An Introduction to BPMN

By Derek Miers, CEO – BPM Focus.

Abstract: *This paper is drawn from Chapter 5 of the BPMN Modeling and Reference Guide, which provides detailed information on BPMN elements introduced. This document is designed to provide the reader with a gradual introduction to BPMN, taking an easily understood scenario and then slowly building upon it, bringing in further BPMN functionality within that described context.*

Designed for those coming to BPMN for the first time, it allows them to familiarize themselves with the core features of the Notation without being overwhelmed by the complexity of some of the more esoteric aspects. Most of the functionality described here is limited to the "core" set of BPMN elements – it is not the intention of this paper to provide a complete introduction to BPMN.

Building out a Process with BPMN

The central scenario revolves around a fictitious organization Mortgage Co. They take applications from potential customers, make an assessment whether or not to offer the mortgage, and then either reject the application or make the offer (see Figure 1).[1]

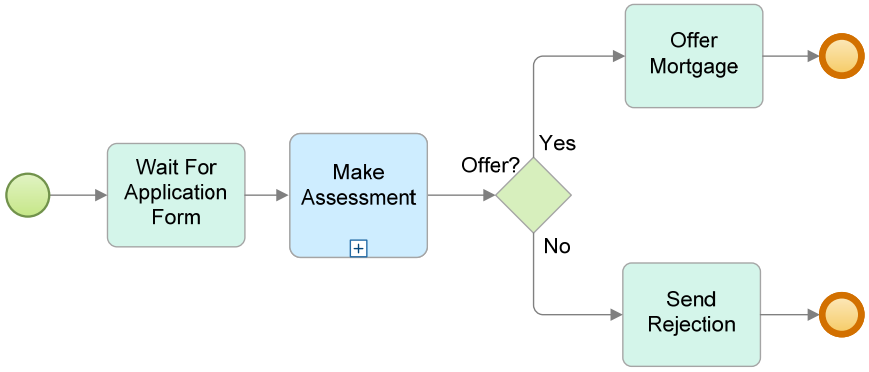

Figure 1—The underlying mortgage offer scenario

Clearly, this is a rather simplistic picture of how such a process might operate. But it will suffice in providing the backdrop for us to introduce the functionality of BPMN. Through the remainder of this part of the book, we will systematically build on that underlying scenario, embellishing the story and bringing in the appropriate BPMN modeling features to represent the desired behavior.[2]

The Process begins on the left with a Start Event (thin line circle), with two Activities (rounded rectangles) connected to the Start Event with Sequence Flow

[1] All paragraphs that build on the underlying scenario will share this font style (indented slightly and italics).

[2] We refer to the graphical elements of BPMN with Initial Capitals. Where an important BPMN concept is referenced (that is not a graphical element), we have used *italics* within the sentence.

(the arrows). The first Activity is a Task and the second represents a Sub-Process. Following a *decision*, represented by the diamond (called an Exclusive Gateway), the Process then branches to either "Offer Mortgage" or "Send Rejection" (both represented here as simple Tasks). Both branches lead to an End Event (thick circle).

Start Events represent the places that a Process can *start*, End Events represent different *results*, some of which might be desired and others not. As we shall see, BPMN supports a variety of Start Events and End Events. An Exclusive Gateway represents a binary decision—only one *outgoing* Sequence Flow can evaluate to *true*. Again, there are various types of Gateways in BPMN – they represent points in the process where you need to exercise control, either splitting or merging the Sequence Flow. For the purposes of this model, the three Tasks represent simple "atomic" steps, whereas the *collapsed* Sub-Process has a further level of detail.

Setting Timers

Now, let us assume that we want to represent the fact that our potential customer contacted Mortgage Co to ask for a mortgage application form. For the moment, we will not worry about precisely how they contacted the company, but let us assume it was a "message" of some sort. Further, we want to set a clock running to send them a reminder after seven days if Mortgage Co does not receive their application form back (see Figure 2).

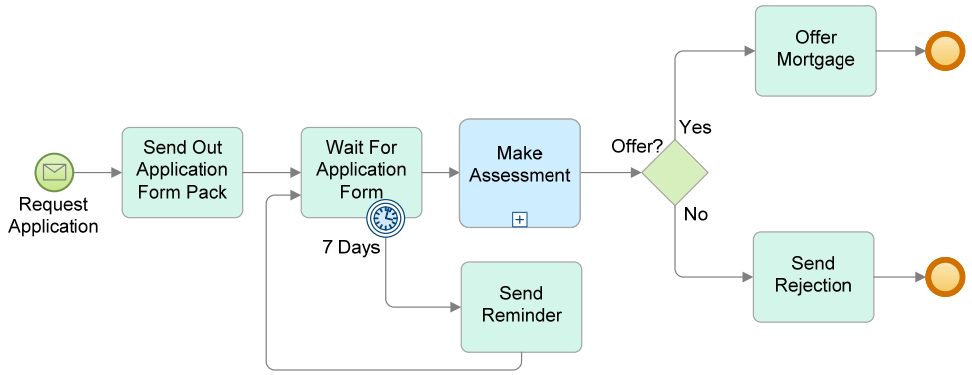

Figure 2—A Message Start Event and a Timer Intermediate Event are introduced

The Process now begins with a Message Start Event representing the message received by Mortgage Co, who then sends out the application form; a Timer Intermediate Event is placed on the waiting task to interrupt it and send a reminder before *looping* back again to wait for the application form again.

There are many types of Start Events in BPMN; here we have used a Message Start Event to indicate how this Process begins. Intermediate Events placed on the boundary of a Task, Activity or Sub-Process means that if the Event fires, then it will interrupt the Activity and send the Process down its *outgoing* Sequence Flow. The generic term "Activity" describes all Tasks and Sub-Processes. If the Activity completes before the Intermediate Event fires, then the Process moves on normally (following the *normal flow* of the Process). The loop is created explicitly with

Sequence Flow although, as we will discover later, there are alternatives (i.e., use a Loop Task).

There is another way to model this scenario using a Sub-Process for the send out application form and wait for the response Figure 3.

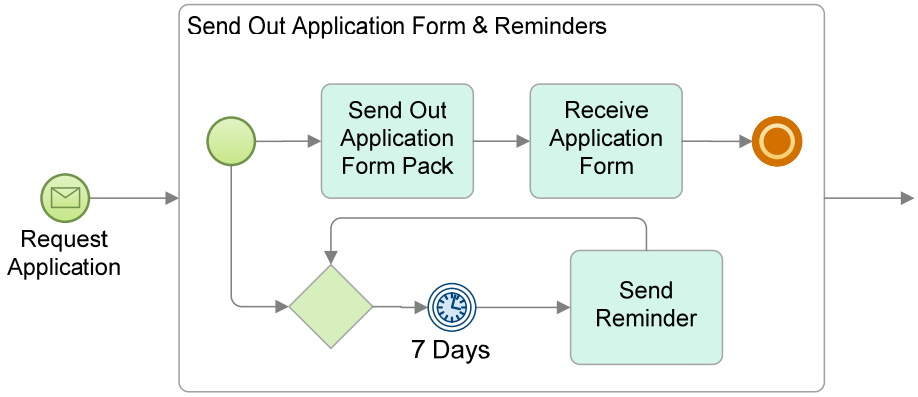

Figure 3—Using a Sub-Process to represent the application form and reminders

The Timer Intermediate Event shown "in line" with the Sequence Flow triggers immediately the Sub-Process begins (the Sub-Process is shown in its *expanded* form). It waits for seven days before that thread of activity moves to the "Send Reminder" Task before looping back to wait for another seven days. When an Intermediate Event is used in line (as in this case), then it can have only one *incoming* and one *outgoing* Sequence Flow. Therefore, merging the *incoming* Sequence Flow before the Timer Intermediate Event requires an Exclusive Gateway. When *merging* Sequence Flow, an Exclusive Gateway immediately passes through any *incoming* Sequence Flow so in this case it serves to clean up the Sequence Flow (but does not represent any sort of delay).

Of course, other *flow objects* (Activities or Gateways) can normally have multiple *incoming* and *outgoing* Sequence Flow. While the Sub-Process could have included a Parallel Gateway to create the split (see Figure 4), it is unnecessary as the Sequence Flow does not require control. Figure 3 and Figure 4 describe exactly the same behavior. A general rule is that Gateways are only required where Sequence Flow requires *control*.

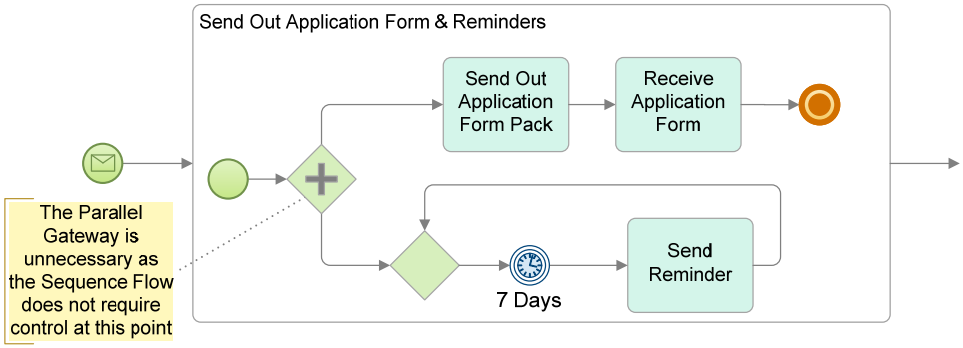

Figure 4—Using a Parallel Gateway is unnecessary

The Sub-Process finishes with a Terminate End Event. The Terminate End Event causes the immediate cessation of the Process on its current level (and below) even if there is still ongoing activity. Effectively, it kills off the reminder *loop*.

Exercise One

Try modeling this process; it will help ensure that the techniques discussed so far sink in:

> Every weekday morning, the database is backed up and then it is checked to see whether the "Account Defaulter" table has new records. If no new records are found, then the process should check the CRM system to see whether new returns have been filed. If new returns exist, then register all defaulting accounts and customers. If the defaulting client codes have not been previously advised, produce another table of defaulting accounts and send to account management. All of this must be completed by 2:30 pm, if it is not, then an alert should be sent to the supervisor. Once the new defaulting account report has been completed, check the CRM system to see whether new returns have been filed. If new returns have been filed, reconcile with the existing account defaulters table. This must be completed by 4:00 pm otherwise a supervisor should be sent a message.

Looping

So far, the *loop* is expressed using explicit Sequence Flow coming back to an earlier part of the Process. BPMN provides another mechanism to represent this sort of behavior—the Loop Task (see Figure 5). A Loop Task has a small semi-circular arrow that curls back upon itself.

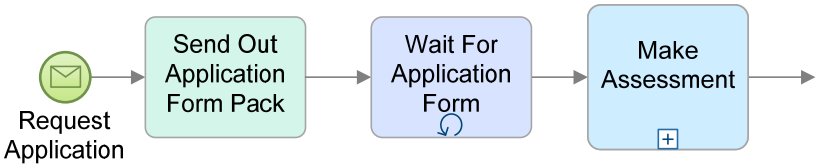

Figure 5—A simple Loop Task

It is possible to set BPMN attributes to support sophisticated looping behavior.[3] This is required to support the necessary complexity required by simulation and process execution environments.

> *Now clearly, it does not make much sense to endlessly loop back to wait for an application form that may never arrive. So after two such reminders, Mortgage Co has decided to cancel the application and archive the details.*

There is another way of setting the loop counter in Figure 6. Instead of using a graphically modeled "Set Loop Counter" Task, the "Send Reminder" Task could set an *assignment* at the level of the attributes. Although invisible, an annotation could then highlight its existence.

It is worth noting that the explicit Sequence Flow *loop* cannot cycle back to the Start Event. Indeed, Start Events cannot have *incoming* Sequence Flow. The *loop* can only go back as far the first Task.

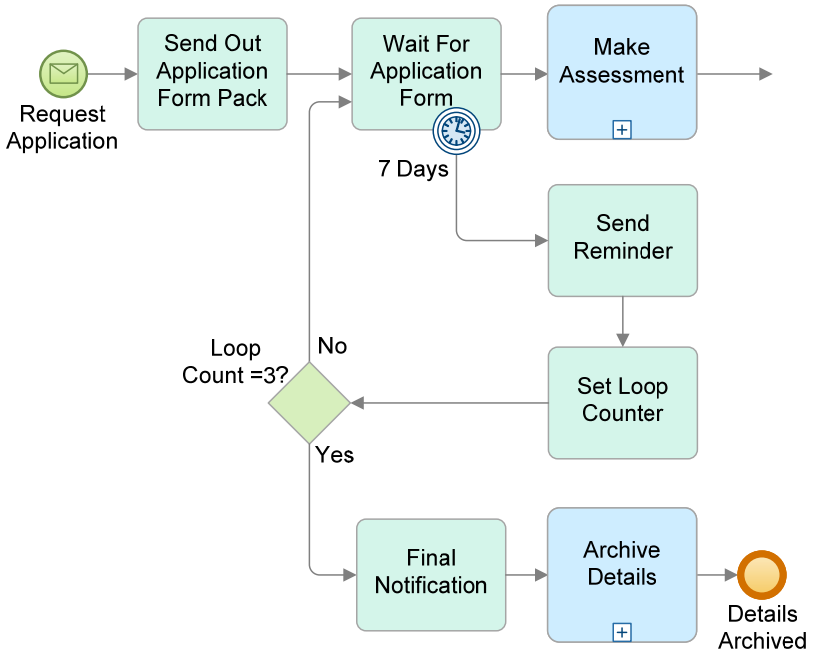

Figure 6—A loop counter is set and after two iterations, the details are archived and the Process ends

Decisions Based On Events

> *Of course, if the customer never sends back their application form, then the process will never get to the assessment phase. But what if the customer does let Mortgage Co know that they do not wish to proceed with the mortgage? The model in Figure 6 does not adequately represent this subtly different scenario.*

[3] Looping and other element attributes store information about the Process that is not shown graphically.

Now, after sending the application pack, Mortgage Co waits for one of three different things to happen. Either they receive the application (it moves on to the "Make Assessment" Task), or they are notified that the customer does not wish to proceed (in which case "Archive Details"), or after 7 days a reminder is sent (twice before sending a final advice and archiving the details).

While it is possible to model such a scenario using Activities, Intermediate Events, Sequence Flow and Exclusive Gateways, the model would become very messy and convoluted. As an additional exercise, it is worthwhile trying to model that problem using just these objects. But there is another, simpler way of modeling this situation. BPMN has an Event-Based Exclusive Gateway (see Figure 7) to handle these sorts of scenarios.

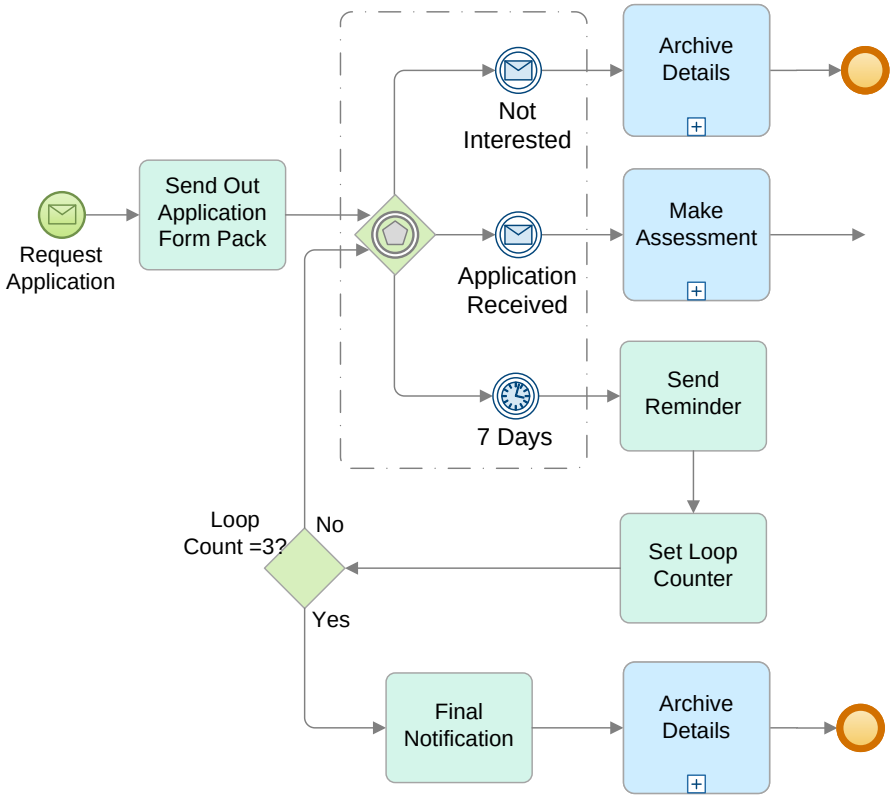

Figure 7—Using an Event-Based Exclusive Gateway

The Event-Based Exclusive Gateway (or informally, the Event Gateway) and its following Intermediate Events are regarded as a whole (the dot-dashed line around them is a BPMN Group used for emphasis only). To differentiate it from other Gateways, the Event Gateway reuses the Multiple Intermediate Event marker in the center of the diamond. Effectively, the Gateway waits for one of the subsequent Intermediate Events to occur. Either a *message* is received (Message Intermediate Event) indicating the customer is "Not Interested" or the "Application Received" Message Intermediate Event occurs (and the Process can progress normally), or the *timer* goes off on that Intermediate Event and the reminder *loop* is initiated. Another Sub-Process could represent the reminder *loop*.

Notice that the "Archive Details" *collapsed* Sub-Process appears twice on the diagram. This Sub-Process is designed as a *reusable* Sub-Process. It might appear in other Processes outside the scope of this example. Effectively, it represents a stand-alone Process referenced by this one. Of course, one could reorganize the diagram to use only one Activity on this model.

Meeting SLAs

> *Now let us assume that Mortgage Co receives the application form back and they have decided to institute a Service Level Agreement with their customers. They are now promising to respond with an offer or rejection within 14 days from the date of receipt of an application form. In support of this, the Process should alert the manager after 10 days if it has not completed, and then every day thereafter. Also, they need to archive the details if the decision was to reject the application (before the end of the Process).*

Thinking about the alert, the first temptation is probably to use a Sub-Process and then attach a Timer Intermediate Event to its border to create the alert (similar to Figure 2 on page 2). The problem with this approach is that it will *interrupt* the work of the Sub-Process, and a *loop* back to the beginning would cause the work to start again (not the desired behavior). The work should not stop just to raise an alert to the manager. Figure 8 shows one approach to solving this problem.

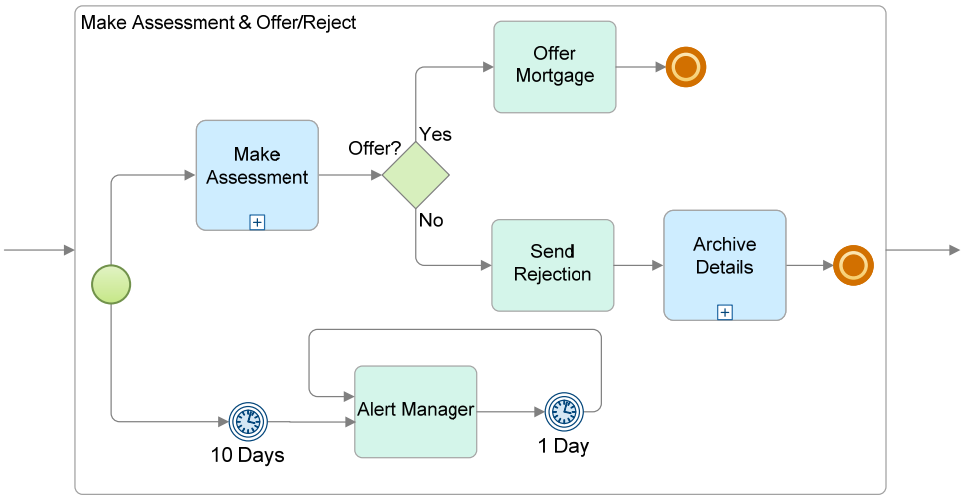

Figure 8—One approach to the non-interrupt alert problem

A separate Process *path* (or *thread*) with a Timer Intermediate Event linked to the Start Event of the Sub-Process is one approach to create a non-interrupting alert. The *timer* kicks in after 10 days if the work of the other thread has not finished—if that work is completed, then one or other of the Terminate End Events will kill off the *timer*. Effectively, a *race condition* occurs between these two strands of the process. Once the "Alert Manager" Task has occurred, it waits another day before looping back.

Representing Roles in Processes

The "Alert Manger" Task in Figure 8 above seems to imply that the manager receives a *message*. However, *messages* have a special importance in BPMN. Message Flow can only move between separate *participants* in a business-to-business situation. Each *participant* operates a separate Process represented by Pools. Message Flow coordinates the Processes of each *participant*.

Essentially, a Process exists within a single Pool. Labeled boxes display the Pool; they also have square corners as opposed to Tasks and Sub-Processes, which have rounded corners. BPMN uses Pools when representing the interaction between an organization and *participants* outside of its control. Within a company, a single Pool covers its own internal operations—it is only when it interacts with external *participants* that additional Pools are required.[4]

> *For example, in our Mortgage Co, the Credit Agency (and the Customer) would have a separate Pool (assuming one was trying to represent the interactions between the parties).*

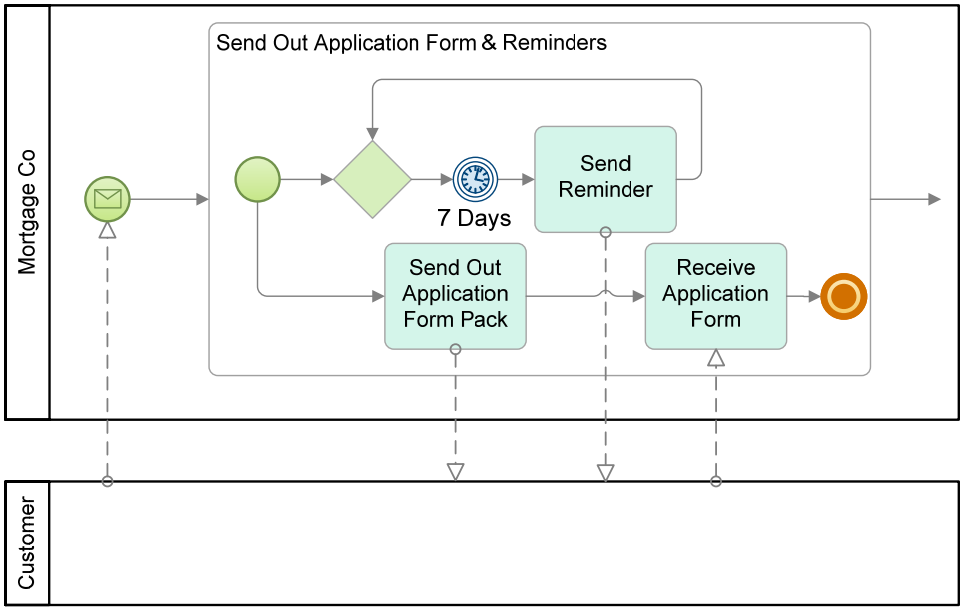

Figure 9—Representing the customer in a separate Pool

Message Flow cannot communicate between Tasks inside a single Pool—that is what Sequence Flow and *data flow* (as we shall see below) does. Sequence Flow is used to moves the Process from one Activity to another. In this example, (see Figure 9) the "Customer" Pool interacts with a fragment of the "Mortgage Co" Process using *messages*.

[4] Separate Pools might be used where an organization had several independent business units that were collaborating. In such a situation, each business unit would not necessarily know the internal operations of the others, yet would need to indentify the interfaces between them.

Mortgage Co does do not know the Customer's internal Process. Hence, the representation for the Customer is a "Black Box Pool." Within the Mortgage Co Pool, the Message Start Event receives an *incoming message* from the Customer, which triggers the Sub-Process. A race condition then starts between the two threads of the Sub-Process.

Two of the Tasks in the Sub-Process are Send types of Tasks, while the third is a Receive Task. In BPMN 1.1, there is no standard graphical way to differentiate Send and Receive Tasks. Their type is implied by the direction of the Message Flow and stored attributes.

Exercise 2

Try this exercise.

> The Customer Service Representative sends a Mortgage offer to the customer and waits for a reply. If the customer calls or writes back declining the mortgage, the case details are updated and the work is then archived prior to cancellation. If the customer sends back the completed offer documents and attaches all prerequisite documents then the case is moved to administration for completion. If all pre-requisite documents are not provided a message is generated to the customer requesting outstanding documents. If no answer is received after 2 weeks, the case details are updated prior to archive and cancellation. [5]

Modeling Data and Documents

> *Mortgage Co handles many documents. They come from lots of different sources—the "Surveyors Report," the "Credit Report," the "Title Search" and the "Application Form." In the context of the Processes of the firm, the documents move through various states as the employees carry out their work. The documents are handled, scanned, sorted, annotated, versioned, archived, etc. Images are linked to customer records, with employees transposing some of their content into data fields for the company's information systems.*

Clearly, there is a need to understand how these data and documents are manipulated within a given process. For example, in Figure 10, the "Rejection Letter" and "Assessment" Documents are represented by Data Objects. Data Objects are the Artifacts of the Process. They do not move along with the Process flow, but act as inputs and outputs of Tasks.

Data Objects exist outside of the Sequence Flow of the Process, but they are available to all *flow object*s in a given Process *instance*. *Data flow* passes information into or out of an Activity. Of course, the implementation mechanism used in any given system is going be specific to the platform used to support the process.

[5] Example answers to these Exercises will be made available online at http://www.bpmnreferenceguide.com/

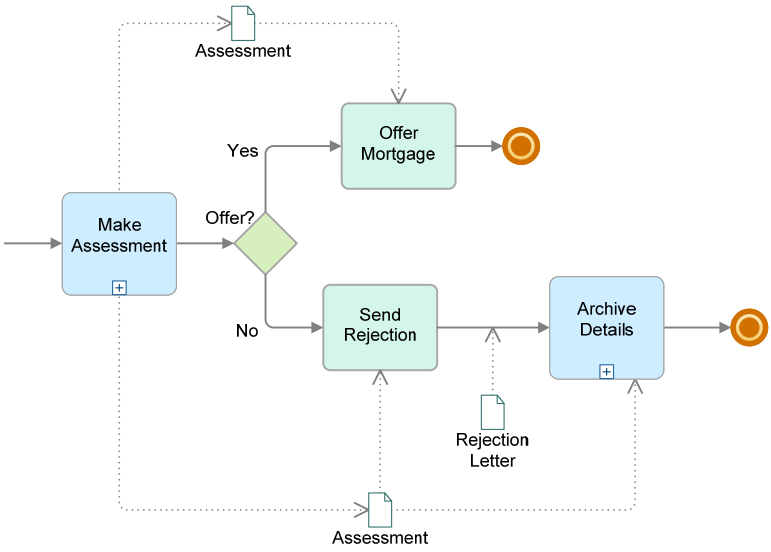

Figure 10—Representing documents in the Process

Figure 10 above demonstrates two different ways of showing *data flow*. The "Assessment" Data Object is *output* from the "Make Assessment" Sub-Process using an Association connector. The "Assessment" Data Object is also *input* to the "Archive Details" Sub-Process. The arrowheads on the Association indicate the direction of the *data flow*.

The "Rejection Letter" Data Object is attached to the Sequence Flow between "Send Rejection" and "Archive Details." This is really a sort of shorthand used when the *data flow* is between two Activities follow each other.

Another subtle implication of the *incoming data flow* is that it tells the reader that these Data Objects must be available in order for the Tasks to start. For example, when the Sequence Flow arrives at the "Send Rejection" Task, it sets the *state* of the Activity to *ready*. It is ready to begin, but it cannot start until all of its inputs (the "Assessment" Data Object) are available. [6]

Coordinating Parallel Threads of Activity

> *Coming back to the processes of Mortgage Co, we have so far avoided a core component of their business—making assessments about mortgages and their viability.*
>
> *The "Make Assessment" Sub-Process is where the real work of the Process happens. Contained within that Activity are a number of Sub-Processes that need to occur in parallel; the credit check, property title search and property survey.*
>
> *The problem is that Mortgage Co also needs to keep its costs down and at the same time respond as quickly as possible to customer requests. So they have*

[6] Actually, it is technically possible to set the underlying attributes of the Activity to allow it to start and have updated Data Objects arrive while the Activity is in progress.

teams and service providers that need to do things in parallel and yet still have the ability to communicate with the other teams should one identify a problem that would invalidate the mortgage application. In the past, they have tried using email for this, but have found it inefficient and prone to cases slipping through the cracks.

While the detail of each of these Sub-Processes is not so important at this point, the key issue to observe is that a bad result in either of these areas will invalidate the mortgage (or at least imply that work in the other areas should halt).

Of course, a good result in any one of these areas means that work can start immediately on preparing the Mortgage offer documents, but that work needs to halt should a negative result come back from one of the other areas. In this way Mortgage Co can enable, as much efficiency as possible, and at the same time reduce the cycle time of the process.

There is another way of handling communication in BPMN. Instead of a directed *message* (that has to go to a particular external *participant*), or Sequence Flow (that cannot cross a Pool or Sub-Process boundary); *Signals* offer a general inter-process communication capability. They can operate within a Process or between Pools and they can cross Process boundaries—think of them like a signal flare or fire siren. They are not directed to a specific recipient, instead all who are interested can look/listen and detect the *signal* and then act appropriately.

The Signal Intermediate Events have two distinct modes of operation. They either send *signals* or listen for them. In Figure 11 below, the Signal Intermediate Events are all set to listen (they are all in the bottom Sub-Process "Prepare Offer Letter"). That is, they *catch* the Signal broadcast by the Signal End Events. All the Signal End Events send *signals*—that is they *throw* the *signal*.

Where Intermediate Events *catch* the *trigger* shown in the center is white (as in a Start Event); where they *throw*, the center is solid (like an End Event). Of course, a Signal Intermediate Event can also *throw* (in which case it would have two concentric thin lines with a solid triangle in the center). Indeed, all *trigger* Events (Start, Intermediate, and End), either *throw* or *catch*. This is inherent to what Events really are.

All Start Events *catch*—that is, that they can only receive *incoming triggers*. It does not make sense for a Start Event to "send," it responds to an Event that happens. Somehow, it is detected and that is what triggers the Event. The markers for all Start Events are white-filled.

All End Events *throw*—they can only fire *triggers* for other Events to *catch*. End Events cannot detect things that happen (what would they do with them, they are at the end?). Instead, they can create Events to which others respond. The markers for End Events are black-filled.

Depending on the sort of Intermediate Event and its contextual usage, the Event either *throws* or *catches* (or both) the *trigger*. Some Intermediate Events always come in pairs; others operate independently. The *catch* Intermediate Event markers are white-filled and the *throw* Intermediate Event markers are black-filled.

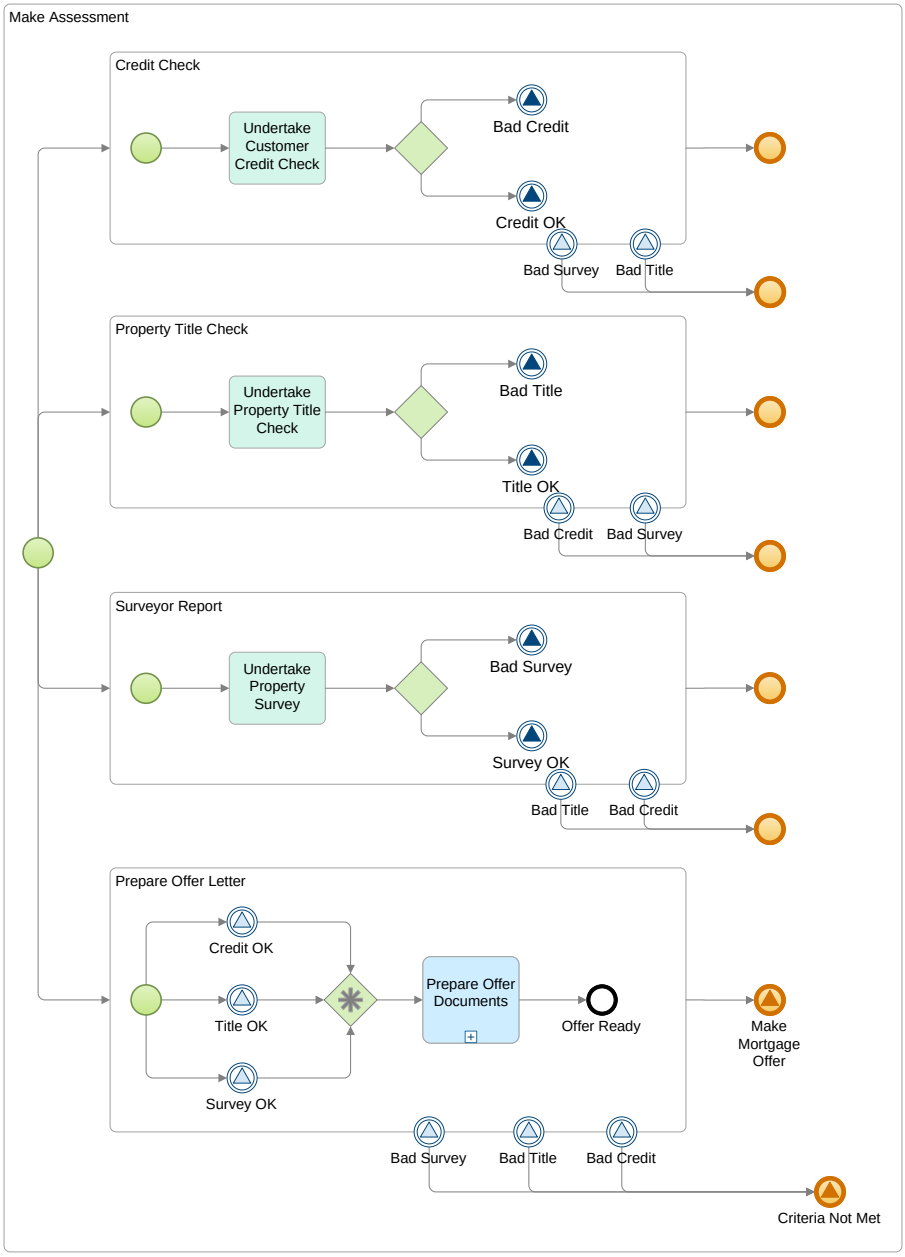

Figure 11—Using Signal Intermediate Events to communicate

In Figure 11 above, *catch* Signal Intermediate Events in the bottom Sub-Process are set to capture the *signals* broadcast by the End Events of the first three Sub-Processes (the "Credit OK," "Title OK" or "Survey OK" results). If a "Bad Survey," "Bad Credit" or "Bad Title" End Event occurs, it will trigger one of the Intermediate Events attached to each of the boundaries of the other Sub-Processes, thereby interrupting all of the work going on there.

The "Prepare Offer Letter" Sub-Process starts along with the other three Sub-Processes, but then waits for any one of these *signals* to occur. As soon as one of

them happens (detected by the Signal Intermediate Event), it moves the Process on to the Complex Gateway (diamond with a bold asterisk at its center). This Complex Gateway is used to merge the Sequence Flow from these three Intermediate Events.

A Complex Gateway enables the modeler to capture behavior that does not exist in the other Gateways. Think of it as a warning that here the system is likely to drop into complex rules or code. In this case, the "Prepare Offer Documents" Sub-Process can start upon detection of any of the three *signals*. But as other *signals* are detected, a new *instance* of the Sub-process is not required. A normal Exclusive Gateway would result in duplicate process *instances* as each new Event happened.

If a "Bad Survey," "Bad Title" or "Bad Credit" Signal Intermediate Event fires, then the "Prepare Offer Letter" Sub-Process is also interrupted leading it to fire a "Criteria Not Met" Signal End Event. Assuming none of those things happen, the entire "Make Assessment" Sub-Process will complete normally with a "Make Mortgage Offer" Signal End Event.

The "Make Assessment" Sub-Process (*expanded* in Figure 11 above, but *collapsed* again in Figure 12), will send one of the two possible *signals* back to the *parent* Process: "Make Mortgage Offer" or "Criteria Not Met."

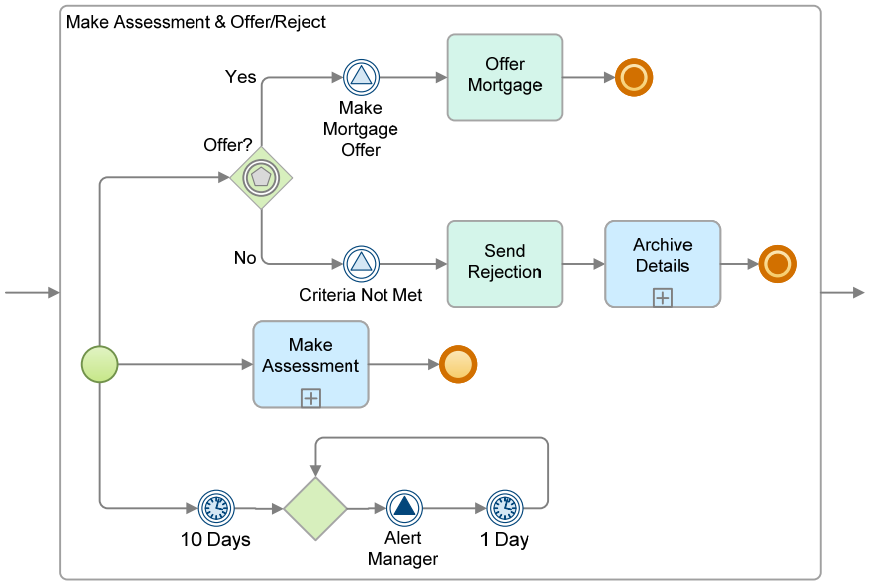

Figure 12—A revised "Make Assessment & Offer/Reject" Sub-Process

The decision (using an Event-Base Gateway) to offer the mortgage now operates in parallel to the "Make Assessment" Sub-Process. It is waiting for either the "Make Mortgage Offer" or the "Criteria Not Met" *signal* (*thrown* by the Sub-Process). If the Event-Based Gateway was inline, after the "Make Assessment" Sub-Process, then the *signals* in the Sub-Process would fire before the *parent* was ready for them (in which case they are ignored). Notice also that the "Make Assessment" Sub-Process goes to a None End Event—those threads will finish without affecting either of the

two branches from the "Offer?" Event-Based Gateway.[7] The same behavior exists in the first 3 Sub-Processes in "Make Assessment" as shown in Figure 11.

The key point to understand is that *signals* can communicate between different levels of the Process (between Sub-Processes and the *parent* Process). Equally, *signals* could communicate from a *parent* Process to its Sub-Processes, or to other Processes. Signals provide a general form of inter-process coordination within BPMN. Without the use of a *signal*, coordination would rely on Process data (and an Exclusive Gateway). In the end, it is a matter of personal choice—i.e. a modeling decision.

Exercise 3

Another brainteaser:

> In November of each year, the Coordination Unit at the Town Planning Authority drafts a schedule of meetings for the next calendar year and adds draft dates to all calendars. The Support Officer then checks the dates and suggests modifications. The Coordination Unit then rechecks all dates and looks for potential conflicts. The final schedule of meeting dates is sent to all the independent Committee Members by email, who then check their diaries and advise the Coordination Unit of any conflicts. Once the dates are finalized (by the Coordination Unit), the Support Officer updates all group calendars and creates meeting folders for each meeting and ensures all appropriate documents are uploaded to system. Committee Members are advised a week before each meeting to read all related documents. The Committee Members hold their meeting, and the Support Office then produces minutes including any Action Points for each Committee Member. Within 5 working days, the Coordination Unit must conduct a QA check on the minutes, which are then sent to all Committee Members. The Support Officer then updates all departmental records.

Another Approach to Escalation

> *Returning to the non-interrupting alert needed (for the manager as discussed for the model in Figure 8), it is unlikely that the Manager works for an external business entity, so the Task is not a Send Task.*

Figure 12 above also uses a Signal Intermediate Event to initiate (*throw*) the interaction with the Manager role. In Figure 13, a corresponding Signal Intermediate Event exists in the Manager Lane to listen for such an escalation—i.e., it is waiting to *catch*. In this case, the Signal Intermediate Event supports communication at the same level within a single Pool but across two Lanes.

Figure 13 provides yet another alternative approach to the non-interrupt alert problem. It also provides an overview of the Process developed so far.

[7] Technically, the *signals* fire at the end of the sub-process which is also the same time that the Event Gateway fires, so the *signal* would probably be detected. This model is drawn in parallel to ensure the required behavior occurs.

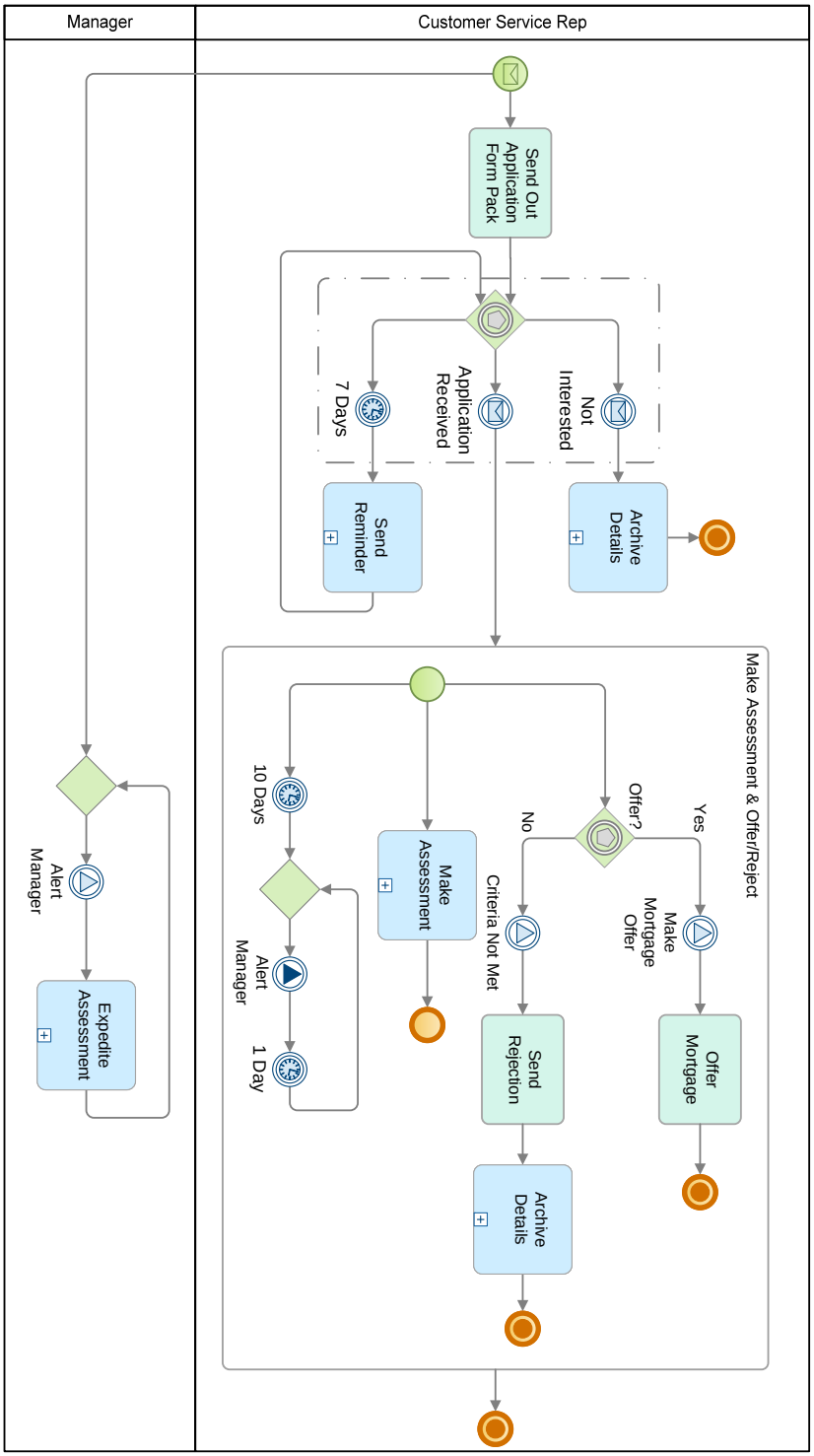

Figure 13—The complete Process employing two Lanes to represent the customer service representative and the manager, making use of Signal Events and to coordinate the offer decision with the Sub-Processes of Figure 11

More Than One Right Answer

Just like decisions taken by the modeler (what detail to include and how to present it), decisions taken within a Process do not always have just one correct answer.

> *Consider Mortgage Co as it compiles the offer documents for its customers. Depending on the mortgage applied for, different documents are required. So a generic mortgage application Process needs mechanisms to differentiate which sub-set of documents to include—let us assume a main proposal plus any number of supplements.*

The precise detail of each rule is not our concern here, but providing a Process backdrop for those decisions is not easy if the modeler is restricted to Exclusive Gateways. Process models would become inordinately complex and difficult to follow.

BPMN provides a couple of mechanisms to handle this sort of challenge. The Inclusive Gateway allows for decisions, where all *outgoing* Sequence Flow *conditions* that evaluate to *true* are activated. This is in stark contrast to the Exclusive Gateway where only the first *condition* that evaluates to *true* activates (all others are ignored).

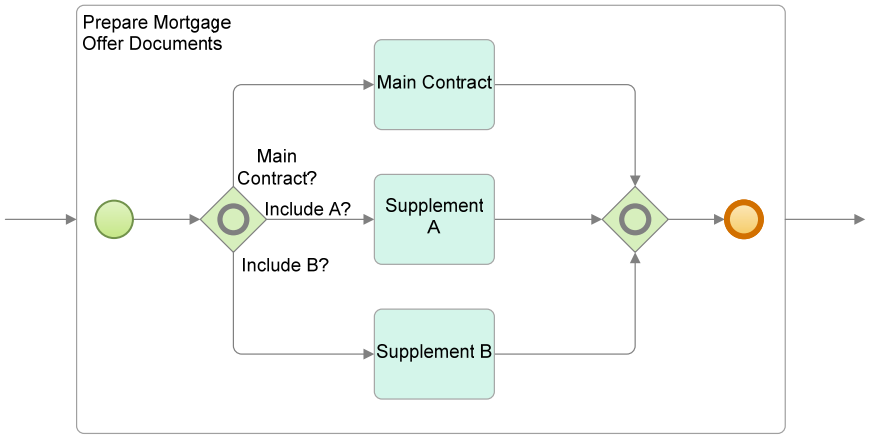

Figure 14—Dealing with decisions that have more than one right answer

The *splitting* Inclusive Gateway has a circle at its center to indicate that each *outgoing* Sequence Flow is evaluated. If it returns a *true* value, then the Sequence Flow is activated.

The other approach is to use Conditional Sequence Flow (see Figure 15). Each Conditional Sequence Flow has a mini-diamond at the point it leaves and Activity. Each is evaluated in turn and if it returns a *true* value then the Sequence Flow is activated.

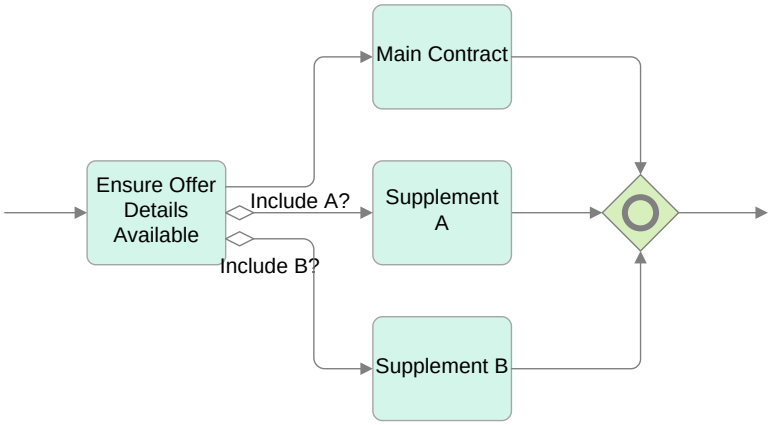

Figure 15—Using Conditional Sequence Flow

Notice that Figure 14 and Figure 15 both use a *merging* Inclusive Gateway to ensure that the correct number of Sequence Flow are joined together again. While there is no requirement that the *outgoing* flows of an Inclusive Gateway merge (they could each follow independent paths and never come back together), if the intention is to rejoin these threads together, then a *merging* Inclusive Gateway is needed. On the other hand, a Parallel Gateway would expect all *incoming* Sequence Flow to fire (and if they did not, the Process would halt at that point, waiting for a Sequence Flow that never arrives). If an Exclusive Gateway were used, it would not merge the paths together at all; instead each thread would pass straight through.

Exercise 4

After the Expense Report is received, a new account must be created if the employee does not already have one. The report is then reviewed for automatic approval. Amounts under $200 are automatically approved, whereas amounts equal to or over $200 require approval of the supervisor.

In case of rejection, the employee must receive a rejection notice by email. The reimbursement goes to the employee's direct deposit bank account. If the request is not completed in 7 days, then the employee must receive an "approval in progress" email

If the request is not finished within 30 days, then the process is stopped and the employee receives an email cancellation notice and must re-submit the expense report.

Exercise 5

After the Process starts, a Task is performed to locate and distribute any relevant existing designs, both electrical and physical. Next, the design of the electrical and physical systems starts in parallel. Any existing or previous Electrical and Physical Designs are inputs to both Activities. Development of either design is interrupted by a successful update of the other design. If interrupted, then all current work is stopped and that design must restart.

In each department (Electrical Design and Physical Design), any existing designs are reviewed, resulting in an Update Plan for their respective designs (i.e. one in Electrical and another in Physical). Using the Update Plan and the existing Draft of the Electrical/Physical Design, a revised design is created. Once completed the revised design is tested. If the design fails the test, then it is sent back to the first Activity (in the department) to review and create a new Update Plan. If the design passes the test, then it tells the other department that they need to restart their work.

When both of the designs have been revised, they are combined and tested. If the combined design fails the test, then they are both sent back to the beginning to initiate another design cycle. If the designs pass the test, then they are deemed complete and are then sent to the manufacturing Process [a separate Process].