Real World BPM Patterns

Jim Ladd
Wazee Group, Inc.

November 13, 2018

Contents

Wazee Group
Software Construction & Renovation

Introduction

During a client engagement a few years ago, several patterns in the Business Process Management (BPM) process models where identified. Once a pattern emerged, it was quickly added to our BPM analysis/design language. It was common to hear "It's like the Record and Report process." or "That's another Simple Approval model". The actual process model files were commonly CPE (copy, past, and edit) and resulted in new development that was completed in hours instead of days or weeks.

The objective of this document is that someone new to BPM modeling can realize the same benefits that my client did without the same learning curve.

Here's a brief description of the process patterns:

- *Scheduled* – This is a trivial pattern that emulates a batch process. The objective is to explore the BPM tool's capability and ease of scheduling the execution of process models in a periodic manner. In my experience, the BPM tool and models are easier to develop, execute, and monitor "batch" processing than my clients' other options.

- *Notification* – One of the client's system did not support database row locking. Anyone with the proper credentials (and there are several in a large organization) could update columns in critical rows of data like a product template. To compensate for the system's lack of control, I developed a process model to notify key personnel when certain types of data were updated.

- *Record And Report* – Once the notification type of emails became known by the business communities, requests were submitted for notifications that quickly overwhelmed the email inboxes. These requests evolved into a daily report that contained the changes to certain data over the past 24 hours. This pattern consists of two process models, one to record the events in real time and then a report model that queries the data store for all of those events since the last time the report was generated.

- *Simple Approval* – This is another trivial model that every BPM tool should support without issue. I explicitly named this model to distinguish it from the more complex Multi-Level Approval model.

- *Multi-Level Approval* – This model is used when multiple rounds of approval or review are required. I used this pattern for a purchase order (PO) approval process where multiple levels of the organization chart had to review the PO depending on the department, the PO type, and the amount. The insights to this model is to understand the limitations of the BPM tool. This pattern uses two external database tables for specifying the financial approval authority of each role in the process and a table for maintaining who could fulfill those roles.

- *Parallel Cancel* – This pattern addresses the scenario for easily terminating long running process instances. For example, in the PO approval process described above, this pattern allowed the person who submitted the PO for approval to cancel the process.

- *Singleton* – This pattern snippet prevents process instances from executing on the same data at the same time when that scenario is not desired. In the purchase order process, the business required only one approval instance to be executing for a given PO number.

- *Wrapper* – The wrapper pattern is used in situations where different event formats are received by the BPM tool but are intended for the same process model. This pattern was used when an interim event format was used with the intent of changing to the final, different format at some point in the future.

Background

The BPM system that served as the platform for this document was deployed in a fairly traditional fashion. As events are generated by other systems in the enterprise, the BPM receives those events of interest and instantiate the appropriate process model. These process instances would then run to completion. A diagram of the reference interface architecture is shown below.

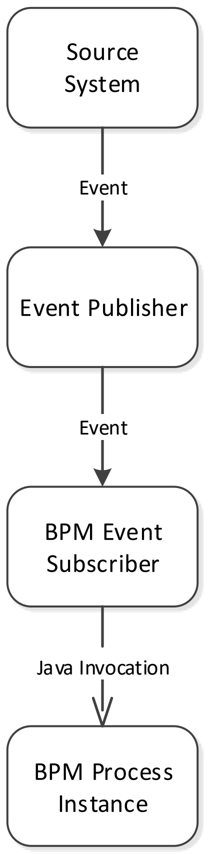


Figure 1 - Reference interface architecture

The format of the XML-based events is straightforward. There is some "header" information that is included with all event types and a set of event data that is specific to the event type. One unique characteristic of this implementation is that each data element included both the "old" and "new" values. Similar to most database triggers, the "old" value represented the state of the attribute before the transaction and the "new" value represents the state after the transaction. The XML schema of the event is shown below:

```
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
             xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="EventData">
    <xs:complexType>
```

```
        <xs:sequence>
          <xs:element type="xs:string" name="Publisher"/>
          <xs:element type="xs:string" name="EventName"/>
          <xs:element type="xs:string" name="Operation"/>
          <xs:element type="xs:string" name="EventId"/>
          <xs:element type="xs:dateTime" name="SentTimestamp"/>
          <xs:element name="Document">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="ElementData"
                            maxOccurs="unbounded"
                            minOccurs="0">
                  <xs:complexType>
                    <xs:sequence>
                      <xs:element type="xs:string" name="Name"/>
                      <xs:element type="xs:string" name="Value"/>
                      <xs:element type="xs:string"
                                  name="OldValue"
                                  minOccurs="0"/>
                    </xs:sequence>
                  </xs:complexType>
                </xs:element>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
```

An example of the XML-based message is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<EventData>
    <Publisher>EventPublisher</Publisher>
    <EventName>PRODUCT_CHANGE</EventName>
    <Operation>UPDATE</Operation>
    <EventId>8c0bdb7a-48f3-4924-8e00</EventId>
    <SentTimestamp>2015-08-05T16:07:48.106-06:00</SentTimestamp>
    <Document>
        <ElementData>
            <Name>currentService</Name>
            <Value>PRODUCT_CATALOG</Value>
        </ElementData>
        <ElementData>
            <Name>startTimeMillis</Name>
            <Value>1375733074864</Value>
        </ElementData>
        <ElementData>
            <Name>owner</Name>
            <Value>JLADD</Value>
        </ElementData>
        <ElementData>
            <Name>startTime</Name>
            <Value>2013-08-05 14:04:34</Value>
        </ElementData>
        <ElementData>
            <Name>sessionID</Name>
            <Value>10.0.128.150:64883_2-0</Value>
        </ElementData>
```

```
<ElementData>
    <Name>environment</Name>
    <Value>DEV</Value>
</ElementData>
<ElementData>
    <Name>version</Name>
    <Value>07.05.01</Value>
</ElementData>
<ElementData>
    <Name>ProductId</Name>
    <Value>12345</Value>
    <OldValue>12345</OldValue>
</ElementData>
<ElementData>
    <Name>STATE</Name>
    <Value>INACTIVE</Value>
    <OldValue>ACTIVE</OldValue>
</ElementData>
```

Since this document focuses on process model patterns, these patterns should be applicable regardless of the BPM software used to implement them. To avoid any BPM vendor dependencies, a generic set of icons were created to represent the building blocks of the process patterns. These icons are shown below:

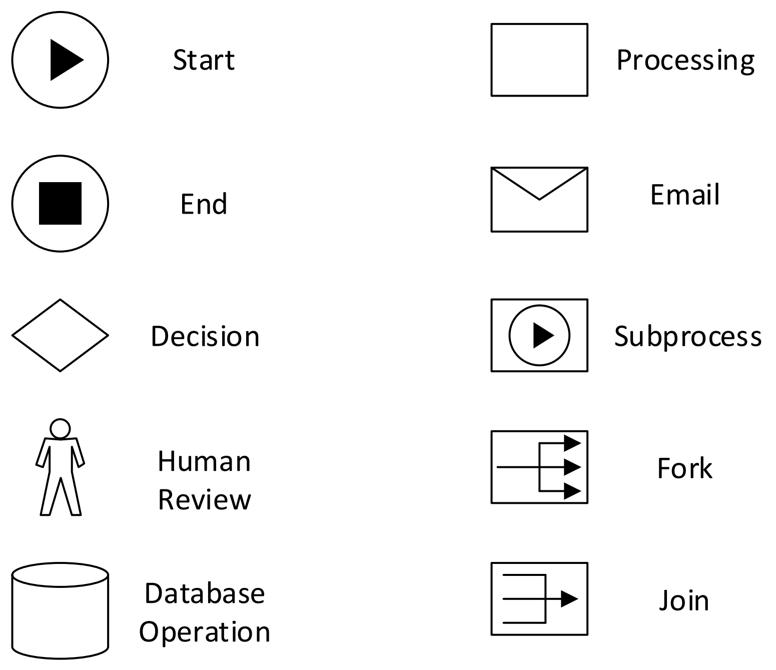


Figure 2 - Legend

Brief descriptions of this icons are presented below:

- *Start* – This node is the beginning of the process model execution. Only a single path can originate from this node. A process model must have one and only one Start node.

- *End* – This is the last node of the process model. Multiple nodes may have paths that terminate at the End node. A process model must have one and only one End node. A process model instance can be terminated (i.e. cancelled) at any time by an external entity with execution halted immediately.

- *Decision* – This node represents a decision point in the process path. The Decision node can have multiple inbound paths and multiple outbound paths.

- *Human Review* – The Human Review node is used to represent an interface for a user to interact with the process model. Typically, an email is sent to one or more designated users with one or more possible "actions" provided. The user may select a link to "approve" the request or a link to "reject" the request.

- *Database Operation* – This node represents common database actions to a database. These transactions include retrieving data from the BPM's internal database and reference tables from external databases.

- *Processing* – The Processing node is truly an abstraction and represents a wide range of functionality. This node can represent an interface to an external system and embedded code within the process model.

- *Email* – This node represents the sending of an email to one or more recipients.

- *Subprocess* – The Subprocess node denotes the creation of another process model instance. Traditionally, these "child" process can be executed in a synchronous mode where the execution of the "parent" process is halted until the child process completes. The alternative is an asynchronous mode where the parent process creates the child process instance and then continues its execution independent of the child instance.

- *Fork* – The Fork node is used in place of an asynchronous subprocess. The Fork node has multiple outbound execution paths that execute independently.

- *Join* – This node is used with the Fork node and is responsible for "joining" the independent execution paths from a Fork node. There is only one outbound path from a Join node and will not be traversed until all inbound paths are executed up to the Join node.

Scheduled Process

The first and simplest of the process model patterns is the Scheduled. This pattern is commonly used for "batch processing" needs. Several times in my experience, the BPM tool facilitates the development, deployment, scheduling, and monitoring of the batch process models better than other tools used by the clients. A diagram of this model is shown below:

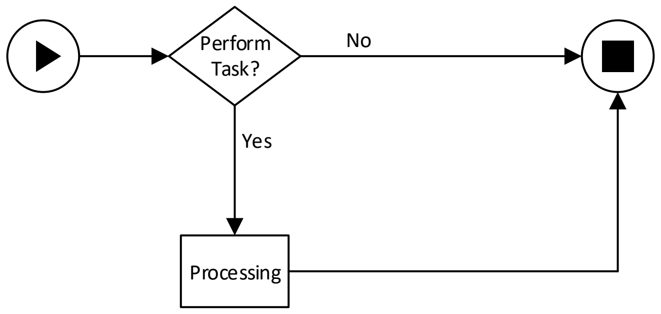


Figure 3 - Simple scheduled process

One of the best practices in BPM modeling is to consider preconditions in the model before transitioning to the major activities or processing steps. The range of preconditions is wide, diverse, and dependent on the business rules. It is best NOT to assume that only the appropriate events are used as inputs to the process instances.

A common improvement is to send an email notification containing the success status and/or statistics of the processing. One of the best practices regarding emails is to include the environment denoted so that recipients can determine quickly the origin of the email.

Another enhancement to the Scheduled model is to incorporate the Singleton pattern so multiple process instances are not running at the same time.

Notification Process

Due to a lack of control at the database row level in one of the source systems, the business owners could not restrict certain data to be modified. An alternative was to notify them when the restricted data was updated. This pattern is shown below:

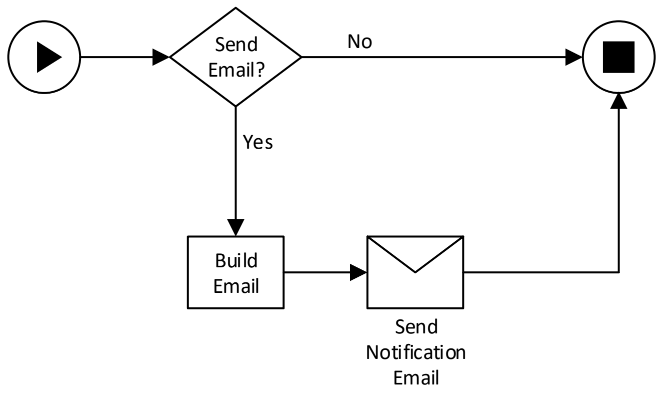


Figure 4 - Notification process

Each notification process model had preconditions to ensure that emails were not sent due to someone just looking at the data row. One source system would update the timestamp and send an event even though no data was actually changed.

The email notifications contained who updated the row, when it was changed, and the “old” and “new” values of each data element that was changed.

Record and Report Process

Soon after the first Notification-type processes were deployed, the business owners requested reports of the changes for certain data to be emailed on a daily basis. This solution consisted of two process models. The first was the “record” process where events were received and, if the criteria was met, saved in the database. The second model was a scheduled process that would query the database for all of those events received and stored since the time the previous report was generated. The two process models are shown below:

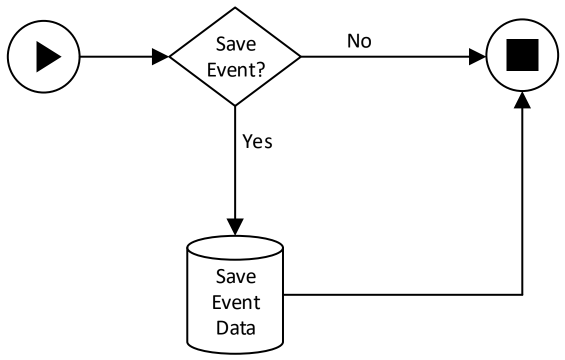


Figure 5 - Record process

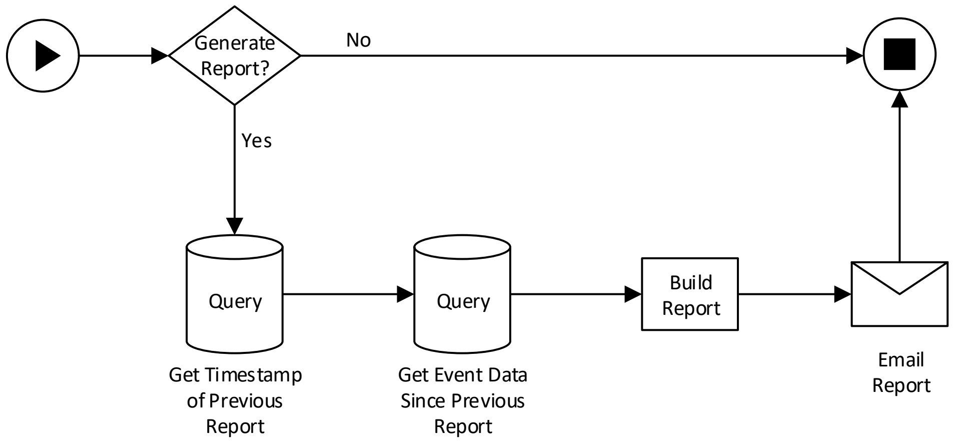

Generate Report?
No
Yes
Query
Get Timestamp of Previous Report
Query
Get Event Data Since Previous Report
Build Report
Email Report

Figure 6 - Report process

Simple Approval Process

The Simple Approval process model is one that every BPM tool should support. The only reason for including this model is to distinguish it from the more complex Multi-Level Approval pattern and to make our BPM design language more comprehensive.

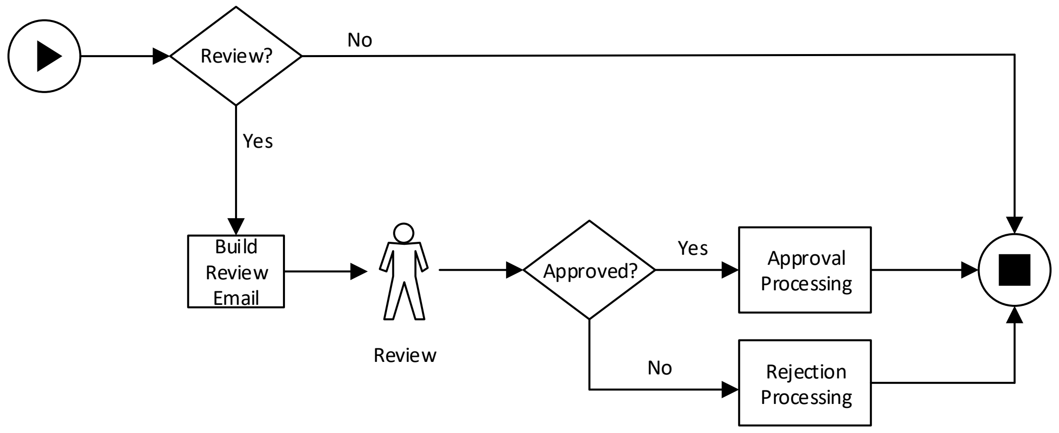


Figure 7 - Simple approval process

If this type of process model is long running, the modeler should consider enhancing it with the Parallel Cancel and Singleton patterns.

Wazee Group
Software Construction & Renovation

Multi-Level Approval Process

The Multi-Level Approval process allows a dynamic number of approval levels to participate in the process flow. This pattern relies on two external database tables to specify the number of levels, the criteria for each level, and the personnel who can participate at each level. The key concept is a loop in the process flow than continues until the criteria for a given level is not met (for example, the approval limit of a vice president is greater than the purchase order amount).

At any time an approver can reject the reviewed object. A rejection email notification is sent and the process instance is terminated. When a reviewer approves the object and that person's role has a higher limit than the reviewed object, an approved email is sent and the process instance is terminated.

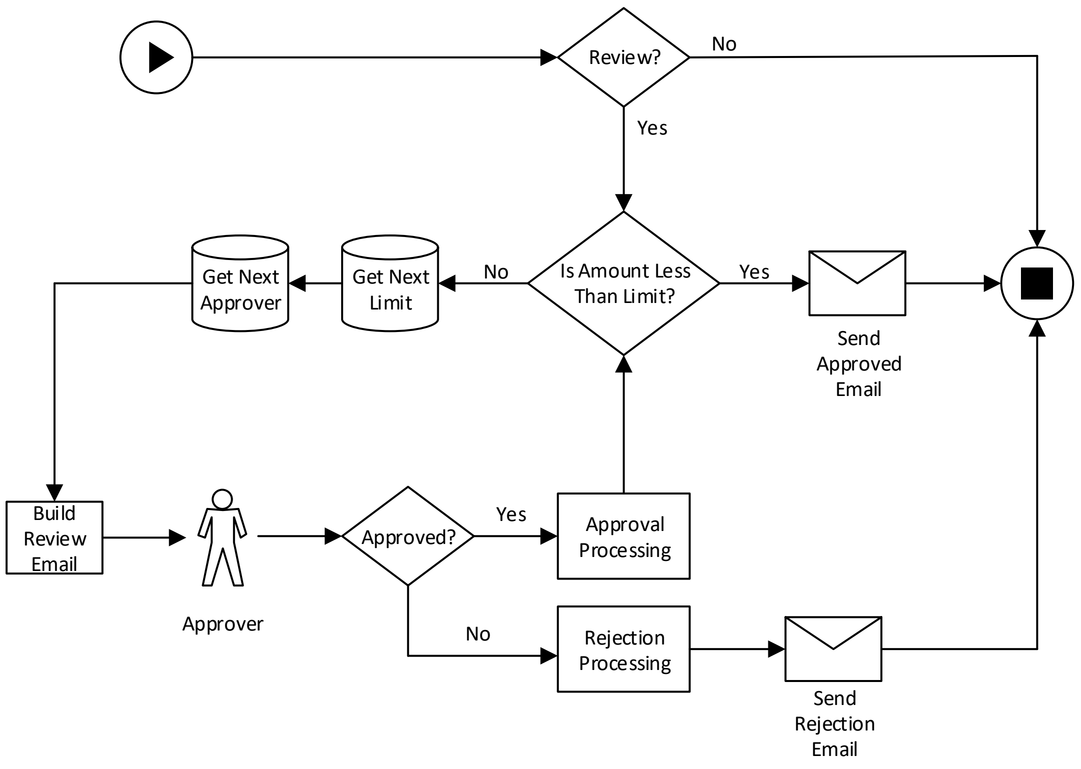


Figure 8 - Multi-level approval process

If this type of process model is long running, the modeler should consider enhancing it with the Parallel Cancel and Singleton patterns.

Parallel Cancel Process

Long running process instances can be beneficial for an organization and, at the same time, a pain to maintain. Many times, persons new to process modeling neglect the termination of process instances and rely on BPM administrators to manually cancel process instances. One technique to empower the end users provides a cancel option that is parallel to the main processing. In this scenario, when a user invokes a long running process instance, a request is sent to someone authorized to cancel the process. At any time this person approves of the cancellation, the original process is terminated.

There are two basic ways to achieve this type of behavior. The approach selected depends on the capabilities of the BPM tool. Most tools support the creating of a sub-process where one process (i.e. the parent) can invoke another process (i.e. the sub-process or child process). Typically, the option of synchronous or asynchronous creation is provided. The asynchronous mode must be used for this pattern to be successful. The next two diagrams show the asynchronous pattern.

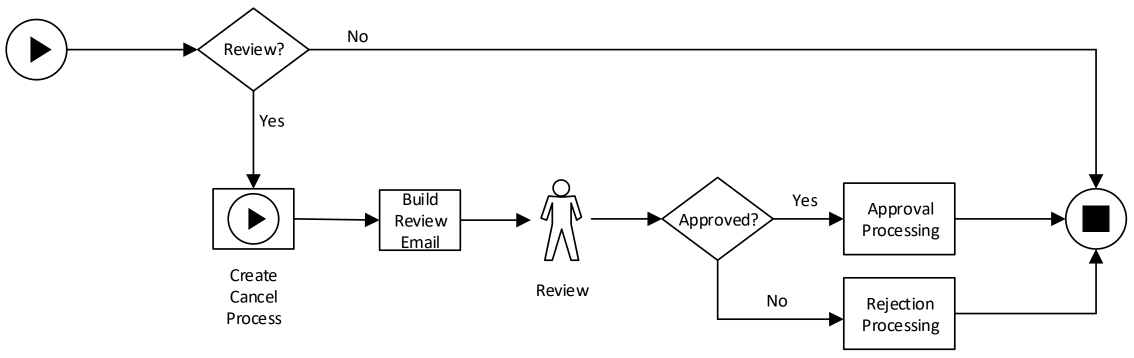


Figure 9 - Long running process

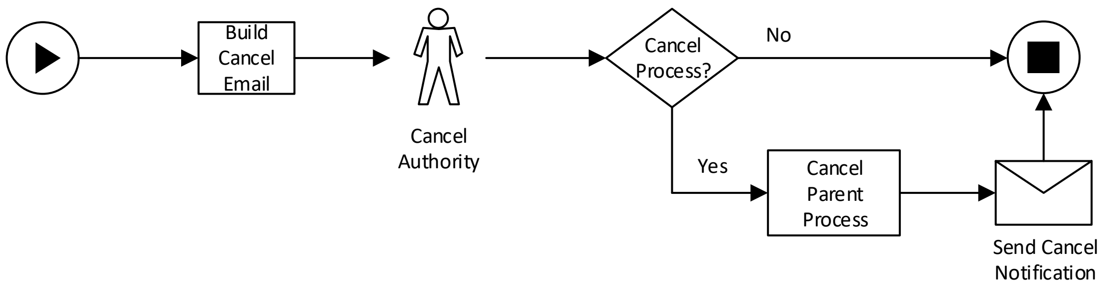


Figure 10 - Asynchronous cancel process

The other option is to use a “fork” and “join” features of the tool if provided. This feature supports parallel execution of paths within the same process model. This type of model is shown below.

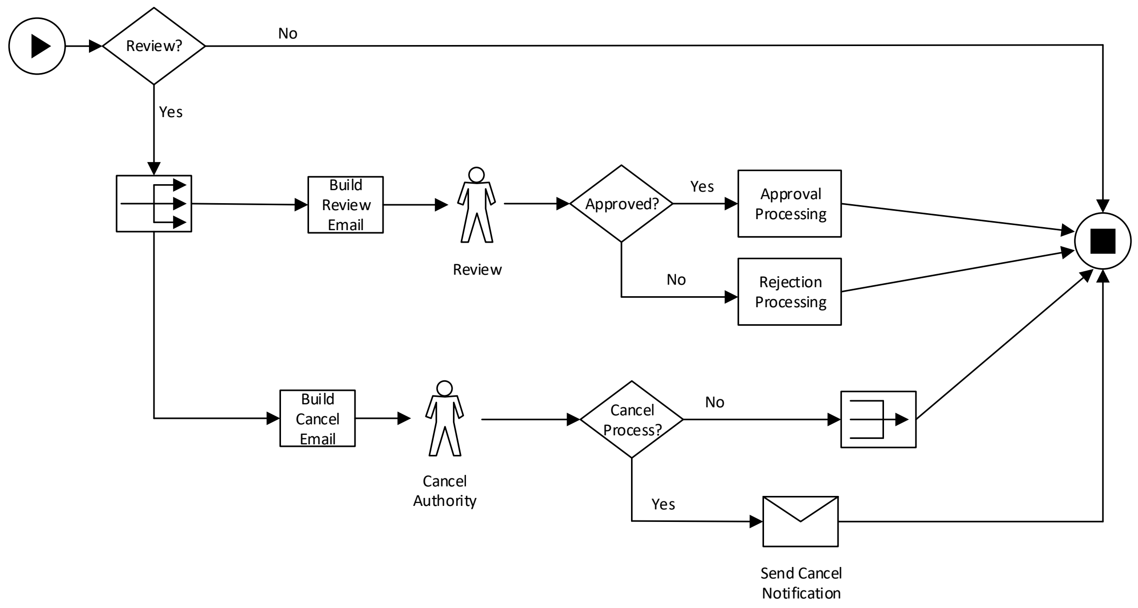
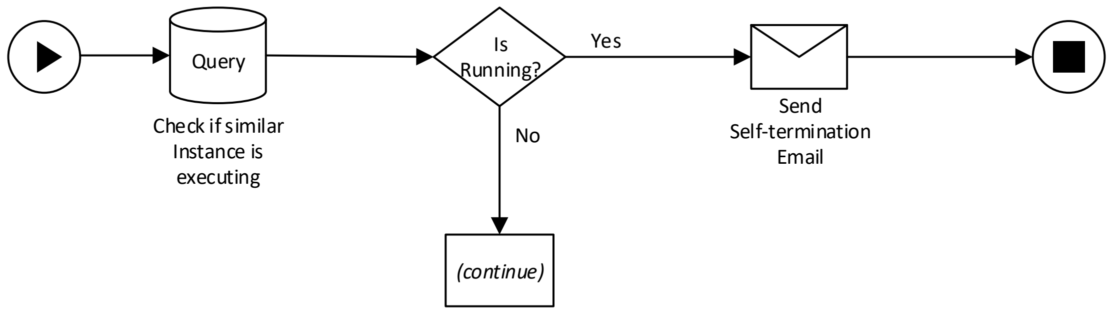


Figure 11 - Concurrent cancel process

Another consideration with terminating long running instances is to research the expiration or timeout period for a given task. For example, on the Human Review task, there is typically a configuration for sending a reminder email to the review and an expiration period that will cause the canceling of the instance.

Singleton Process

The Singleton is a snippet or enhancement to another process model. It prevents multiple instances to be executing on the same data at the same time. The key is to query the database for an active instance of the same process model type and some of the process variables with the same values. If the results of the query are greater than one (there should always be one which is the instance executing the query) then that process instance is completed, allowing the other process instance to execute without interference. An enhancement is to send a self-termination email.


Query
Check if similar Instance is executing
Is Running?
Yes
No
Send Self-termination Email
(continue)

Wrapper Process

During the development and testing in the client's development environment, the communication between the Event Publisher and the BPM Subscriber worked without issues. When the process models were deployed to the production environment, the communications between the two agents would be operational for 2-3 hours and then fail with a lost connection error. The communication software would not recover from the failure and would require a restart of the BPM server to become operational again. The events published after the failure would be queued. After a server restart, the queued events would be sent and the communications would remain operational for 2-3 hours before failing again.

After exhausting the vendor's list of suggestions, an alternative route was designed, constructed, and implemented. This approach was intended to be used until the vendor's software upgrade was deployed. At that time, this interim solution will be decommissioned and the vendor's reference infrastructure will be deployed.

One of the driving factors in the design of the new communication solution was to leverage as much of the existing assets as possible. Since the client already had software deployed that had events being consumed by an Enterprise Service Bus (ESB), a similar architecture was selected. The architecture used the vendor's System Adapter solution to subscribe to events from the Event Publisher. Once an event is received, the System Adapter sends an HTTP request with the event data to a custom web service on the ESB. The web service converts the HTTP data to a JMS message and publishes it on a JMS queue. The BPM Subscriber receives the JMS queue and routes the inbound messages to the creation of the appropriate process model. A diagram of this approach is shown below:

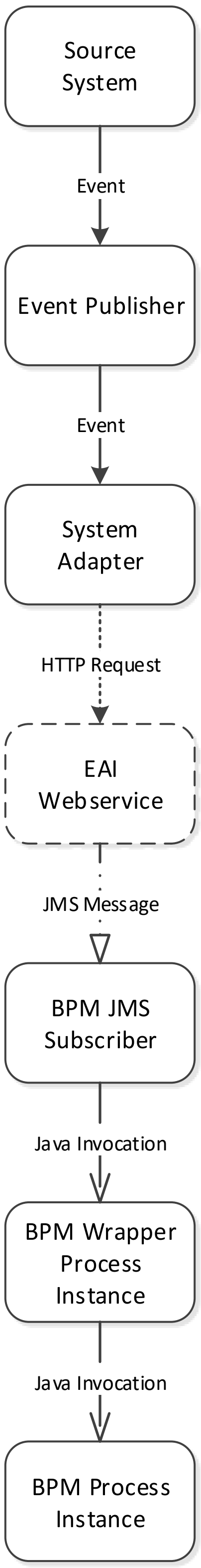


Figure 12 - Actual interface architecture

While this approach appears to be complex, fragile, and problematic, it proved to be extremely reliable. This biggest issue was that the format of the inbound JMS message was different than the message published by the Event Publisher. To address the issue of the different formats, the Wrapper Process pattern was used on all of the process models. This pattern has a “wrapper” process that performs the necessary inbound data transformation and mapping. After the message data is mapped to the internal representations, the wrapper process invokes the “core” process model that performs the relevant processing.

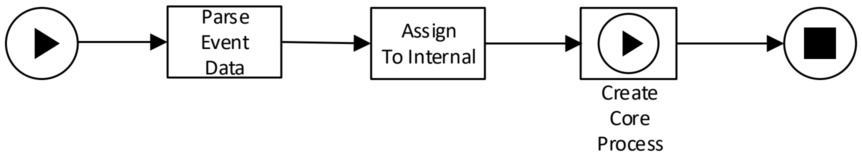


Figure 13 - Wrapper process

The challenge of using the wrapper process approach is that the inbound event data has to be explicitly mapped to the native process variables before the “core” process can be created. Typically, these process variables are defined manually in the process model editor and then the inbound XML elements are manually mapped to them.

While the manual technique is acceptable for small sets of process variables, the event data consumed by the process models can have 200 – 400+ variables. To reduce the amount of manual labor and associated errors, a Java application was created to populate the “Parse Event Data” and “Assign to Internal” processing tasks with the appropriate configuration. The use of this Java program greatly minimized the additional development required for the wrapper models.

The alternative interface architecture was used successfully for several years. Recently, the client decided to upgrade the vendor’s tool suite. After much testing, the new versions of the Event Publisher and BPM Subscriber proved to be much more reliable than the previous versions. The decision to use the reference architecture was made and the alternative architecture was decommissioned. The BPM Subscriber was reconfigured with each inbound event being mapped from the “wrapper” process model to the “core” process model. The entire conversion was completed in under an hour.