In this package, you will find:

A Biography of the author of the book
A preview chapter from the book, Chapter NO.5 "Short and Long-Running Processes"
A synopsis of the book’s content
Information on where to buy this book

About the Author

Michael Havey started his career as a consultant in the BPM and SOA space thirteen years ago, after many years in the role of 'perpetual student.' He attained undergraduate degrees in Computer Science, Philosophy, and Mathematics (at the University of Toronto), and even spent time studying Philosophy at the Masters level (at Carlton University), before finally resolving to descend the ivory tower and join the workforce. In his professional life, Michael has worked extensively with application integration and business processes. His first book, Essential Business Process Modeling, was published by O'Reilly in 2005. He has also published a number of articles in SOA World, Java Developer's Journal, Weblogic Developer's Journal, BPTrends, and O'Reilly Net.

He is an architect in TIBCO's financial services consulting group. Previously he worked as a consultant for IBM, BEA, and Chordiant.

Without my family's support, I could never have completed this book. Thanks Paola, Napoleon, Katrina, Mason, and Ireland for inspiring me. And thanks to my mother for being my most loyal reader.

Many thanks also to my reviewers, busy fellows to begin with, for graciously accepting my invitation to evaluate the first draft. Their constructive criticism helped me make numerous improvements to the book. Ravi Ravindra, Frank Jennings, and Stuart Charlton: I owe you a favor.

And what a team Packt put together for me! To Adil Ahmed, Nikhil Bangera, Brinell Lewis, Rajashree Hamine, and Mithun Sehgal: You're the best in the business!
SOA Cookbook

Design Recipes for Building Better SOA Processes

SOA Cookbook is a set of recipes for the architecture, design, and simulation of SOA. Perhaps unexpectedly, the recipes teach you about processes rather than services. The central argument of the book is that SOA is fundamentally about processes. The prevailing view of SOA, the one that the textbooks teach, is that the technical implementation of SOA consists of the definition of a set of service operations in a WSDL and the development of an implementation class whose methods implement those operations. The implementation, according to this view, tends to be rather simple, often just a wrapper of existing functionality.

However, as the book argues, few businesses require the development of services defined in this narrow way. On the contrary, they require processes. It's not a coincidence that their use cases invariably express requirements as the flow of control of activities. The leading vendor platforms leave the technical implementers no choice but to build processes anyway. If you buy an SOA platform from Oracle, BEA, IBM, or TIBCO, you quickly discover that development of a service, even a relatively simple one, requires the development of a process. More ambitious orchestration processes, which have a complex control flow and large number of steps, follow the same model.

Vendors blur the distinction between service and process. Developing either of these is about assembling boxes and arrows on a canvas. This point-and-click approach is easier than writing source code, vendors boast. That approach also betrays the process-oriented nature of SOA.

The bulk of the book is a set of hands-on exercises in process-oriented SOA. We build examples using SOA tools from BEA, Oracle, and TIBCO, and we make frequent use of the standard orchestration language, BPEL. If you are new to these technologies, don't despair. The recipes focus on the underlying concepts rather than specific language features. The chapters teach as they go. If you're new to BPEL, for example, this book won't overwhelm you. Before embarking on the book, you would not have known what a pick is, but the book will teach you about picks by showing them in use, as part of examples that demonstrate fundamental concepts in SOA.

As a pre-requisite, you should know the basics of XML and web services. The book is technical in nature, and is written chiefly for developers and architects.

For More Information:
What This Book Covers

In this book, we reveal eight secret recipes for better SOA. SOA Cookbook is a kind of conceptual cookbook for process integration, the equivalent of the sort of recipe book that, in addition to describing how to prepare food, teaches concepts of food that every good chef should know. It's one thing to know the steps to grill a steak, but from better cookbooks we also learn what makes a choice cut and how a steak's marbleisation affects its taste. Each of the eight chapters in SOA Cookbook that follows the introduction provided in Chapter 1 covers an important concept in process-based SOA and teaches techniques for building solutions based on the concept. The chapters are summarized as follows:

Chapter 1 is an introduction to SOA. This chapter looks back at a landmark book on distributed architecture from the CORBA era: Client/Server Survival Guide by Orfali, Harkey, and Edwards. The architecture presented in this earlier work has much in common with contemporary SOA architecture, but it differs in one key respect: the CORBA-based architecture, an object-oriented approach, lacks the sense of process that is so prevalent in SOA. We then examine the contemporary SOA stack (which we call the model stack), and map its layers to the product suites of the four major SOA vendors: IBM, Oracle, BEA, and TIBCO. We look, briefly, at examples of orchestration processes and ESB mediation flows on these platforms. These examples give us a sense of the style of programming on these platforms. In subsequent chapters, we take a deeper dive.

Chapter 2 presents an approach to documenting and diagramming process-oriented SOA architecture using '4+1', ARIS, SCA, UML, and BPMN. With this unusual concoction, we cover all of the important 'views' and draw box-and-arrow process diagrams that carefully link activities to data and services. In our scheme, labeling is an exact science. We discover why the expression Account.getRecord(req.accountNum): acctRec is so much more useful than the casual Get Account Record.

Chapter 3 takes a closer look at the model stack and teaches, by example, how to separate a use case into BPM and SOA parts. We demonstrate two designs for credit card disputes processing: one in which a BPM process manages the end-to-end control flow and uses short-running SOA processes for integration, the other in which a long-running SOA process drives the end-to-end flow but delegates human tasks to BPM processes. This chapter will have you drawing circles in your sleep!

Chapter 4 begins by distinguishing between those oft-confused terms orchestration and choreography, and then presents an approach for modeling choreography, in BPMN and BPEL, as an invisible hub. The leading choreography standard, WS-CDL, is not known for its wealth of implementations; we build the choreography for electricity market enrollment in its leading tool, pi4SOA. The chapter concludes with tips on modeling orchestration; the discussion presents an algorithm for 'dependable' inbound event routing.
Chapter 5 classifies processes by duration, dividing them into three categories: short-running, mid-running, and long-running. Long-running processes need state, so we examine three data models to keep process state: those used in BEA Weblogic Integration and Oracle's BPEL Process Manager, and our own custom model, which borrows ideas from these two. We then discuss how to build a long-running process out of several short-running processes (implemented in TIBCO's BusinessWorks) tied together with state in our custom data model. We conclude by showing how short-running BPEL processes can be compiled for faster execution.

Chapter 6 observes that most processes today are modeled 'naïvely'. Those who design them drag all of the boxes they require onto a canvas, connect them with arrows, and create a graph so meandering and expansive that it's as difficult to navigate as the roads of an unfamiliar city. We propose a structured approach known as flat form, which breaks the graph into simple pieces and assembles them in a controller loop. Flat processes are, by design, flat, and thus avoid the deep nesting characteristic of naïve processes. There are three variants of flat form: event-based, state-based, and flow-based. We build examples of each in BPEL.

Chapter 7 describes the change problem—the problem of changing the definition of a process that has live cases in production—and considers examples of changes (for example, adding an activity, removing an activity, changing the sequence of activities, and introducing parallelism) which cause trouble for existing cases. We also consider dynamic process styles that take the preventative approach to the change problem by attempting to be adaptable in the first place. Dynamic forms can be process-based, rule-based, or goal-based. We study examples of each.

Chapter 8 presents an approach for simulating BPEL processes using concepts from discrete event simulation and the Poisson process. Simulating a BPEL process is fundamentally more difficult than simulating a single-burst service. BPEL processes are long-running, have multiple bursts and both initial and intermediate events, frequently go to sleep for an interval of time, and, in many implementations, queue inbound events rather than responding to them as they come. In this chapter, we build a simulator that supports this usage pattern, run a series of examples through it, and study the results. The salient conclusion is to keep bursts short!

Chapter 9 presents a formula for scoring SOA processes on complexity. We position complexity analysis as an important step in design oversight and governance. The approach we consider allows the governance team to rate each process as red, yellow, or green and to flag reds for rework. Intuitively, the 'complexity' of a process is the amount of branching or nesting in its graph. Flat form, introduced in Chapter 6, scores well on complexity because it avoids excessive branching. Naïve processes score poorly. Our scoring method is a variant of McCabe cyclomatic complexity.
Short and Long-Running Processes

As a process moves from activity to activity, it consumes time, and each activity adds to the overall duration. But different sorts of activities have different durations, and it's not uncommon to observe a ten-step process that outpaces, say, a five-step one. It depends, of course, on what those activities are doing.

In SOA, process cycle times range from one second or less to one or more years! The latter sort need not have a large number of activities. The pyramids might have been built rock-by-rock over several decades, but protracted SOA processes typically span only a few dozen tasks, a handful of which consume almost the entire interval.

As we discuss in this chapter, most of that time is spent waiting. The disputes process introduced in Chapter 3 often requires several months to complete, because at various times it sits idle waiting for information from the customer, the merchant, or the back office. Business processes crawl along at human speed, and, as we argued in Chapter 3, it often makes sense to let SOA manage the end-to-end flow.

It's not easy to build an SOA process engine that can simultaneously blaze through a sub-second process but keep on top of a one that hasn't moved in weeks. On the other hand, when a long-running process rouses, we expect the engine to race very quickly to the next milestone. The central argument of this chapter is that both long-running and short-running processes run in very quick bursts, but whereas a short-running process runs in a single burst, a long-running process might have several bursts, separated by long waits. To support long-running processes, the process engine needs a strategy to keep state.
In this chapter, we examine the fundamental differences between long-running and short-running processes. We discuss how to model state, and demonstrate how to build a long-running process as a combination of several short-running processes tied together by state. We also show how to compile short-running BPEL processes to improve the execution speed of a burst.

**Process Duration—the Long and Short of It**

SOA processes have the following types of activities:

1. Tasks to extract, manipulate, or transform process data
2. Scripts or inline code snippets
3. Calls to systems and services, both synchronous and asynchronous
4. Events, including timed events, callbacks, and unsolicited notifications from systems

The first three sorts of activities execute quickly, the first two in the order of milliseconds, the third often sub-second but seldom more than a few seconds (in the case of a synchronous call to a slow system). These activities are *active*: as the process navigates through them, it actively performs work, and in doing so ties up the process engine. Event times are generally much longer and more variable. Events come from other systems, so (with the exception of timed events) the process cannot control how quickly they arrive. The process *passively* waits for events, in effect going to sleep until they come.

An event can occur at the beginning of a process—indeed, every SOA process starts with an event—or in the middle. An event in the middle is called an *intermediate event*. The segment of a process between two events is called a *burst*. In the following figure, events are drawn as circles, activities as boxes, and bursts as bounding boxes that contain activities. Process (a), for example, starts with an event and is followed by two activities—*Set Data* and *Sync Call*—which together form a burst. Process (b) starts with an event, continues with a burst (consisting of the activities *Set Data* and *Call System Async*), proceeds to an intermediate event (*Fast Response*), and concludes with a burst containing the activity *Sync Call*. Process (c) has two intermediate events and three bursts, and (d) has a single intermediate event and two bursts.
Processes are classified by duration as follows:

- **Short-running**: The process runs comparatively quickly, for not more than a few seconds. Most short-running processes run in single burst (as in process (a) in the figure), but some have intermediate events with fast arrival times—as in (b), where the intermediate event, a response to an asynchronous system call, arrives in about two seconds—and thus run in multiple bursts. TIBCO's BusinessWorks and the BPEL compiler described later in the chapter are optimized to run both single-burst and multiple-burst short-running processes. BEA's Weblogic Integration can run single-burst, short-running processes with limited overhead, but, as discussed further next, treats cases like (b) as long-running.

- **Long-running**: The process has multiple bursts, and the waiting times of its intermediate events are longer than the process engine itself is expected to run before its next restart! In process (d), for example, the engine is restarted for maintenance while the process waits two days for a human action. The process survives the restart because its state is persisted. At the end of its first burst (that is, after the Assign Work step), the engine writes the state to a database, recording the fact that the process is now waiting on an event for a human action. When the engine comes back up, it fetches the state from the database to remember where it left off. Most BPEL processes are long-running. In Weblogic Integration, stateful processes can run for arbitrarily long durations.
• **Mid-running:** The process has multiple bursts, but the waiting times of its intermediate events last no more than a few minutes, and do not need to be persisted. Stakeholders accept the risk that if the process engine goes down, in-flight processes are lost. Chordiant's Foundation Server uses mid-running processes to orchestrate the interaction between agent and customer when the customer dials into a call center. The call is modeled as a conversation, somewhat like a sequence of questions and answers. A burst, in this design, processes the previous answer (for example, the Process Answer activity in (c)) and prepares the next question (Prepare Question). Intermediate events (Get Answer) wait for the customer to answer. State is held in memory.

**Stateful and Stateless Processes in BEA's Weblogic Integration**

In Weblogic Integration, single-burst processes are *stateless*, but multiple-burst processes, even short-running ones, are *stateful*. Even if the wait between bursts is very small (one or two seconds perhaps), Weblogic Integration nonetheless persists process state to a database. The distinction is subtle, but Weblogic Integration provides visual clues to help us detect the difference. In the next figure, the process on the left is stateless. The process on the right is the same as that on the left except for the addition of an event step called Control Receive; the step, in effect, puts the process in a wait state until it receives a specific event. When this step is added, Weblogic Integration changes the appearance of its start step—Start—from a circle with a thin border to one with a thick border, indicating that the process has changed from being stateless to stateful.
Those who designed Weblogic Integration thought process state so important that they worked into their notation whether a process is stateful or stateless. We now study one of the most critical pieces of any process engine: how it keeps state.

**How to Keep Long-Running State**

In this section, we study the data models for long-running process state in two commercial process integration platforms: Oracle's BPEL Process Manager and BEA's Weblogic Integration. We also develop our own model, a generalization of the Oracle and BEA approaches, which enables us to achieve the effect of a long-running SOA process from a group of short-running processes. We put this model to practical use later in this chapter, in the email money transfer example.

SOA process state models contain information about the following:

- Process metadata, including the types of processes currently deployed, their versions, and how their activities are assembled.
- Process instances, including status, start time and end time, and the position of the instance in a call graph (that is, parent/child relationships). Some models also track the status of individual activities.
- Pending events, and how to correlate them with process instances.

**State in Oracle's BPEL Process Manager**

The following figure shows the core tables in the Oracle BPEL model (version 10.1.2).
Short and Long-Running Processes

In this model process, metadata is held in two tables: `Process_Default` and `Process_Revision`. The former lists all deployed BPEL processes and their current revision numbers; the `process_id` field is not a technical key but the name of the process specified by the developer. The latter lists all the revisions; for a given process, each revision has a distinct GUID, given by the field `process_guid`.

The seemingly-misnamed table `Cube_Instance`—actually, `cube` is synonymous with `process` in the internals of the product—has information about current and completed process instances. The instance has a unique key, given by `cikey`. From `process_guid` we can deduce, by joining with `Process_Revision`, the process type and revision of the instance. Other important information includes the instance creation date, its parent instance, and its current state. Possible states are `active`, `aborted`, `stale`, and `completed`, although the `state` field uses numeric codes for these values.

The `Work_Item` table tracks the status of instance activities. `Cikey` indicates the instance to which the activity belongs. Within an instance the activity is identified by the combination of `node_id`, `scope_id`, and `count_id`. The first two of these indicate the position of the activity in the process graph and the scope level to which it belongs; the `label` column is a friendlier alternative to these, assuming that the developer applied a useful label to the activity. `Count_id` is required in case the activity executes more than once. `Work_Item` has its own `state` field (again numeric), which indicates whether the activity is completed or pending, was cancelled, or encountered an exception.

Dlv Subscription records pending events and correlates them with instances. `Conv_id` is a conversation identifier known to both the BPEL process and its partner service. To trigger the event, the partner service passes this identifier as part of its message. The process matches it to a `subscriber_id`, which uniquely identifies the activity that is waiting on the event. Thus, when the event arrives, the process knows exactly from which point to continue. (Technically, `subscriber_id` is a delimited string, which encodes as part of its structure the values of `cikey`, `node_id`, `scope_id`, and `count_id` that point to a unique `Work_Item` record.) The partner also specifies an operation name, which specifies which type of event it is firing. If the process is waiting on several events in the same conversation (as part of an event pick, also known as a deferred choice), `operation_name` determines which path to follow. The combination of `operation_name` and `conv_id` points to a unique activity (that is, to a unique `subscriber_id`).
State in BEA's Weblogic Integration

The following figure shows three important tables in the Weblogic Integration model:

<table>
<thead>
<tr>
<th>WLI_Process_Def</th>
<th>WLI_Process_Instance_Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>PK process_type</td>
<td>PK process_instance</td>
</tr>
<tr>
<td>activity_id</td>
<td></td>
</tr>
<tr>
<td>user_node_name</td>
<td></td>
</tr>
<tr>
<td>flow_tag_name</td>
<td></td>
</tr>
<tr>
<td>parent_activity_id</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>WLI_Message_Broker_Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>PK rule_name</td>
</tr>
<tr>
<td>PK subscriber_instance</td>
</tr>
<tr>
<td>filter_value</td>
</tr>
</tbody>
</table>

WLI_Process_Def has metadata about types of deployed processes and their activities. The table has one row for each activity. Process_type is the human-readable name of a process. Activity_id is the numeric identifier of an activity in the process, although user_node_name, the descriptive name provided by the developer is more intuitive.

Process instance information is held in WLI_Process_Instance_Info. Each instance has a unique numeric identifier, given by process_instance. Process_type specifies the process definition on which the instance is based. Process_status specifies, in a numeric code, whether the instance is active, pending, or aborted. The table also tracks process start and end times, as well as time in excess of the SLA (sla_exceed_time). Through Weblogic Integration's administration console, the administrator can configure an SLA on process cycle time.

In Weblogic Integration a process instance can receive intermediate events by several means. One of the most important of these is by listening for messages published by Weblogic Integration's message broker system. The table WLI_Message_Broker_Dynamic keeps track of specific events waiting on broker messages. The column subscriber_instance is the process instance identifier; it matches the process_instance value in WLI_Process_Instance_Info. Rule_name is, in effect, a pointer to the event in that instance. Filter_value is an XQuery expression that checks the content of the message to determine whether to accept the event. When a message arrives, the broker checks for any subscription events, and triggers those whose filter test passes.
Our Own State Model

Our own model, shown in the next figure, follows a design approach similar to that of the Oracle and BEA models.

To begin, the model features a single metadata table, called ProcessStarter, which enumerates the types of processes deployed (processType) and specifies for each the type of event that can start it (triggeringEventType). The table's main purpose is to route start events: when an external event arrives, if ProcessStarter can map it to a process, then a new instance of that process is created from the event.

Several tables track the state of process instances. The Process table assigns a unique identifier to each instance (procID), indicates its type (processType), locates it in a conversation (convID), and records its start time, end time, and status (pending, completed, or aborted). The ProcessVariable table persists process variables, ensuring that instance-specific data survives system restarts. A variable is identified...
by a name (name) that is unique within its level of scope (scope) in a process instance (procID). The ProcessAudit table keeps a chronological list of important occurrences in a process instance. It is tied to a specific instance (procID), and has both a timestamp and a text entry. The entry can optionally be associated with a specific process activity (activityID). Implementations can extend the model by providing a custom state table (such as the hypothetical MyAppState in the diagram) that associates application-specific fields (myState, in this example) with an instance.

Finally, the PendingEvent table assists in correlating intermediate events. An event is identified by the combination of its process instance (procID), its activity node in the process (activityID), and if it is part of a deferred choice, the identity of that choice (choiceActivityID). (If the event is not part of a choice, choiceActivityID is zero or null.) There are two types of events: timed events and events triggered by a message. If the event is a timed event, timeToFire specifies the date and time (somewhere in the future) when the event should fire. If the event is message-based, triggeringEventType indicates the type of message that triggers it. When the event is created, the Boolean field isDone is set to false. When the event fires, isDone is switched to true. If the event is part of a choice, isDone is set to true for all events in the choice, thereby ensuring that only one event is chosen.

The model assumes that all messages carry the following fields:

- Event Type
- Recipient Process Type
- Conversation ID

When a message arrives, the following logic determines how to route it:

- If there is an instance of the process in the conversation (that is, if there are rows in Process where processType and convID match the values from the message), check whether it has a pending event of the given event type (that is, check for rows in PendingEvent where procID matches the value from Process, isDone is false, and triggeringEventType matches the event type). If it does, fire the event. Otherwise, discard the event.
- If there is no instance of the process in the conversation, check whether the process can be started by this type of event. (That is, check for rows in ProcessStarter where processType and triggeringEventType match those from the message.) If so, instantiate the process. Otherwise, discard the event.

We put this model to use in the next section. Refer to the discussion of correlation in Chapter 4 for more details on this approach, especially the use of optimistic locking to prevent two simultaneous events from firing.
Combining Short-Running Processes with State in TIBCO's BusinessWorks

The next discussion covers the TIBCO implementation of the email transfer process.

Our Use Case—Sending Money by Email

With this model in place, we build a process that spans several days as a set of short-running processes, none of which lasts more than a few seconds. The use case we consider is email money transfer, introduced in our discussion of choreography in Chapter 4. In a transfer there are four main parties: the sender, the sender’s bank, the recipient, and the recipient’s bank. We build the process for the sender's bank.

The following figure depicts the required flow of events:

When the bank receives the request to send funds from the sender (Sender’s Request), it validates the request (Validate Request), and rejects it if discovers a problem (Send Reject to Sender). If the request is valid, the bank informs the sender of its acceptance (Send Accept to Sender), notifies the recipient by email (Send Email To Recipient), and sets aside funds from the sender's account (Allocate Funds). The first burst is complete, but several possible paths can follow:

1. There is a time limit on the transfer, and if it expires the transfer is aborted.
2. The sender may cancel the transfer.
3. The sender's bank may reject the recipient's bank's request to move the funds into the recipient's account. The recipient may try again later.

4. The sender's bank may accept the recipient's bank's request to move the funds into the recipient's account.

The control flow to support this logic is a deferred choice inside a loop. The loop runs for as long as the variable `loopExit` is false. The process initializes the value to false (Set `loopExit=false`) immediately before entering the loop. Paths 1, 2, and 4 set it to true (Set `loopExit=true`) when they complete, signaling that there is no further work to do and the loop need not make another iteration. Path 3 leaves the `loopExit` flag alone, keeping it as false, thus allowing another iteration (and another chance to complete the transfer). Each iteration is a burst.

There are three events in the deferred choice, one for expiry (path 1), one for cancellation (path 2), and one for the recipient's bank transfer request (paths 3 and 4). The logic for cancellation and expiry (headed by the events `Sender's Cancellation` and `Expired` respectively) is identical: the process sends a cancellation email to the recipient (Send Email Recipient), informs the sender that the transfer is aborted (Send Abort to Sender), and restores the funds to the sender's account (Restore Funds). In the transfer request path (starting with the event `Recipient Bank's Transfer Request`), the sender bank validates the transfer (Validate Transfer) and sends the outcome to the recipient's bank (Send Reject to Recipient Bank or Send Accept to Recipient Bank). If validation passes, the process also notifies the sender that the transfer is complete (Send Completion to Sender) and commits the funds it had earlier allocated (Commit Funds).

The sender's bank's process is long-running, typically spanning several days from start to finish. To build it using a short-running process engine, such as TIBCO's BusinessWorks, we need to break it into smaller processes: one to handle the sender's request to send funds, one to handle the recipient's bank's request to complete the transfer, one to handle the sender's cancellation, one to handle expiry, and one to manage the overall event routing. In dividing the process into pieces, we lose the loop and deferred choice, but we add housekeeping responsibility to each piece.
The Router Process

The next figure shows the BusinessWorks process to handle the overall routing.

When it receives an inbound message on a JMS queue in GetEvent, the router process checks the event type to determine to which BusinessWorks process to route the event. There are three event types:

- **Request**: Sent by the account holder (known as the sender). Because this request starts the process, it must not contain a conversation identifier. If it does, the route process immediately logs the event as an error and discards it (Log Illegal Input). Otherwise, it queries the ProcessStarter table, in the step Check Starter Enabled, to verify that the email transfer process may be started by this type of event. (It checks that there is a record in the table that matches the given event type and process type.) If this check passes, the route process creates a unique conversation identifier (Set Conv ID) and calls the request process to handle the event (Call Request Process).

- **Transfer**: Sent by the recipient bank. The route process checks that the message has a conversation identifier. If it does, it calls the transfer process (Call Transfer Process) to handle the event. Otherwise, it logs the event and discards it (Log Illegal Input).
- **Cancel**: Sent by the sender or internally by the timer process (discussed further next). The route process checks that the message has a conversation identifier. If it does, it calls the cancellation process (**Call Cancel Process**) to handle the event. Otherwise, it logs the event and discards it (**Log Illegal Input**).

### The Request Process

The next figure shows the BusinessWorks process to handle the sender's request to send funds:

The process begins by creating a unique process identifier (**Set Proc ID**) and then validates the request (**Validate Request**). If the request is invalid, the process sends a rejection to the sender (**Send Reject to Sender**) and writes three records to the database:

- A record in the **Process** table (using **Add Process Record Aborted**) that sets the status of the instance to **ABORTED**. The process identifier is the one created in **Set Proc ID**.
- A log of the validation failure (using **Add Audit Invalid Req**) in the **ProcessAudit** table.
- A copy of the inbound message in the **ProcessVariable** table, using **Add Variable Request**. The earlier step **RequestAsString** converts the message from XML to string form.
Thus, there is a record that the instance was aborted, an explanation in the audit trail why it failed, and a copy of its message data.

The happy path, in which the request passes validation, contains three steps that we described earlier: Send Email Recipient, Send Accept to Sender, and Allocate Funds. It also creates the following records in the database:

- A record in the Process table (using Add Process Record Pending) about the instance, with a status of PENDING and the identifier created in Set Proc ID.
- An indication that the validation passed (using Add Audit Valid Request) in the ProcessAudit table.
- A copy of the inbound message (using Add Variable Request 2) in the ProcessVariable table.
- Three PendingEvent records, for transfer, expiry, and cancel respectively (using the steps Add Transfer Event, Add Expiry Event, Add Cancel Event). The records share a common choiceActivityID, and for each the isDone field is set to false.
- A record in the custom table EXState (using Add EXState), which extends the Process table with information specific to email transfers. The next figure shows the EXState table and its relationship to Process. The table adds one field to the mix, numRejects, which is initialized here to zero and is incremented each time the sender's bank rejects the recipient's bank's transfer request.

```
<table>
<thead>
<tr>
<th>PK</th>
<th>Process</th>
<th>PK,FK1</th>
<th>EXState</th>
</tr>
</thead>
<tbody>
<tr>
<td>procID</td>
<td>convID</td>
<td>procID</td>
<td>numRejects</td>
</tr>
<tr>
<td>procID</td>
<td>startTime</td>
<td>procID</td>
<td>status</td>
</tr>
</tbody>
</table>
```

When the happy path completes, the PendingEvents table has, among its contents, three records similar to the following:

<table>
<thead>
<tr>
<th>Proc ID</th>
<th>Activity ID</th>
<th>Choice Activity ID</th>
<th>Is Done</th>
<th>Time To Fire</th>
<th>Triggering Event Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>Cancel</td>
<td>1</td>
<td>False</td>
<td>-</td>
<td>EX.Cancel</td>
</tr>
<tr>
<td>123</td>
<td>Expiry</td>
<td>1</td>
<td>False</td>
<td>Dec 13, 2008</td>
<td>-</td>
</tr>
<tr>
<td>123</td>
<td>Transfer</td>
<td>1</td>
<td>False</td>
<td>-</td>
<td>EX.Transfer</td>
</tr>
</tbody>
</table>
According to this information, process instance 123 has three pending events, whose activityIDs are Cancel, Expiry, and Transfer respectively. These events are set in a single deferred choice, whose choiceActivityID is 1. None of these events has occurred, indicated by isDone being false. The Cancel and Transfer events are triggered by the inbound events types EX.Cancel and EX.Transfer respectively. The Expiry event does not have a triggering event type, but has a timeToFired configured for December 13, 2008; Expiry is a timed event.

When one of these events arrives, it is processed only if the isDone field is false; otherwise it is discarded. When it is processed, the isDone flag is set to true for all three events. Marking all three true in effect marks the whole deferred choice as complete, and prevents a second event from occurring.

The Transfer Process

The process that handles the recipient’s bank’s request for transfer is shown in the following figure.
Short and Long-Running Processes

The process begins immediately by querying the PendingEvent table to check that its event is still pending (FindEvent). If it has already been marked as completed, the process rejects the request (Send Reject to Recipient Bank Event Not Found) and quits. Assuming the event is permitted, the process marks the choice as completed (Remove Event) and validates the request (Validate). If validation passes, the process, as already discussed, sends an acceptance to the recipient's bank (Send Accept Recipient Bank) and a completion notification to the sender (Send Completion Sender), commits the funds (Commit Funds), and then performs the following table updates:

- In the Process table, it sets the instance status to COMPLETED (using Close Process).
- It adds an entry to the ProcessAudit table (using Add Audit), indicating that the transfer succeeded.
- It saves the transfer request message to the ProcessVariable table. If a previous version of the message is already there, the process overwrites it (Update Variable); otherwise, it inserts a new message (Insert Variable).

If validation fails, the process sends a rejection message to the recipient bank (Send Reject Recipient Bank) and makes four table updates:

1. It restores the deferred choice (using Restore Event), setting isDone to false for each of the three events (Restore Event).
2. It increments the numRejects field in the EXState table (Add Reject).
3. It adds an entry to the ProcessAudit table (using Add Audit), indicating that the transfer failed.
4. It saves the transfer request message to the ProcessVariable table, using the same logic as above.

The successful validation path effectively terminates the larger process by removing all of its pending events. The failed validation path effectively loops back in the larger process to an earlier point, giving each of the events another chance to fire.

The Cancellation Process

The process to handle cancellation, shown in the next figure, starts out much the same way.
The process first checks that the event is still pending (Find Event), and if so, disables the deferred choice (Remove Event). The process then notifies the sender and the recipient of the cancellation (Send Recipient Email and Send Abort to Sender), restores the funds (Restore Funds), and update the tables as follows:

- It marks the status of the instance as ABORTED (Close Process).
- It adds an audit entry indicating cancellation (Add Audit).
- It saves the cancellation event to the ProcessVariable table (Save Variable).

**The Expiration Process**

The process to handle expired transfers, shown in the next figure, is somewhat different.
Short and Long-Running Processes

The expiration process is not designed to handle the expiry of a single transfer. Rather, it scans the PendingEvents table for all expired transfers (Get Expired Transfers), and fires a cancellation event for each of them. The outer box labeled For Each Expired is a for loop that, for each record returned by the query, constructs a cancellation message (Create Cancellation Message) and launches a cancellation process (Launch Cancellation Process) to handle the message. It launches the process by sending a message on the JMS queue to which the routing process listens. The routing process, when it receives the event, routes it to the cancellation process. Thus, it is the cancellation process that will disable the deferred choice and abort the instance, not the timer process.

A Note on Implementation

TIBCO's BusinessWorks is designed for performance, and admittedly our processes make database updates rather liberally. (The request process has seven updates in the happy path!) More efficient alternatives are to flatten the data model (so that there are fewer tables to update) or build stored procedures to bundle updates (resulting in less IO to the database server).

Another option is use TIBCO's proprietary checkpoint mechanism to serialize process state to the disk. The checkpoint feature is clumsy but is often an efficient way to achieve the effect of long-running state in an engine that is designed for short-running processes. As a proprietary capability, it does not work as part of a generalized state model, which is why we did not demonstrate it here.

Fast Short-Running BPEL

We conclude with a discussion of compiled BPEL.

Uses of Short-Running Processes

Having developed an approach to keep SOA processes running for an arbitrarily long time, we now turn our attention to short-running processes and ask: how can we make them run as fast as possible? The two most common uses of a short-running process are:
1. To implement a synchronous web service operation. The process begins with an input message, runs through a quick burst of logic to process it, sends back the output message, and completes. The client application blocks for the duration, as diagram (a) in the next figure shows. If the process moves too slowly, the client will complain about the response time.

2. To perform complex routing for the ESB. As David Chapelle discusses in his book *Enterprise Service Bus* (O'Reilly, 2004), a good ESB can natively perform basic content-based- and itinerary-based-routing, but it needs orchestration processes to handle more complex routing patterns. In diagram (b) in the figure, when the ESB receives a message, it passes it to an orchestration process that proceeds to perform in eight steps a series of transformation and invocation maneuvers that could never be achieved with the basic branching capabilities of the ESB. Again, speed is critical. The ESB prefers to get rid of messages as soon as it gets them. When it delegates work to an orchestration process, it expects that process to move quickly and lightly.
Architecture for Short-Running Processes

In considering a design to optimize the performance of these two cases, we assume that our stack, like the model stack we discussed in Chapters 1 and 3, has both an ESB and a process integration layer. All messages in and out of the stack go through the ESB. The ESB, when it receives an inbound message, routes it to the process integration engine for processing. The process integration engine, in turn, routes all outbound messages through the ESB. Further, we assume that the ESB uses message queues to converse with the process integration layer. Client applications, on the other hand, typically use web services to converse with the ESB.

The following figure shows how we might enhance this architecture for faster short-running processes. (The implementation we consider is a Java-based BPEL process engine.)
When a client application or partner process calls through the ESB, the ESB routes the event, based on the event's type, either to the general process integration engine or to an engine optimized for short-running processes. To route to the general engine, the ESB places the message on the Normal PI In Queue. That engine is drawn as a cloud; we are not concerned in this discussion with its inner workings. To route to the optimized engine, the ESB either queues the message on SR In Queue or, to reduce latency, directly calls the short-running engine's main class, ProcessManager. (Direct calls are suitable for the orchestration routing case described in the previous figure; there, processes run as an extension of the ESB, so it makes sense for the ESB to invoke them straightaway.) A set of execution threads pulls messages from SR In Queue and invokes ProcessManager to inject these inbound events to the processes themselves. The role of ProcessManager is to keep the state of, and to execute, short-running processes. Each process is represented in compiled form as a Java class (for example, ProcessA or ProcessB) that inherits from a base class called CompiledProcess. Compiled classes are generated by a tool called BPELCompiler, which creates Java code that represents the flow of control specified in the BPEL XML representation of the process. ProcessManager runs processes by creating and calling the methods of instances of CompiledProcess-derived classes. It also uses TimeManager to manage timed events. Processes, whether running on the general engine or on the optimized engine, send messages to partners by placing messages on the outbound queue Out Queue, which the ESB picks up and routes to the relevant partner.

A general process engine is built to handle processes of all durations, long and short alike, and, with a mandate this extensive, does not handle the special case of time-critical short-running processes very effectively. There are three optimizations we require, and we build these into the short-running engine:

1. Process state is held in memory. Process state is never persisted, even for processes with intermediate events. Completed process instances are cleaned out of memory immediately, so as to reduce the memory required.

2. Processes are compiled, not interpreted. That is, the process definition is coded in Java class form, rather than as an XML document. Compilation speeds the execution time of a burst.

3. The process may define timed events of a very short duration, to the order of milliseconds. Furthermore, the engine generates a fault when the process exceeds its SLA. The process may catch the fault or let it bubble up to the calling application.

The architecture we sketched in this section, as we discover presently, is designed to meet these requirements.
**Example of a Very Fast Process**

The next figure shows a short-running process with multiple bursts that benefits from these optimizations.

When the process starts, it initializes its variables (`InitVars`) and asynchronously invokes a partner process called the Producer (`Call Producer Async`). It then enters into a loop (`FetchLoop`) that, on each iteration, waits for one of the two events from the Producer: `result` or `noMore`. If it gets the `result` event, it, in parallel, invokes two handler services (`Call Handler A` and `Call Handler B`), and loops back. If it gets the `noMore` event, the process sets the loop's continuation flag to false (`Set Loop Stop`). The loop exits, and the process completes. While it waits for the producer events, the process also sets a timed event (`too long`) that fires if neither event arrives in sufficient time. If the timer expires, the process sends an exception message to the producer (`Send Exception Msg Producer Async`), and loops back.

The timing characteristics are shown in parentheses. The producer, on average, sends a `result` or `noMore` event in 80 milliseconds. The handlers that the process invokes to handle a `result` event average 50 milliseconds and 70 milliseconds, but because they run in parallel, their elapsed time is the greater of these two times, or 70 milliseconds. Thus, an iteration of the loop with a `result` event averages roughly 150 milliseconds. An iteration with a `noMore` event averages just 80 milliseconds, because the activity `Set Loop Stop` runs nearly instantaneously. The cycle time of an instance with one `result` iteration and one `noMore` iteration is just 220 milliseconds. The `too long` timed event has a duration of 200 milliseconds, which in itself is rather a small interval, but is a huge chunk of time compared to the normal cycle time. The cycle time of an instance whose three intermediate events are `result`, `too long`, and `noMore` is 420 milliseconds on average. Times this fast cannot be achieved on a general-purpose engine.
Running the Very Fast Process on the Optimized Engine

The sequence diagram in the following figure illustrates how this process runs on the short-running engine:
Short and Long-Running Processes

The process starts when client application sends a message intended to trigger the process' start event. The ProcessManager receives this event (either as a direct call or indirectly via an execution thread that monitors the short-running inbound queue) in its routeMessageEvent() method. It then checks with the process class—shown as Process in the figure, a subclass of the CompiledProcess class we discuss presently—whether it supports the given start event type (hasStartEvent()). If it does, the ProcessManager invokes the process (onStartEvent()). The process, as part of its logic, performs the activities InitVars and CallProducerAsync and enters the first iteration of the while loop, in which it records in its data structures that it is now waiting for three pending events (Set Pending Events). Because one of these events is a timed event, it also registers that event with the TimeManager (addEvent()). The first burst is complete.

In the second burst, the producer process responds with a result event (result: routeMessageEvent()). The ProcessManager checks whether the process instance is waiting for that event (hasPendingEvent()) and injects it (onIntermediateEvent()). The process invokes the two handlers (that is, it invokes CallHandler on HandlerA and HandlerB), completing the first iteration of the loop. It now loops back, resets the pending events (Set Pending Events), and registers a new timed event (addEvent()). The second burst is complete.

Assuming the producer does not respond in sufficient time, the timer expires, and the TimeManager, which checks for expired events on its own thread notifies the ProcessManager (routeTimedEvent()). ProcessManager gives the event to the process (calling hasPendingEvent() to confirm that the process is waiting for it and onIntermediateEvent() to inject it), and the process in turn performs the SendExceptionMsg activity, completing the second iteration of the loop. The next iteration starts, and the process resets its pending events. The third burst is complete, and we leave it there.

Managing Inbound Events and Timeouts

The state information needed to tie all of this together is held in memory. ProcessManager maintains a data structure called instanceList that, much like the Process table just described, keeps a list of process instances indexed by the combination of conversation identifier and process type. The list contains references to CompiledProcess-derived objects. The logic for routeMessageEvent(), in pseudo code, is the following:

Does instanceList have instance for specified process type and conv ID?

If no
   Instantiate this instance
   Create a unique PID
Add instance of instanceList
Call process.hasStartEvent() to check if proc supports
start event of specified type
If no, return error
Else
   Call process.onStartEvent()
End If
Else
   Call process.hasPendingEvent() to check if proc supports
   intermediate
   Event of specified type
   If no, return error
   Else
   Call process.onIntermediateEvent()
End Id
End If

TimeManager keeps a list of timed events, each tied to a particular wait node in
a process instance. TimerManager's thread periodically sweeps through the list,
finding events that have expired. It calls ProcessManager's routeTimedEvent() method
injected to inject the event to the instance. Three types of timed events are supported:

- wait activity
- onAlarm activity
- SLA on the instance

The first two event types simply wake up the process. If the process previously
entered a wait activity, for example, the timed event causes it to complete. The
third generates a fault. If the process has a handler for this fault, control moves
immediately to the handler. Otherwise, the instance is immediately aborted.

Compiled Form
The CompiledProcess class (the base class for compiled BPEL processes) keeps track
of variables, current pending events, and permitted start events, holding in memory
the same sort of data that is defined for the tables ProcessVariable, PendingEvent,
and ProcessStarter. Here is an excerpt of the code:

```java
public abstract class CompiledProcess {

   List pendingEvents = new ArrayList();
   Map variables = new HashMap();
   String pid;
   String convID;

   public abstract BPELGraph getGraph();
```
Short and Long-Running Processes

```java
public boolean hasStartEvent(...) {
    check graph to see if specified event is allowed
}

public boolean hasPendingEvent(...) {
    check pendingEvents list to see if specified event is allowed
}

public void onStartEvent(...) {
    walk();
}

public void onIntermediateEvent(...) {
    remove event from pending events
    walk();
}

void walk() {
    From the node containing the current event, navigate forward in
    the graph until the process completes or we hit another
    intermediate event.
    When we hit an intermediate event, add it to the pendingEvents
    list.
    For timed events, register them with TimerManager too.
    Update process variables as needed.
}
}
```

Notice that the class is marked `abstract`, and that its method `getGraph()` is not
implemented. In our design, each BPEL process is run through a special compiler
utility that generates a Java class extending `CompiledProcess`. The utility, called
`BPELCompiler`, is a Java program that takes as input the XML source code for the
BPEL process. It parses the XML and outputs a Java source file that is later compiled
and loaded into the address space of the process engine. At runtime, the BPEL
process runs at the speed of compiled Java. We thus save the performance-stultifying
effect of runtime XML parsing and serialization that afflicts many process engines.

Here is a snippet of the Java source of the class for our sample short-running process:

```java
public class SRProcess extends CompiledProcess {
    static BPELGraph graph = null;

    static {
        graph = new BPELGraph();
        graph.initGraph();
        graph.addSequence();
        graph.addArc(1,3);
    }
}
```

For More Information:
The class does nothing except build a graph representing its process definition. It begins by declaring a class-scoped member variable called `graph` (static `BPELGraph graph = null`). In the static initializer code that follows (beginning with `static {`), it instantiates this attribute (`graph=new BPELGraph();`) and proceeds to construct it as a set of nodes (for example, `graph.addSequence()`, `graph.addReceive()`, `graph.addAssign()`, `graph.addInvoke()`, `graph.addWhile()`, `graph.addPick()`, and others not shown) and arcs (`graph.addArc()`). The class also overrides the `getGraph()` method that is left abstract in the base class. This method simply returns a reference to the `graph` variable.

And that's all there is to the generated class. It inherits the most important methods from the base class. Its job is to fill in the one missing ingredient: the actual process definition. Significantly, it creates this definition (that is, the graph) at class scope, so that there is only one copy of it in the process engine, not one copy per process instance. This saves a lot of memory.
Short and Long-Running Processes

The structure of the graph is similar to that of the XML-defined process in the source—which is not surprising given that this code is generated from a parse of the XML. The next figure depicts the graph constructed in the compiled process.

Here is a snippet of the corresponding BPEL source, predictably similar to the graph:

```xml
<process>
  <sequence>
    <receive>
      <assign name="InitVars" />
      <invoke name="Call Producer Async" />
      <while>
        <pick>
          <onMessage name="result">
            <flow>
              <invoke name="Call Handler A" />
              <invoke name="Call Handler B" />
            </flow>
          </onMessage>
          <onMessage name="noMore">
            <assign name="Set Loop Stop" />
          </onMessage>
          <onAlarm name="Too Long">
            <invoke name="Send Exception Msg" />
          </onMessage>
        </pick>
      </while>
    </receive>
  </sequence>
</process>
```
The surest way to learn the functionality of compiled processes and the short-running engine is to play with the accompanying compiler demo. See About the Examples for a download link.

## Compiled Code—What Not To Do

An alternative to the graph implementation is to represent the process as a single block of code, as follows:

```plaintext
InitVars();
CallProducerAsync();
While (loopContinue)
  WaitNextEvent()
  If (event is result)
    Fork (CallHandlerA)
    Fork (CallHandlerB)
    Join the forks
  Else if (event is noMore)
    Set loopContinue = false
  Else if (event is too slow)
    SendExceptionMsg();
  End If
End While
```

Though simple, this code hampers performance, because the intermediate event in `WaitNextEvent()` ties up an execution thread while it waits. That's one less thread for the process engine to work with, which might be needed elsewhere. The graph implementation might be a little harder to code—*that code is generated by a tool anyway*—but it uses resources more efficiently. Performance is the point, after all.

## About the Examples

The source code for this chapter is available for download in the code bundle. Refer to the `README` file for information on how to set up and run the examples.

The example of email funds transfer, which demonstrates how to build a long-running process out of several short-running processes, uses TIBCO's BusinessWorks 5.6 and Enterprise Message Service 4.4, as well as an RDBMS. TIBCO products can be downloaded from [http://download.tibco.com](http://download.tibco.com). You must have an account to access this site. Once in, there are several installation programs to download; refer to our `README` file for the complete list.
The BPEL compiler is a set of Java programs. To run them, you require JDK 1.4 or higher. If you wish to compile the source code or run the programs from Eclipse, you need Eclipse 3.0 or later.

**Summary**

SOA processes have both active and passive activities. Active activities include calls to systems and services, data manipulations and transformations, and scripts or inline code snippets. Passive activities are *events*. When performing active activities, the process is actively performing work, tying up the process engine. Events put the process into an idle wait state.

An event can occur at the beginning of the process or in the middle. Every SOA process starts with an event. An event in the middle is called an *intermediate* event, and not every SOA process has one. The segment of a process between two events is called a *burst*; in a burst, the process performs active activities.

Processes are classified by duration as *short-running*, *long-running*, or *mid-running*.

Short-running processes span no more than a few seconds. Many short-running processes are a single burst, but some have intermediate events, which break the process into multiple bursts. Languages that support short-running processes include TIBCO's BusinessWorks and BEA's Weblogic Integration.

Long-running processes run longer—often days, weeks, months, or years—than the uptime of the process engine on which they run. Most of the time is spent waiting on intermediate events; the bursts themselves are quick. The engine persists the state of such processes to a database to survive a restart. Languages that support long-running processes include BPEL and Weblogic Integration.

Mid-running processes run for about the duration of a phone call in a call center. In call center usage, processes are structured as question-and-answer conversations between agent and customer. Bursts process the previous answer and prepare the next question; intermediate events wait for the customer’s next answer. The engine keeps process state in memory. If the engine goes down, in-flight instances are lost. Chordiant’s Foundation Server is an example of this sort of implementation.

Process data models include process metadata (information about the types of processes currently deployed), instance data (the status of live instances of processes), and pending events (and how to correlate them with instances). We studied the data models in Oracle's BPEL Process Manager and BEA's Weblogic Integration, and developed our own model that generalizes these. We used this model to build a use case that requires a long-running process (email funds transfer) from several short-running processes in TIBCO's BusinessWorks.
We concluded by designing a process engine optimized for short-running processes. The design is able to run short-running processes faster than a typical process engine because process state is held in memory (never persisted), processes are compiled rather than interpreted, and the process may define timed events of a very short duration. Further, the engine generates a fault when the process exceeds its SLA; the process may catch the fault or let it bubble up to the caller.
Where to buy this book

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.