Freeway Patterns for SOA Systems

Vinod Sarma N
MindTree Ltd.
Global Village, RVCE Post, Mysore Rd
Bangalore – 560069, India
+91-80-67064000
vinodsn@mindtree.com

Srinivas Rao Bhagavatula
MindTree Ltd.
Global Village, RVCE Post, Mysore Rd
Bangalore – 560069, India
+91-80-67064000
srinivasrb@mindtree.com

ABSTRACT
Business processes typically contain multiple process steps. In a service oriented landscape, these process steps are realized as services. An implementation of a business process is hence composed of multiple service invocations. In a service oriented landscape, a process model doesn't exist in isolation; it is supplemented by other paradigms which allow the process model to be executed. This paper presents all of these as patterns, and describes how these can be tied together to create a dynamic service oriented landscape. The patterns that this paper describes are:

- Service orchestration – the modeling of a business process as a set of process steps
- Service registry – a mapping of process steps to service endpoints
- Service monitor – a mechanism to monitor the health of an endpoint

Categories and Subject Descriptors
D.2.11 [Software Engineering]: Software Architectures – *Patterns*

General Terms
Design

Keywords
SOA, Pattern, Architecture

1. INTRODUCTION
This paper describes three architecture patterns for use in building a service oriented architecture (SOA) solutions. It then describes a fourth pattern that composes these three patterns.

2. INTENDED AUDIENCE
The first three patterns are targeted at people who are starting out on service oriented architecture. The fourth – Freeway – is targeted at solution architects who are looking to design service oriented landscapes.

3. PATTERN 1 – SERVICE ORCHESTRATION

3.1 Context
Consider a business process that consists of multiple steps to achieve its functionality. In an SOA environment, each of these steps could be realized as services. While it is possible to string together these services in code to implement this entire process, this makes it hard to quickly change the process.

3.2 Problem
In a service oriented landscape, how can a business process be implemented as a composition of multiple services?

3.3 Example
The creation of a customer savings account in a bank requires the following steps:

- Gathering customer information
- Verifying customer information
- Checking for duplicates in the bank's systems
- Carrying out a background check
- Creating the account in the core banking system

Each of these is implemented as a service. The services that perform these steps are:

1. Add Customer Information Service – invoked by a front-end application to enter customer information as present in the KYC (Know your Customer) form, into the system
2. Check Customer Information Service – used to enter validity of customer information as gathered by field agents, into the system
3. Check Duplicate Service – used to check if this customer exists in the bank's loan account system
4. Link Account Service – used to link the customer's loan account to this savings account
5. Blacklist Check Service – used to query blacklist lists
6. Create Customer Account – used to create the customer account in the core banking system

A flowchart for this is:

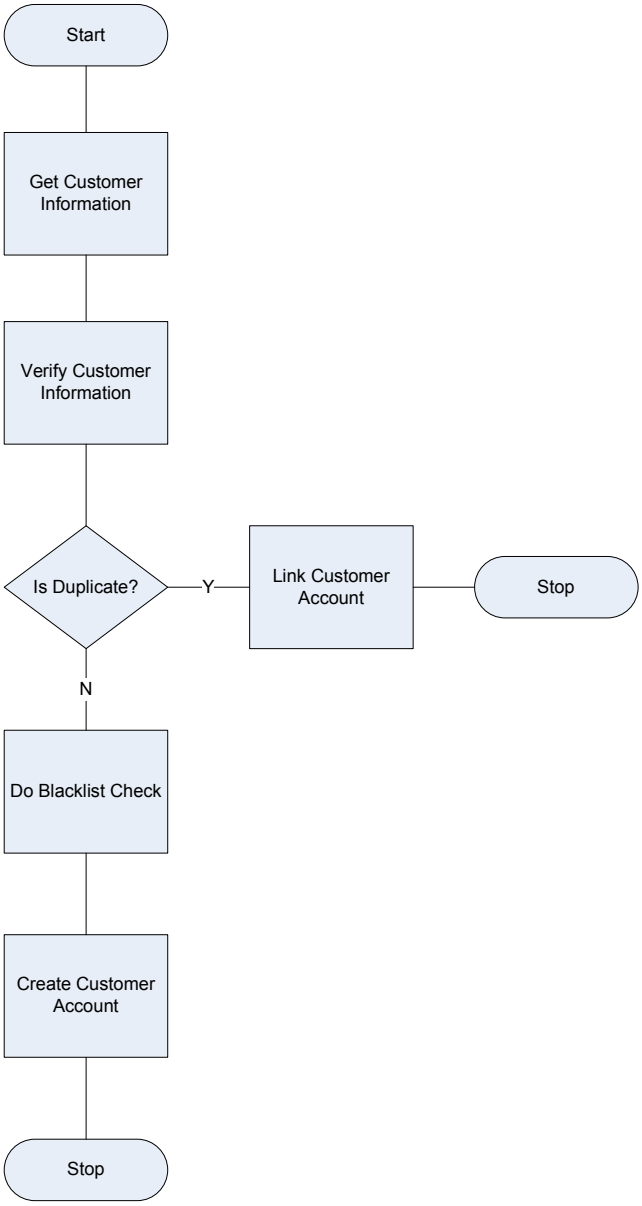

Figure 1. Example flowchart for customer account creation

While this is the current process, it should be possible to modify this process – for instance, the process could be changed to also check for political exposure.

3.4 Forces

In solving this problem, various forces need to be balanced. Some of these are:

1. Business processes consist of multiple steps, and are liable to change with steps being changed, added, or removed.
2. The steps can be executed either in a serial or a parallel fashion, or a combination.
3. Business processes can also contain constructs like loops, decision points, joins, etc.
4. There may be multiple service implementations that can service a process step; the model may not have adequate information to decide on the actual service endpoint during its design.

3.5 Solution

Represent the business process as a composition of process steps using a well-defined schema. The schema is a representation of two types of elements – paths, and tasks. Paths are a serial combination of tasks. Tasks are of multiple types. Some examples are:

- Service tasks that invoke a service endpoint
- Decision tasks that choose between two paths
- Parallel tasks that spawn off a new parallel path of execution
- Synchronization tasks that join multiple paths

The tasks are not tied to a particular service endpoint. Rather, they are identifiers of services that can be looked up via the Service Registry. This distinction allows the business process description to be independent of the implementation, and allows for choosing different endpoints based on QoS or SLA requirements.

This representation is termed Service Orchestration.

3.6 Structure

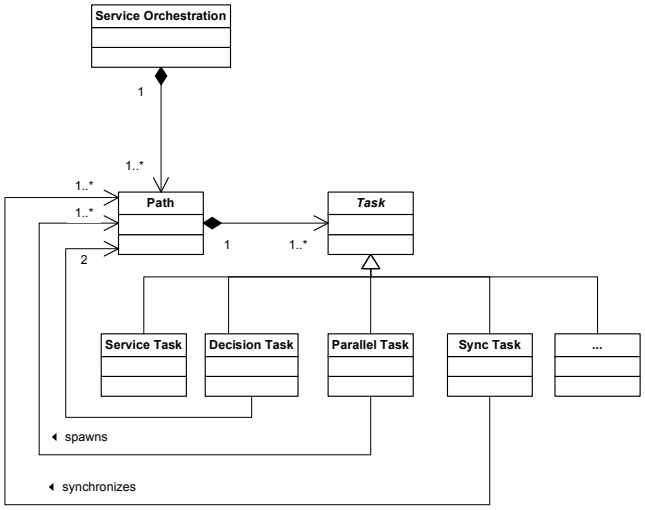

Figure 2. Service Orientation Structure

As described above, a service orchestration is a collection of paths, each of which contain a number of tasks. All tasks derive from an abstract task, which contains common functionality across tasks. Tasks can themselves result in additional paths being created (for instance, the decision task, or the parallel tasks), and can coalesce multiple paths into one (for instance, the sync task).

3.7 Example Resolved

The example in the Example section can be represented as an orchestration in XML format; a sample representation is shown here:

```
<?xml version="1.0" encoding="utf-8"?>
<process name="CreateCustomer">
  <path id="start">
    <task id="GetInfo" type="service">
      <service id="AddCustomerInfo"/>
    </task>
    <task id="CheckInfo" type="service">
      <service id="CheckCustomerInfo"/>
    </task>
    <task id="CheckDuplicate" type="decision"
        decisionservice="CheckDuplicate">
      <path id="IsDuplicate" type="TruePath">
        <task id="LinkAccount" type="service">
          <service id="LinkLoanAccount"/>
        </task>
      </path>
      <path id="NotDuplicate" type="FalsePath">
        <task id="CheckBlacklist"
type="service">
          <service id="CheckBlacklist"/>
        </task>
        <task id="CreateCustomer"
type="service">
          <service id="CreateCustomerAccount"/>
        </task>
      </path>
    </task>
  </path>
</process>
```

Figure 3. Sample XML representation for customer creation process

Each of the service IDs here represent an identifier that allows the actual service implementation to be looked up from the Service Registry. Through this model, the process definition can be modified by modifying the XML representation. For example, to add a check for political exposure – assuming the service to do this is already implemented – a new task element is added between the blacklist check and customer creation tasks.

3.8 Consequences

Using a service orchestration has the following positive consequences:

1. The definition of the business process is moved from code to a separate representation, thus making this representation directly editable.
2. The model contains information about how steps are to be executed, and how they can be combined.
3. The model contains information about constructs like decisions, etc.
4. The tasks in the orchestration are not explicitly tied to service endpoints, thus making it easy to replace endpoints without changing the business process itself.

The service orchestration pattern has the following negative consequences:

1. The model needs to be interpreted by an "orchestration engine" which adds complexity.
2. The model itself does not directly address issues like exception handling; this needs to be sufficiently specified in the orchestration engine.

3.9 Implementation

Implementing service orchestration has the following steps:

3.9.1 Model Specification and Storage

The orchestration model can be specified as XML, and stored on the file system. The schema can then be specified using an XSD. The orchestration model may also be loaded and cached in memory to improve performance.

3.9.2 Model Interpretation

The orchestration model can then be parsed by a parser, and the different types of elements and their sequence can be interpreted. The parsed model can also be kept cached in memory, to avoid repeated caching. The action taken for each of the elements depends on the type of the element:

- Service task elements need to have the indicated service invoked.
- Decision task elements need to have the deciding service invoked, and based on the outcome, execute either of two paths. Each of these paths can be composed of any number of tasks.
- Parallel task elements need to fork multiple paths of execution, and execute them on different threads of execution.
- Synchronization tasks wait for multiple target threads of execution to complete, before passing on control to subsequent steps.

3.9.3 Model Processing

In processing the mode, the following key aspects need to be kept in mind:

- Context flow: Since the entire model is about executing a business process, the different steps need to operate in the context of that business process. The context must hence flow across the steps of the model. The system executing the model must create an initial context, and pass that in through each of the tasks of the model. A task can modify the context with the result of its operation, thus enabling downstream tasks to modify their behavior if required. When the model requires creating new threads of execution, the context must be passed into each thread of execution. When parallel threads of execution are joined back in a synchronization task, the context must be merged back, and any conflicts must be resolved.
- Transactional Behavior: The model may need to be augmented to specify transactional behavior of the business process. Each task then needs to honor this transaction specification, and also pass on this context in any threads of execution that have been created. Services that implement a

standard like WS-Transaction [1] can participate directly in a transaction, while those that do not will need to implement compensating transactions, which can be called in the event of a rollback.

- Exception Handling: An exception that occurs in any of the tasks needs to be signaled. This can be done in multiple ways. The simplest is to set an exception flag in the context, and let the model executor take the appropriate action.

3.10 Known Uses

1. Business Process Execution Language (BPEL) [2] provides a way to represent business processes in a defined schema that allows the representation of the elements described above, for the service orchestration pattern. Web Services BPEL (WS-BPEL) [3] also allows representation of abstract processes, which are the representation of abstract tasks without endpoint information.
2. Sonic ESB [4] has the concept of an Itinerary [5] that is equivalent to the service orchestration.

4. PATTERN 2 – SERVICE REGISTRY

4.1 Context

Consider a service orchestration that provides a representation of a business process. If the orchestration definition is tied to specific service endpoints, then the successful execution of the orchestration depends on whether or not all the endpoints are operational. This context also applies to any client applications that are tied directly to a specific endpoint.

In another context, consider a process step in an orchestration that is implemented by multiple service endpoints, but which have different QoS or SLA specifications. In this case, there needs to be a way to store which endpoints implement which process steps, with which SLA, so as to be able to choose one appropriately.

4.2 Problem

In a service oriented landscape, how can an endpoint be interchanged for another, without affecting callers?

4.3 Example

When creating a bank account for a customer, one of the steps is that of checking a blacklist. Checking the blacklist is a vey intensive process, since it involves checking against multiple databases, and thus takes a lot of time. For high net-worth individuals (HNI), it is required that this check take place more quickly than others. For this purpose, the service is being planned to be hosted on two sets of servers – one more powerful than the other. The account creation program must not be hard-coded to use either service hosting.

4.4 Forces

The following forces need to be balanced here:

1. A caller is only aware of the service contract, and not the actual service implementation itself.
2. One process step can be implemented by multiple endpoints, each of which has a different SLA or QoS specification.
3. A service implementation's health may vary with time, load, and other factors.

4.5 Solution

Abstract the knowledge of actual service endpoints from callers by having a registry of service endpoints that include both regular services, as well as service orchestrations, identified by unique identifiers, and let callers look up the endpoint from the identifier. In the context of a service orchestration, the identifiers are representations of the process steps of the orchestration.

In its raw form, the service registry is just that – a registry of service endpoints – but in a service oriented landscape, where multiple service endpoints can implement the same functionality, it is also important to attach additional information along with the endpoint. Examples of such information could be the QoS level for this endpoint, the SLA that can be associated with this endpoint, etc. These will subsequently be used to discover services based on SLA requirements of callers.

In addition to the above additional information, the registry can also contain information related to the current state or health of the service endpoint. This information is updated by the service monitor. (See sec. 4 for more detail on the service monitor).

4.6 Structure

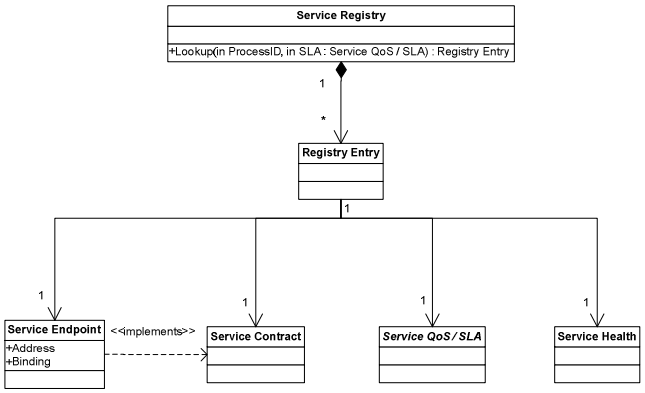

Figure 4. Service Registry Structure

The service registry contains a number of registry entries, each of which contains all information about the service endpoint. The information is that of the endpoint's location, (the address) the binding (how the service is to be invoked), the contract implemented by the service endpoint, the QoS or SLA supported by the service endpoint, and the current health of the endpoint. The service contract is not an instance of the contract, but the location of the contract description. In a web service scenario, this would be the location of the WSDL [6] for the service.

Since a given functionality (a process step in the context of service orchestration) can be implemented by multiple services, each process step / functionality is identified uniquely by a process ID. In the service registry, there can be multiple registry entries for one process ID, each corresponding to an implementation of this functionality at one endpoint.

The registry will contain multiple ways to lookup a registry (one is illustrated as a method in the class diagram as an example). These lookups can be based on complete or partial SLA requirements, in addition to the process identifier itself, which is essential. The process identifier is an identifier for the process step functionality that this service implements. The lookup operation will return one service registry endpoint that satisfies this SLA requirement, and that is considered healthy.

4.7 Example Resolved

In the bank account creation scenario, the service registry contains the following entries: (The contract, and health columns are omitted, and SLA is shown in a simplified fashion here.)

Table 1. Sample service registry entries for customer creation process

Process ID	Endpoint (Address, Binding)	SLA
AddCustomerInfo	http://banking.example.com/AddCustomer.asmx	Normal
CheckCustomerInfo	http://banking.example.com/CheckCustomer.asmx	Normal
CheckDuplicate	http://banking.example.com/CheckDuplicate.asmx	Normal
LinkAccount	http://banking.example.com/LinkLoanAccount.asmx	Normal
CheckBlacklist	*http://banking.example.com/CheckBlacklist.asmx*	*Normal*
CheckBlacklist	*http://hni.example.com/CheckCustomer.asmx*	*High*
CreateCustomer	http://banking.example.com/CheckCustomer.asmx	Normal

In this example, when a caller requires a blacklist check operation, instead of binding directly to a specific service endpoint, the caller requests the service registry to return a service endpoint based on the process ID (CheckBlacklist). The blacklist check operation requested for a HNI will have a requested SLA of "high", while that for a regular account has a requested SLA of "Normal". The registry can then return the correct blacklist check service endpoint. In case one of the endpoints is down, the registry can choose to return the other endpoint, thus extending availability.

4.8 Consequences

Using a service registry has the following positive consequences:

1. The caller is decoupled from the actual endpoint of the service, thus allowing for a switch to a different endpoint as required.
2. One service contract can be implemented at multiple service endpoints, thus resulting in flexibility of providing different SLAs for each implementation.
3. The logic of ascertaining an appropriate endpoint based on various parameters is moved from the caller to the registry. This makes implementing the caller simpler.
4. The registry acts as an authoritative directory of all services present in the landscape.

The service registry has the following negative consequences:

1. The caller needs to call into the registry at least once, to look up the endpoint, as opposed to directly being aware of the endpoint. There is hence a tradeoff between the overhead of this additional lookup and the benefit of the abstraction.

4.9 Implementation

Implementing the service registry has the following aspects:

4.9.1 Service Information Storage

The set of services can be stored in any durable store, such as a database. Since the number of service lookups may be high, the set of service can be kept cached in memory, and synchronized with the durable store. The cache can then be used for queries.

4.9.2 Service Lookup

Service lookups can be provided in multiple ways, based on a variety of parameters. The service registry can hence choose to expose multiple methods, each of which provides for different parameters being passed. A better alternative is to pass in a lookup context that encapsulates these parameters, and let the service registry return an endpoint based on the values of the parameters. The implementation of the lookup logic itself can follow the strategy pattern, so as to enable new implementations of the lookup logic at a later point.

Basic lookup implementations could simply be a round-robin across entries, while more sophisticated ones could involve selections based on advertised SLAs, requested SLAs, and actual current health.

4.9.3 Calling Clients

Calling clients can choose to lookup the registry for every call. On the other hand, to mitigate the overhead of repeated lookups, a client may choose to cache the returned registry entry, either for a period of time or till the entry becomes invalid. This would require additional error handling and retry logic on the client side.

4.10 Known Uses

Most enterprise service bus [7] implementations have a service registry. Examples of service registry are those in the Sonic ESB, and the WebSphere Service Registry [8].

5. PATTERN 3 – SERVICE MONITOR

5.1 Context

Consider a service oriented landscape where callers are abstracted from actual service endpoint information, by means of a service registry. In such a scenario, when a caller needs to actually use the service, the endpoint may or may not be available, or be able to deliver a particular performance level. The health of the endpoint is hence variable, and dependent on many factors. This can then affect the SLA or response time of the caller itself.

5.2 Problem

How can the health and performance of a service endpoint be monitored and published, so that callers are aware of this information prior to calling the service?

5.3 Example

One of the steps in the customer creation process is that of checking duplicate entries for a customer. If the duplicate-check service is down, the caller of this service would attempt to make a call to this service, and only if the service doesn't respond in time, would the caller choose alternative actions like returning an error, etc. In some cases, the caller may need to wait for a timeout in case the service is down. If the duplicate-check service were also

hosted on multiple endpoints, then the caller would need to make the same check on each service endpoint in turn, thus potentially taking a lot of time trying to make this call, which may not be acceptable.

5.4 Forces

The following forces need to be balanced here:

1. Having self-monitoring built into the service overloads the service, and isn't isolated from the service.
2. Services can be implemented on multiple technology stacks, monitoring on each of which could be different.
3. Querying endpoint information repeatedly can be an overhead both on the caller, and the entity supplying the health information.

5.5 Solution

Have external agents that monitor one service endpoint each. A central service monitor communicates with each agent, for consolidating results across all services. The service monitor then publishes these results either into its repository, or into the service registry. The agents record the current characteristics of health and status of the endpoint at multiple levels – at a physical node level, at a service level, and at specific functionality level. Monitoring at a specific functionality level requires the service to have additional functionality to do the health check – for instance, providing a heart beat function that can be called by the monitor.

The agent is specific to the operating system it is deployed on, but is generic across the technology stack of the endpoint itself, as long as standards, like web services, are used by the endpoint.

5.6 Structure

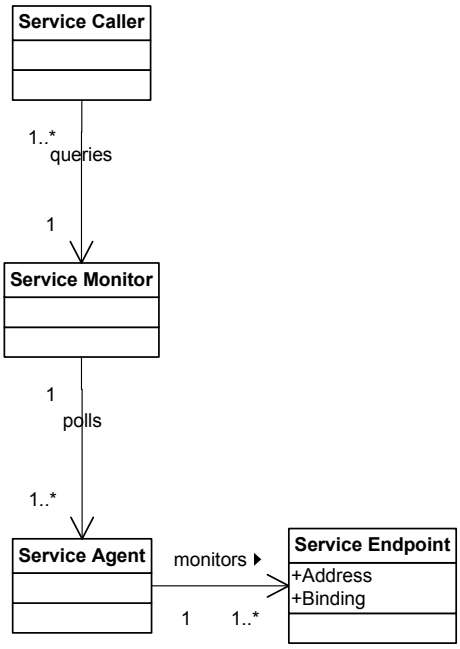

Figure 5. Service Monitor structure

This structure depicts the core participants, and doesn't depict additional participants that would result for the implementation of the subscription or polling models.

The service agent communicates with the service endpoint for checking its health. One service agent would typically monitor multiple endpoints on the same physical machine. The service monitor gets its data from the various service agents present. An alternative is to have the agent publish its results to the service monitor. The service caller queries the service monitor for the health for a given endpoint, before invoking it.

5.7 Example Resolved

In the above example, the agent monitoring the duplicate-check endpoint checks would have detected that the particular endpoint is down, and updates the status with the service monitor. The caller of this service then, before trying to invoke the service, checks its health first. The caller thus becomes aware of the service health without having to invoke it and wait. The caller can then take alternative actions, like returning an error, or using other endpoints.

5.8 Consequences

Using a service monitor has the following positive consequences:

1. Having an agent separate from the service allows the service to focus on its functionality, and not affect its own performance.
2. A per-OS agent allows multiple operating systems to be targeted. The agent interaction with the service itself is independent of technology in the scenario of a standard technology like web services.
3. The centralized service monitor allows endpoint health queries not to impact service performance.
4. A service monitor centralizes the mechanisms of collecting health information from disparate endpoints. It hence moves this concern out of callers, so callers are aware only of one monitor, as opposed to multiple mechanisms per endpoint.

The service monitor has the following negative consequences:

1. Depending on the type of notification mechanism used, there will be an overhead on the endpoint itself, due to the monitoring.

5.9 Implementation

Multiple mechanisms can be employed for keeping the health status up-to-date. These are listed below:

1. Polling-based: The service monitor polls each agent on a periodic basis; the agent then queries the health, and returns the health information to the monitor.
2. Subscription-based: The service agent is subscribed to events on the endpoint, and is notified of health changes. The monitor is subscribed to these agent events and is notified when the agent is notified of health changes. This mechanism is limited in that it doesn't automatically detect outages.
3. Polling + subscription based: Have agents collect health information via both subscriptions to events and polling, to detect outages. The monitor is subscribed to these agent events and is notified when the agent updates health information. This mechanism doesn't detect agent outages.

The monitor itself can then expose a mechanism such as publish-subscribe using the Observer [9] pattern, and a query mechanism

for callers to use, to be notified of health changes, or to query for current health status.

The service health can be stored directly in the service registry itself, thus allowing the service registry to directly return a service endpoint based on the additional facet of service health.

5.10 Known Uses

1. SNMP systems [10] use monitors and agents to monitor system health.

6. PUTTING IT ALL TOGETHER – THE FREEWAY PATTERN

6.1 Context

In a service oriented landscape, even with entities such as a service orchestration, a service registry, and a service monitor, the task of actually executing an end-to-end business process is still complex. To correctly ascertain which service endpoint to invoke, the system must perform the following steps:

1. A caller looks up a service orchestration from a process identifier.
2. The caller then parses the service orchestration, and picks up the individual process steps.
3. For each process step, the caller queries the service registry to match an endpoint to the service ID and the SLA required.
4. The caller then queries the service monitor for the current health of the endpoint.
5. The caller then invokes the endpoint, and passes control through the rest of the orchestration.

Executing the end-to-end business process in this fashion is similar to working out a physical path from one destination to another:

1. A commuter looks up a map to figure out how to get from one place to another.
2. The commuter then "parses" the map, and works out intermediate towns that will be encountered.
3. The commuter then works out choices of which type of road (freeways, by lanes, country roads, etc.) to use for each of the segments on the map, based on the requirements for speed, traffic conditions, etc.
4. When traveling, the commuter ensures that the road in question is actually still okay (for instance, by listening to traffic reports, looking for deviation signs, etc.), and if needed, re-evaluates alternatives (for instance, use a parallel country road instead.)
5. The commuter then continues through the rest of the map.

6.2 Problem

How does a caller execute a service orchestration without needing to bother about the underlying complexity of discovering and matching the most appropriate endpoints to individual process steps?

6.3 Example

The creation of a customer savings account in a bank requires the execution of a number of steps. The overall creation process has a different expected completion time based on the type of customer – whether the customer is a regular customer, or a high net worth individual (HNI). Each of the steps in the process, such as customer information entry, duplicate checks, background checks, etc. are implemented as services. These services are deployed in two sets of environments, so that the performance of creation of the HNI accounts is not impacted by that of the creation of regular customer accounts. The software which drives the account creation process is generic, and hence must choose the service endpoints based on the end to end process, the type of account, and which of the services is capable of performing better.

6.4 Forces

The following forces need to be balanced:

1. Getting to the set of endpoints takes a number of steps.
2. Each of the steps can be realized in multiple ways – SLA matching can be implemented via varied algorithms, orchestration parsing is dependent on the format of the orchestration, etc.
3. Callers should not be aware of too many entities (like the orchestration, registry, monitor, etc.) to be able to take advantage of a service oriented landscape.

6.5 Solution

Have an orchestration engine to which callers can supply what functionality is to be achieved, and what SLAs are required, and let the orchestration engine encapsulate the various steps needed to decide the actual endpoints to be invoked, and carry out the invocation of these endpoints as per the orchestration.

The orchestration engine ties together the three patterns described above, to be able to achieve this function. This is done via two additional components – the orchestration interpreter, and the service dispatcher – in the following sequence:

1. Caller invokes the orchestration engine to execute a process.
2. The orchestration engine looks up the service registry to pick up the service orchestration for this business process.
3. The orchestration engine invokes an orchestration interpreter to parse the orchestration, and to derive individual process steps and paths.
4. The orchestration engine then evaluates and executes each of the process steps. For each process step that involves invoking a service, the orchestration engine passes control to a service dispatcher component.
5. The service dispatcher uses the service registry to look up the endpoint, and evaluates the current health of the endpoint using the service monitor. If required, the dispatcher can look up additional endpoints as well.
6. The service dispatcher invokes the endpoint, and returns the results back to the orchestration engine.
7. The orchestration engine then goes through the rest of the orchestration.

The pattern of how the orchestration engine interacts with and ties together the service orchestration, the service registry, and the service monitor, using the other components of the service dispatcher and the orchestration interpreter is termed the Freeway Pattern.

6.6 Structure

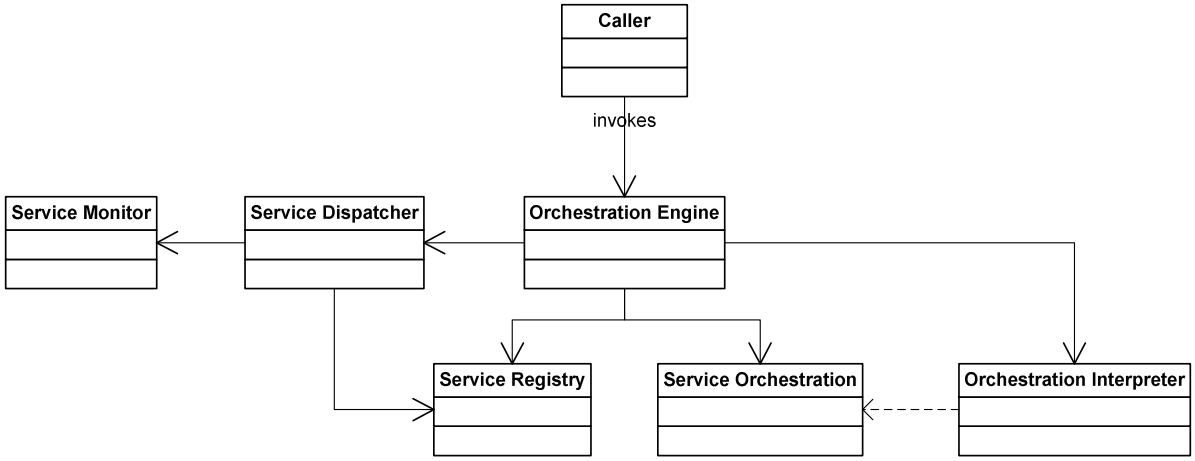

Figure 6. Freeway Pattern structure

The roles in this structure are:

1. The **Caller**: The caller interacts with the orchestration engine to execute a business process.
2. The **Orchestration Engine**: The orchestration engine is aware of the following entities:
   - The service registry, to look up the orchestration.
   - The orchestration interpreter to interpret the orchestration returned by the registry.
   - The service dispatcher to which it passes the steps returned by the orchestration interpreter.
3. The **Orchestration Interpreter**: The orchestration interpreter is essentially a parser that is aware of the schema of the orchestration. The orchestration interpreter parses the orchestration, and returns the set of paths that make up the orchestration.
4. The **Service Dispatcher**: The service dispatcher is responsible for invoking the service specified in the task returned by the orchestration interpreter. In this, it interacts with the service registry to look up the service endpoint for the task, and for the endpoint, it fetches the health of the endpoint through the service monitor, based on which it invokes the endpoint. The result of the invocation is passed back to the orchestration engine, which then goes through the rest of the steps specified in the orchestration.

6.7 Example Resolved

The account creation now simply invokes the orchestration engine, and asks it to execute the customer account creation process. The orchestration engine then goes through the steps detailed above, to invoke the individual services. During the lookup of the service from the service registry, it also passes along a priority (high, for HNI, normal, for regular accounts), based on which the service registry returns the most suitable endpoint. Before invoking the endpoint, the actual health is obtained via the service monitor. If the intended endpoint is busy or down, then an alternative endpoint is invoked, thus ensuring that the overall process is completed in an optimal manner.

6.8 Consequences

Using the freeway pattern has the following consequences:

1. The caller only follows a one-step process of invoking the orchestration engine.
2. The specifics of different operations are encapsulated in the service dispatcher and the orchestration interpreter, thus shielding the caller from the complexity of their implementation. The implementation can subsequently be varied if needed.
3. The caller is aware of only one entity – the orchestration engine.
4. Service orchestrations can be executed in an optimum fashion, taking into account both static factors (for instance, an endpoint's QoS) and dynamic factors (for instance, the health of an endpoint).
5. Additional services and service endpoints can be plugged in with little or no impact to the rest of the landscape.

6.9 Implementation

6.9.1 Orchestration Engine

The orchestration engine manages the state of the overall process. In addition, since each of the services may need to act within the context of the business process, the orchestration engine passes a context to each of the services through the service dispatcher. Services can update this context, which can then be used for subsequent decisions by the orchestration engine.

6.9.2 Orchestration Interpreter

The orchestration interpreter's implementation is dependent on the schema of the orchestration. The implementation can be done using the Builder pattern [11].

6.9.3 Service Dispatcher

The service dispatcher's implementation is straightforward, and involves a service lookup using the service registry, and a health lookup using the service monitor. The dispatcher passes along the context it obtains from the orchestration engine to the endpoint,

and returns the resulting context and the invocation results back to the orchestration engine.

6.10 Known Uses

1. Enterprise Service Bus [7] implementations like Sonic ESB [4] follow similar paradigms, using a service orchestration, a service registry, and a service dispatcher. The service dispatcher essentially sends a message to an "endpoint" which is an abstraction of the actual connection. The endpoint is then mapped to a "connection" which is the actual service endpoint that receives the invocation.
2. Microsoft's Connected Service Framework (CSF) [12] implements a way to specify a service combination, a registry of services, and a service monitoring mechanism to dynamically map, invoke and route between services.
3. MindTree's SOA framework, Momentum, provides similar functionality using a service registry and a service monitor, though it doesn't have the orchestration component in it.

7. RELATED PATTERNS

Table 2. Related Patterns

Item	Description
Observer	Is used in this pattern for monitors and brokers to be notified of service host status
Pipes and Filters [13]	Is a "straight-path" implementation of a service orchestration that provides a single line of control flow
Broker [14]	Pattern to hide the implementation details of identifying the service host by encapsulating them into a layer other than the business component itself
Factory Method [15]	Optionally could be used in this pattern to create the service proxy

8. ACKNOWLEDGMENTS

Bobby Woolf, for PLoP shepherding and for feedback during the IBM-Hillside Patterns Workshop.

Ademar Aguiar, Adriana Chis, Alexander M Ernst, Atsuto Kubo, Filipe F Correia, Hugo Ferreira, Joseph W. Yoder, Nuno Flores, Peter Sommerlad, Peter Swinburne, Ralph Johnson, Rebecca Wirfs-Brock, for the Writers' Workshop in PLoP 2008, Nashville.

Richard Gabriel, Rebecca Rikner, Kyle Brown, for the Writer's Workshop (IBM-Hillside Patterns Workshop).

Paul Adamczyk, for shepherding in preparation for VikingPLoP.

9. GLOSSARY

This section contains a brief description of some of the terms used in this document.

Table 3. Glossary of Terms

Item	Description
Address	In this context, the location of a service endpoint (for example, a URL for a web service)
Binding	In this context, the protocol by which a service is invoked (for example, SOAP over HTTP is the binding for web services)
BPEL	Business Process Execution Language – a way to specify a service orchestration n model
(Service) Caller	An invoker and consumer of a service's functionality
(Service) Contract	The specification of a service's interface
(Service) Endpoint	A concrete implementation of a service contract
ESB	Enterprise Service Bus – middleware that allows callers of services, and implementations of services to be connected via a hub-and-spoke model
(Service) Host	A process that hosts a service endpoint
QoS	Quality of Service; in this context, a measure of a service endpoint's performance capability
SLA	Service Level Agreement; in this context, a measure of a service endpoint's performance and availability guarantee
SNMP	Simple Network Management Protocol – a protocol to monitor hardware resources like servers
WSDL	Web Service Definition Language, a way to specify the service contract for web services
XML	Extensible Markup Language – a HTML-like tag-based representation of data.
XSD	XML Schema Definition – a way to describe the schema of an XML document

10. REFERENCES

[1] OASIS: Web Service Atomic Transaction – http://docs.oasis-open.org/ws-tx/wsat/2006/06

[2] Business Process Execution Language – http://en.wikipedia.org/wiki/Business_Process_Execution_Language

[3] OASIS: Web Services Business Process Execution Language – http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html

[4] Progress Software: Sonic ESB – http://www.sonicsoftware.com/products/sonic_esb/index.ssp

[5] Progress Software: Sonic ESB – ESB Architecture & Lifecycle Definition – http://www.sonicsoftware.com/products/sonic_esb/architecture_definition/index.ssp

[6] Web Services Description Language (WSDL) 1.1 – http://www.w3.org/TR/wsdl

[7] Enterprise Service Bus - http://en.wikipedia.org/wiki/Enterprise_Service_Bus

[8] IBM: WebSphere Service Registry and Repository - Software – http://www.ibm.com/software/integration/wsrr

[9] Observer pattern: Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides – Design Patterns: Elements of Reusable Object Oriented Software

[10] Simple Network Management Protocol – http://en.wikipedia.org/wiki/Simple_Network_Management_Protocol

[11] Builder pattern: Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides – Design Patterns: Elements of Reusable Object Oriented Software

[12] Microsoft Connected Services Framework – http://msdn.microsoft.com/en-us/library/aa303436.aspx

[13] Pipes and Filters pattern – Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. Pattern-Oriented Software Architecture: A System Of Patterns

[14] Broker pattern – Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. Pattern-Oriented Software Architecture: A System Of Patterns

[15] Factory Method pattern: Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides – Design Patterns: Elements of Reusable Object Oriented Software