

Keeping BPM Simple for Business Users: Power Users Beware

Michael Havey

*in heavenly realms of hellas dwelt
two very different sons of zeus
– e. e. cummings*

This article is a thought experiment: Invent a simple business process modeling notation suitable for a workshop in which business analysts (BA's) help business users chart their as-is and to-be processes. The experiment is no idle diversion. True, process languages abound today, and the last thing the world needs is another one. (Besides, Yet Another Workflow Language, or YAWL, has already been invented.) The motivation for the experiment is the goal of continuity in the software lifecycle for BPM projects; in particular, that the process deployed to production is the same model (by then fully developed and tested) as the one created during the initial requirements phase. The movement of this one model through the phases of the cycle is depicted in Figure 1.

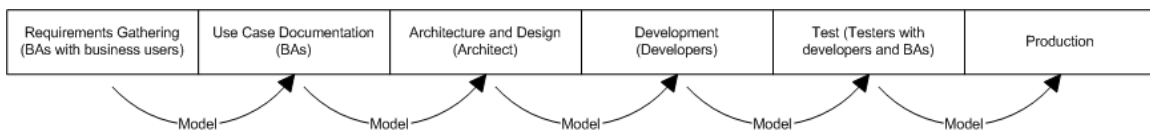


Figure 1. One Model for the BPM Lifecycle

Getting to production requires getting things right in the prior phases. The very first phase (requirements gathering) will fail unless the model is based on an approach that *business users* can understand. The goal of the experiment is to invent a language that will resonate with business users but can also be used through the remainder of the lifecycle. The second part is the hardest. Will the model created during requirements be in good shape for use case definition, or will the BA need to redraw it in a different form? Will the architect, or design team, for that matter, be able to work with it, or will he or they need to adopt a more technical modeling approach? The goal is to avoid such a break in continuity.

What Works in the Workshop

In a workshop setting, modeling notation is merely a means to an end. The goal of the session is to talk as a group through processes: The users describe how their business works now and how it *should* work, the BA's try to solicit and isolate firm requirements. The talk never stops. During the session, a BA sketches on a whiteboard a crude flowchart that captures the process flow emerging in the discussion. The diagram is, in a sense, yet another voice in the exchange, which the participants understand as plainly as each other's words. The sketch is continually reworked; by the end of the meeting, it represents crudely the users' notion of their process, and it gives the BA material to work with when drafting a more carefully constructed business use case document.

Business users know their business but are not strong on notation. When users are presented with a diagram drawn in a formal notation – such as Business Process Modeling Notation (BPMN) or UML 2.0 Activity Diagrams – they get the gist of it but need to be guided gently through the fine points. Creating such a diagram with them, or even modifying one already started, requires too much handholding. Too much time is wasted explaining design elements (e.g., you need to use an interruptible region here, you need an intermediate cancel event there);

an unwelcome topic on the agenda. The goal is to talk through processes; the more plainspoken the diagram, the better.

The notation proposed by this thought experiment must therefore be dumbed down as far as possible, consisting of just boxes and arrows. A box represents an activity performed by some person or system; it is labeled with the activity's name, actor (i.e., who or what system performs it), trigger (e.g., time-driven, event-driven), input and output data, and other aspects the user wishes to document. An arrow moves control from one activity to another; it can be left unlabeled, but can also specify a Boolean guard condition (more on this shortly), and can be annotated with any other pertinent information. Nothing else is required. Swim lanes are excluded because they take up too much space on the board (too much effort is expended managing whiteboard real estate), and writing the performing actor in the activity box achieves the same effect anyway. Diamond-shaped decision points are excluded because using an arrow with an annotated condition is equivalent and simpler to notate.

Figure 2 shows a fragment, drawn in the box-and-arrow notation (call it the Simple notation), of a bank loan approval process. The fragment has three activities run sequentially. The first activity, named "Get Approval," is, as the first bullet describes, a manual task to be performed by a credit officer within three business days; the logic for a timeout is not specified. Once the approval is granted, the account database is updated ("Update Account DB") using a real-time interface, which, as the notes tell us, replaces the existing batch feed, must deal with the anomaly of grandfathered fields, and should be retried twice on failure. The final activity ("Send Welcome Package") sends a welcome letter to the applicant, preferably using an automated letter print function such as the one used by the bank's VISA group.

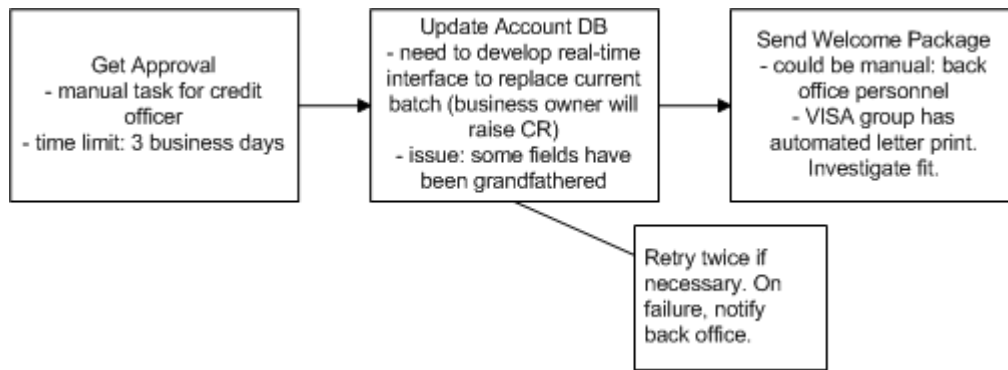


Figure 2. Loan Approval Process Fragment in Simple Notation

The style is intuitive and informal. Only the happy path is shown graphically; detailed requirements, as well as notes on implementation and exception handling, are documented in notes. Granted, the model is too imprecise to generate executable code, but it conveys the flow to business users better than a more precise model (e.g., a model that added extra boxes and arrows to deal with the retries on "Update Account DB").

The Limits of Simple

With only two graphical elements (boxes and arrows) in its set, this language necessarily has remarkably simple execution semantics, supporting only basic sequential flow (as in the example in the previous section) and AND/OR splits and joins. Figure 3 shows examples of AND and OR in the Simple language and, for comparison, in BPMN.

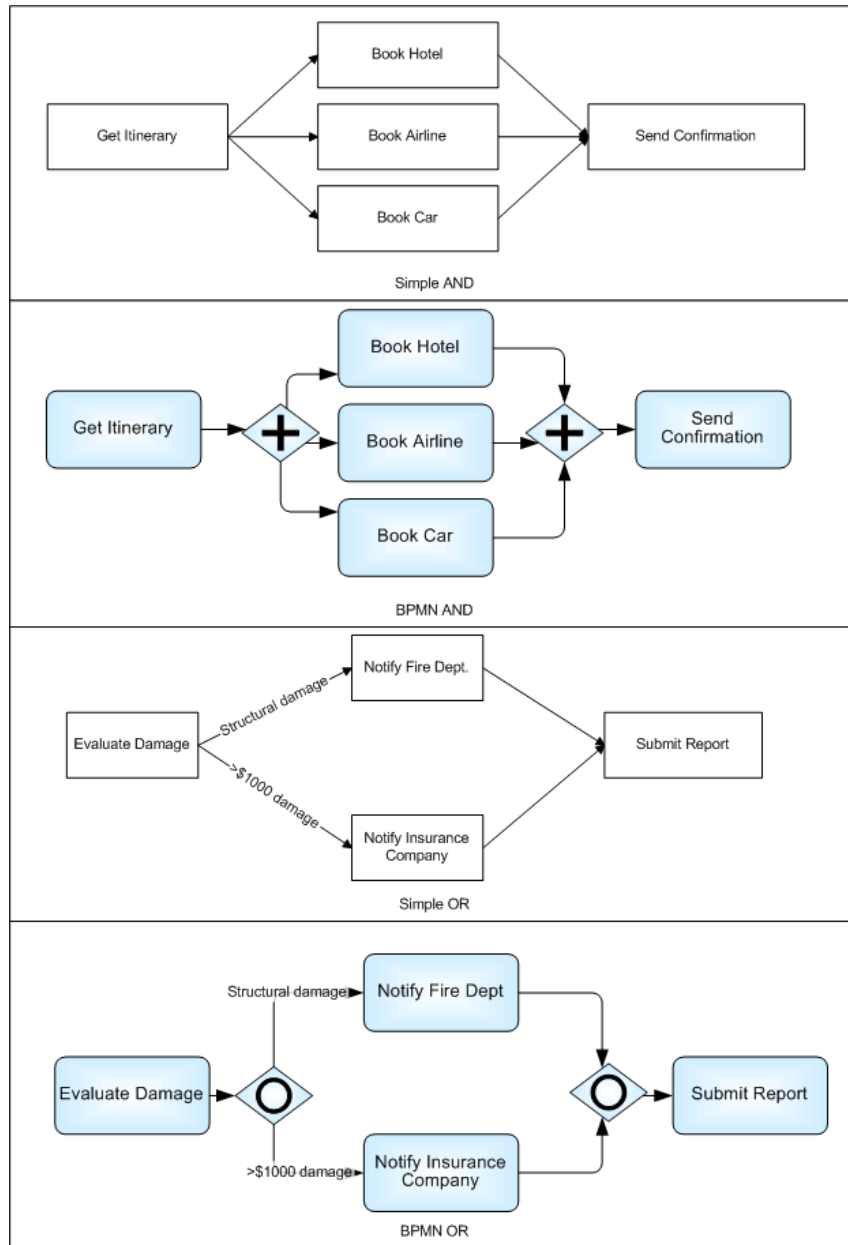


Figure 3. AND and OR split/join in Simple and BPMN notations

In the travel reservation process shown in the first set of diagrams, the activity “Get Itinerary” runs first, and then branches into three *parallel* activities – “Book Hotel,” “Book Airline,” and “Book Car” (demonstrating an AND split). *After all of these have been completed*, “Send Confirmation” is run (demonstrating an AND join). The split is shown by having one activity link to multiple activities; the join is shown by having multiple activities link to one activity. The BPMN representation is similar, but uses *parallel gateways* (diamonds with a plus sign) to manage the split and join. (See Stephen White’s article, referenced below, for an account of BPMN’s support for this and other patterns.)

The second set of diagrams demonstrates the OR split/join. The process shown determines the actions taken by the maintenance department of a commercial building after a small fire. If the fire caused structural damage, the department must contact the fire department; if the damage

exceeds \$1000, the department must contact the insurance company; in either case, a report must be submitted to the building manager. The OR split moves control from “Evaluate Damage” to “Notify Fire Dept.” and “Notify Insurance Company.” The conditions determining which way to branch are labeled on the arrows. In this example, the OR condition is *inclusive*; it is possible for the damage to be both structural and greater than \$1000 in value, in which case the respective branches are run in parallel. (In an *exclusive* OR, the conditions are mutually exclusive and only one branch is followed.) The OR join is from “Notify Fire Dept.” and “Notify Insurance Company” to “Submit Report.” If multiple branches were selected by the OR split, “Submit Report” waits for all to complete. (To help keep track, the *dead-path elimination* technique, used in languages such as BPEL, can be applied, transparent to business users) In the BPMN representation, inclusive gateways (diamonds with an O) are required to manage the split and join.

More complicated patterns cannot be expressed easily in our language. Figure 4 shows Simple and BPMN representations of two of the patterns documented by Wil van der Aalst, *et al*, at www.workflowpatterns.com.

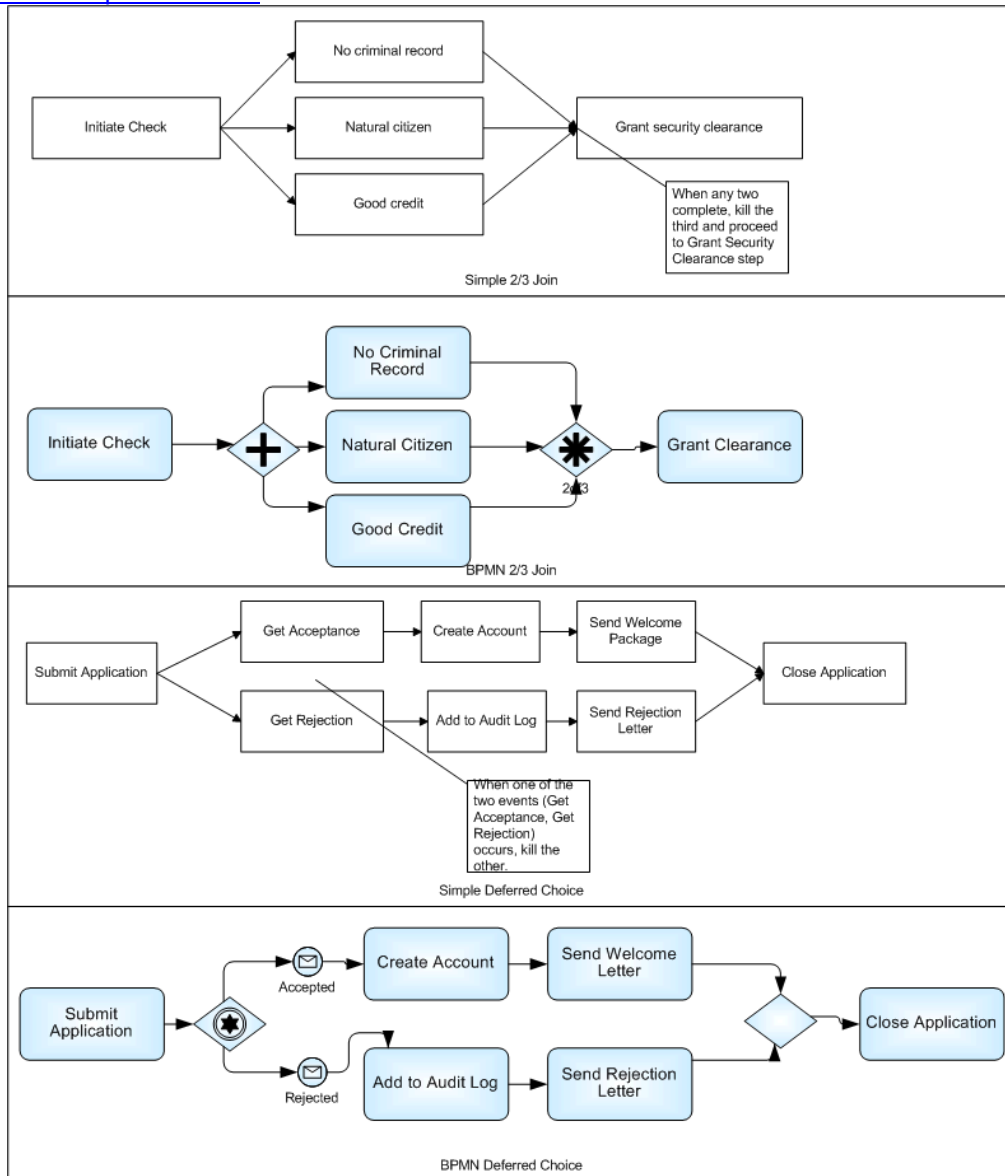


Figure 4. 2-of-3 Join and Deferred Choice in Simple and BPMN notations

The first two diagrams in Figure 4 model a 2-of-3 join. To process a security clearance application, federal officers check concurrently whether the applicant has a criminal record, is a natural citizen, and has good credit. As soon as two of these checks pass, the applicant is granted clearance, and the third check is aborted. This pattern, a special case of the “N/M join,” is hard, if not impossible, to model in our simple notation. The activities “No Criminal Record,” “Natural Citizen,” and “Good Credit” are run in parallel (having been launched by the AND split from “Initiate Check”). In the relaxed style of the Simple notation, the complex two-of-three join to “Grant Security Clearance” is shown as an AND join annotated with descriptive English text describing the 2-of-3 condition. As before, don’t expect to generate any code from this casual model. The BPMN diagram, on other hand, uses a special *complex gateway* element (diamond with an asterisk) to model the 2-of-3 join – more precise, to be sure, but business users are probably more comfortable reading the clumsy annotated text of the Simple diagram.

The second set of diagrams depict a *deferred choice* (also known as an “event choice” or a “pick”), which waits for exactly one of several events to occur, and then executes the logic for that event. The loan approval process shown in the figure submits the application to a credit officer (“Submit Application”) and waits for either acceptance (“Get Approval”) or rejection (“Get Rejection”). The path for acceptance runs the activities “Create Account” and “Send Welcome Letter” in sequence; the path for rejection executes “Add to Audit Log” and “Send Rejection Letter.” When the chosen path completes, the activity “Close Application” is run. Although many contemporary process languages support this pattern directly (the BPMN representation uses an *exclusive event gateway*, shown as a diamond with a circumscribed star), the Simple language keeps it simple: The two paths are run in parallel using an AND split, but, as the annotation in the diagram explains, as soon as one of the events occurs, the other is immediately disabled.

Figure 5 demonstrates another recurring design construct – the loop.

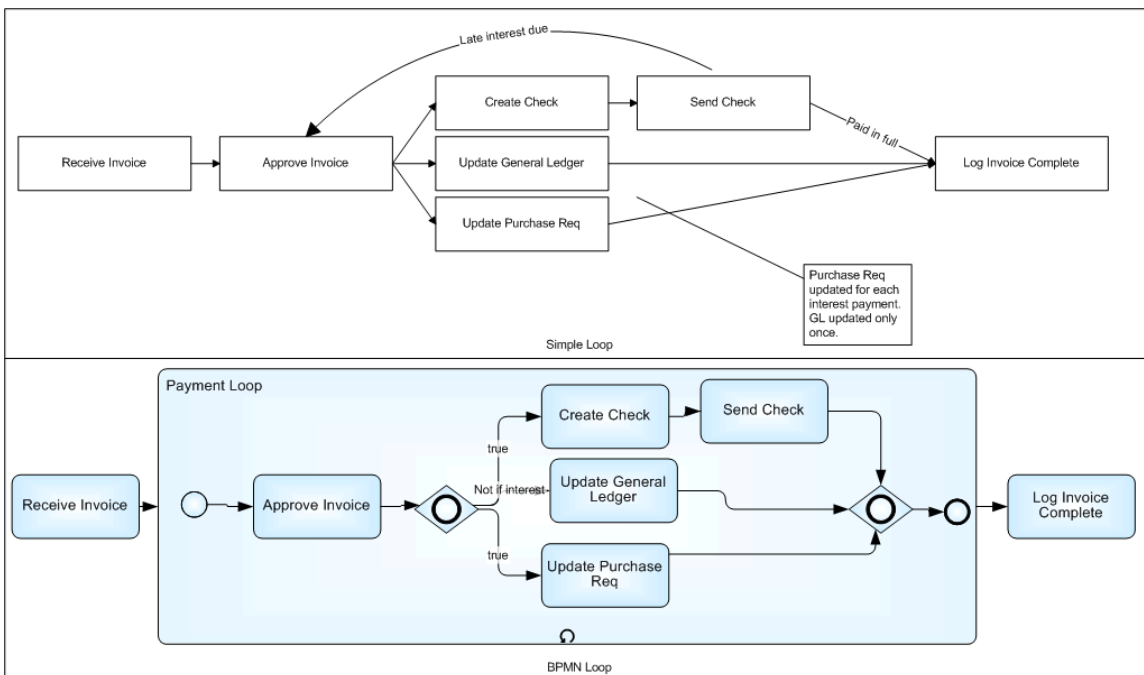


Figure 5. Loop in Simple and BPMN notations

The process shown describes how a buyer company pays an invoice from a supplier. After the buyer receives the invoice (“Receive Invoice”), a financial manager examines it and approves it

“Approve Invoice”), contacting the supplier to resolve any discrepancies, whereupon several actions are performed in parallel: The original purchase requisition and the general ledger are updated with the invoice information (“Update Purchase Req” and “Update General Ledger”), and a check to pay the invoice is created and sent to the supplier (“Create Check” and “Send Check”). If, by the time the check is sent, the buyer determines it has incurred late interest charges (because of delays in the approval or check processing), the process loops back to the “Approve Invoice” activity, and that part of the process repeats, except the “Update General Ledger” step is skipped, as the note indicates. The loop can repeat multiple times (if additional interest payments are required), and stops only when the invoice has been paid in full, at which point the “Log Invoice Complete” is executed.

The loop is implemented as a GOTO-style jump, or an *arbitrary cycle*, as it is called on www.workflowpatterns.com. Although this visualization is intuitive for the business user, the execution semantics are muddled. In particular,

- Because it is the target of more than one source activity, “Approve Invoice” is a join! Specifically, it joins “Receive Invoice” and “Send Check.” What, then, are the semantic rules for its execution? As a join, it should wait for its two source activities to complete. But because “Send Check” comes after it in the process flow, “Approve Invoice” will be deadlocked waiting for “Send Check” to complete.
- On the second iteration of the loop, what is the behavior of “Update Purchase Req”? If the first iteration of that activity is still running, should it be cancelled and restarted, or should it be restarted once it has completed?
- How does the join to “Log Invoice Complete” work? If the second instance of “Send Check” completes before the second iteration of “Update Purchase Req,” is “Log Invoice Complete” smart enough to wait for the latter? Or, lacking any concept of “iteration,” does “Log Invoice Complete” commence execution as soon as *any instance* of its source activities completes?

(For a good discussion of the anomalies of cyclic loops, see Frank Leymann’s WSFL specification, pp.20ff)

The BPMN representation shown uses a structured loop, which avoids the anomalies of the Simple diagram. (BPMN also supports GOTO-style loops, but its structured approach is better form.) The loop is enclosed in a subprocess (“Payment Loop”), which is configured as a standard while-loop (indicated by the counterclockwise arrow). In the overall flow, after “Receive Invoice” finishes, the loop will run as many times as necessary, followed by the execution of “Log Invoice Complete.” Internally, the loop uses an *inclusive gateway* to prevent “Update General Ledger” from running twice; that activity will run only if the condition “Not if interest” is true, whereas its siblings, “Create Check” and “Update Purchase Req,” will run every time, determined by the condition “true.”

The BPMN diagrams presented in these examples are more intelligible than their Simple counterparts, but several of their design aspects will deceive the business user:

- **Sophisticated splits and joins using gateways.** BPMN rivals jewelry stores in its variety of diamonds. Most users have seen in past flowcharts the diamond as an *exclusive OR splitter*; the others uses are obscure.
- **Modularity through subprocesses.** Most business users would have a hard time following the loop process shown above, and would need a step-by-step walkthrough. The technically-savvy designer sees continuity; the average user is confounded by gaps:
 - For the designer, after “Receive Invoice” completes, control moves to the subprocess “Payment Loop,” whose first activity (following the default start event) is “Approve Invoice,” which implies that “Approve Invoice” follows “Receive Invoice.” The business user is troubled by the absence of an arrow connecting

these activities directly.

- For the designer, “Payment Loop” is a while-loop subprocess (as indicated by the loop symbol), each iteration of which begins with “Approve Invoice,” followed by the parallel set of activities shown. The business user expects a loop to be shown with an arrow moving back to an earlier step.
- The designer sees that when the loop completes (i.e., reaches its end event), control flows to “Log Invoice Complete.” The business user is bothered by the absence of an arrow connecting any activity in the “Payment Loop” subprocess to “Log Invoice Complete.”

UML, with its own rich set of symbols, would not fare better than BPMN, nor would any proprietary vendor notation. Business users demand simplicity.

A Mostly Continuous Lifecycle

Keeping it simple has its trade-offs. The simple approach lacks the machinery to build non-trivial common patterns and complicated flows. To expand the language to include more design elements would complicate it to the point where it would no longer resonate with business users. The established process languages, on the other hand, are rife with gadgets, which makes them indispensable to power users (BA’s and technical architects and designers), but foreign to business users. Both approaches are needed in the software lifecycle – the simple approach during requirements gathering, the sophisticated approach for use cases and design.

Figure 6 shows a mostly continuous software lifecycle that uses the simple approach in the requirements gathering phase. The model that is developed in there is passed to the BA writing the use case; the BA breaks the continuum by redrawing the model in BPMN notation (UML 2.0 is another option). The remainder of the cycle is continuous: The architect is also happy working with BPMN (or UML); the transformation to BPEL (a likely choice) for development and test is precise and tool-driven. (In some projects, the BA’s themselves prefer the simple model, in which case the architect is the one to break to the formal model.)

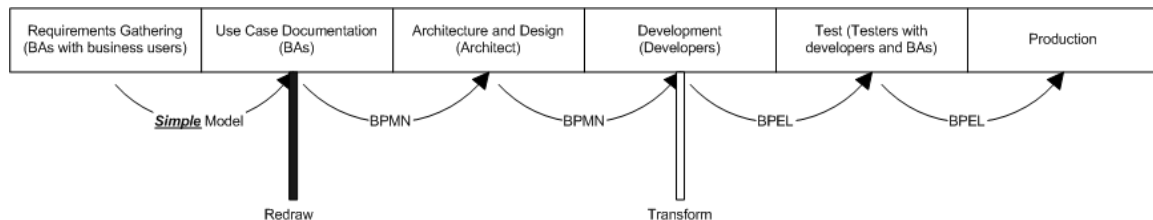


Figure 6. An Achievable Lifecycle

The experiment demonstrates what a simple process language should look like, and why it is inadequate for all but the initial phase of the lifecycle.

References

1. W. M. P. van der Aalst, A. H. M ter Hofstede, B. Kiepuszewski, and A. P. Barrios, “Workflow Patterns,” Technical report, Eindhoven University of Technology, Eindhoven, 2003, *Distributed and Parallel Databases*, 14(1):5–51, 2003. Website: www.workflowpatterns.com
2. Stephen White, “Process Modeling Notations and Workflow Patterns,” *BPTrends*, March 2004.
3. Frank Leymann, “Web Services Flow Language (WSFL 1.0),” IBM, <http://www-306.ibm.com/software/solutions/webservices/pdf/WSFL.pdf>, 2001.