to send a message to multiple receivers, but do not want the step to wait for a response or acknowledgment, use a ForEach instead of a ParForEach.

- There are more than 999 processing branches.

  Check whether you can use a ForEach instead of a ParForEach. In the case of a ForEach you can set the maximum number of processing branches as part of the absolute upper limit (transaction SWPA). The absolute upper limit is set to 999999.

Checklist: Making Correct Use of Mappings

In a transformation step, only use those mappings that cannot be realized by the Integration Server alone: 1:n and n:1 mappings that require the integration process as the context.

For each mapping, check whether it can be executed before or after the integration process. For example, it is possible to replace a mapping that is executed before a sender step with a mapping in the receiver determination.