Design Principles for Process-driven Architectures Using Oracle BPM and SOA Suite 12c

This book is a design handbook and provides skills to successfully design, implement, and optimize business processes on top of SOA. Starting with business process modeling, it shows design principles to architect sound process architectures. It presents best practices for modeling business processes using BPMN, together with design principles for services and composite applications. It provides detailed coverage of how to prepare business processes for execution. An in-depth explanation of human interactions is given and also principles and best practices for using rules.

Moving on, Adaptive Case Management principles are explained, along with the reach of business processes to mobile devices and ensuring multichannel interactions. Business activity monitoring, event-driven architectures, complex event processing in relation to business processes, and enabling integration with events and IoT devices are explained. The design principles and best practices are demonstrated in a practical way on a rental car use case.

Who this book is written for
This book is intended for BPM and SOA architects, analysts, developers, and project managers who are responsible for, or involved in, business process development, modeling, monitoring, or the implementation of composite, process-oriented applications. The principles are relevant for the design of on-premise and cloud solutions.

What you will learn from this book
- Design principles to model business processes and business architectures
- Best practices to produce executable business processes in BPMN
- Principles when designing reusable services and composite applications
- Advanced approaches to human interactions in business processes, including patterns and Adaptive Case Management
- Business rules management and principles for rule design and implementation, including using rules in BPMN and BPEL processes
- Prepare process applications for mobile and multichannel/omnichannel
- Explore the best practices and principles of Business Activity Monitoring to define and monitor Key Performance Indicators
- Extend the processes to Internet of Things devices and processing complex events

A design handbook to orchestrate and manage flexible process-driven systems with Oracle BPM and SOA Suite 12c.
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 4 'Process-driven Service Design'
- A synopsis of the book’s content
- More information on Design Principles for Process-driven Architectures Using Oracle BPM and SOA Suite 12c
About the Authors

Matjaz B. Juric holds a PhD in computer and information science. He is a full-time professor at the University of Ljubljana and heads the Cloud Computing and SOA Competence Centre (http://www.soa.si). Matjaz is an Oracle ACE Director and has been designated Java Champion and IBM Champion. He has more than 20 years of work experience.


He has published chapters in More Java Gems, Cambridge University Press, and in Technology Supporting Business Solutions, Nova Science Publishers, Inc. His work has also been published in several journals and magazines and presented at conferences.
Sven Bernhardt is a leading SOA/BPM architect and works as a solution architect for OPITZ CONSULTING Deutschland GmbH—a German Oracle Platinum Partner. In his role, he follows his passion for designing and building future-oriented, robust enterprise applications based on pioneering technologies. Sven is involved in diverse, large SOA and BPM implementations, dealing with challenges in the areas of business process automation and enterprise application integration. He also has longtime experience as an SOA/BPM coach, trainer, developer, and architect. As a leader of Competency Center for Oracle-based solutions, he develops and prepares implementation best practices and showcases and is responsible for knowledge building with respect to different Oracle technologies in the middleware area. Sven is an Oracle ACE and a frequent speaker at numerous IT conferences.

Hajo Normann works at Accenture in the role of an SOA and BPM’s community of practice lead in ASG. Hajo is responsible for the architecture and solution design of SOA/BPM projects, mostly acting as the interface between the business and IT sides. He enjoys tackling organizational and technical challenges and motivates solutions in customer workshops, conferences, and publications. Hajo, together with Torsten Winterberg, leads the DOAG SIG Middleware; is an Oracle ACE Director; and is an active member of the global network within Accenture. Hajo is also in regular contact with SOA/BPM architects from around the world.
Danilo Schmiedel follows his passion to deliver SOA and BPM solutions based on new technologies and trends. He is one of the leading BPM and SOA architects at OPITZ CONSULTING Deutschland GmbH – a German Oracle Platinum Partner. He is involved in large integrations, business process automations, and BPM/SOA development projects, where he has implemented well-accepted solutions for various customers. Danilo is an Oracle Director (ACE is short for Acknowledged Community Expert); frequent speaker at IT conferences; and author of numerous articles in various technical journals. Before joining OPITZ CONSULTING, he worked as a software engineer in several international projects. The Leipzig University of Applied Science awarded his outstanding work in 2009.
Guido Schmutz works for Trivadis, an Oracle Platinum Partner. He has more than 25 years of technology experience, including mainframes, integration, and SOA technologies in financial services, government, and logistics environments. At Trivadis, he is responsible for innovation in the areas of SOA, BPM, and application integration solutions and leads the Trivadis Architecture Board. He has longtime experience as a developer, coach, trainer, and architect in the areas of building complex Java EE and SOA-based solutions. Currently, he is focusing on the design and implementation of SOA and BPM projects using the Oracle SOA stack. A few other areas of interest for Guido are big data and fast data solutions and how to combine these emerging technologies into a modern information and software architecture. Guido is an Oracle ACE Director for Fusion Middleware and SOA and a regular speaker at international conferences, such as Oracle Open World, ODTUG, SOA & Cloud Symposium, UKOUG conference, and DOAG. He is also a coauthor of Oracle Service Bus 11g Development Cookbook, Do More with SOA Integration: Best of Packt, Service-Oriented Architecture: An Integration Blueprint, Spring 2.0 im Einsatz, Architecture Blueprints, and Integration Architecture Blueprint.
Mark Simpson, in addition to being an Oracle ACE Director, is Consultancy Director at Griffiths Waite, where he leads the UK-based Fusion Middleware development team with a focus on user experience, BPM, and SOA. He has worked with these technologies for over 10 years, having successfully delivered the first UK Oracle BPEL project back in 2004 and then the first BAM dashboard in 2005 for an innovative financial services organization. He is a frequent conference speaker, has received numerous awards, and actively supports the UKOUG and EMEA SOA Community. Recently, Mark was the lead architect on a design-focused ADF and WebCenter project. This project has reignited his passion for great UX and delivering increased customer insight through data visualization using SOA to ensure consistency in processes, services, and data.
Torsten Winterberg is active in several roles at OPITZ CONSULTING, all with a strong focus on delivering value to the customer. Being part of the business development and innovation department, he searches for and evaluates emerging trends and technologies to deliver innovative and differentiating solutions to customers. As a director of the competency center for integration and business process solutions, he follows his passion to build the best delivery unit for customer solutions in the areas of SOA and BPM. Torsten has longtime experience as a developer, coach, and architect in the area of building complex mission-critical Java EE applications. His competence and passion lies in the design and architecture of complex IT systems with regard to BPMN, BPEL, ESB, BAM, and service-oriented architecture in general. He is a known speaker in the German Java and Oracle communities and has written numerous articles on SOA/BPM-related topics. Torsten is part of the Oracle ACE Director team, is active in Enterprise BPM Alliance, and is responsible for the SOA/BPM topics in the DOAG development community.
The implementation and optimization of business processes have become high priorities for most enterprises. However, experience has shown that there is a huge gap between the implementation of a single business process and the definition of an enterprise-wide process-driven architecture that will be able to accommodate business processes over time, while ensuring the benefits of concepts and best practices, including reusability, loose coupling, flexibility, discoverability, and so on.

Experience has shown that only projects based on key design principles and best practices are successful. This book presents the key design principles for process architectures in a practical way via examples, using Oracle BPM and SOA Suite 12c.

This book is a design handbook. It provides you with the skills to successfully design, implement, and optimize business processes on top of SOA. Starting with business process modeling, it shows design principles to architect sound process architectures. It presents best practices to model business processes using BPMN, together with design principles to design and implement services (SOAP and REST) and extending through to composite applications.

It provides a detailed coverage of how to prepare business processes in BPMN for execution purposes on process servers and when to use BPEL. An in-depth explanation of human interactions is given, including different patterns, escalations, renewals, and other important concepts.

Business rules are covered, including principles and best practices to use rules in BPMN and BPEL processes. For scenarios where classical human interactions and rules are not sufficient, the book explains Adaptive Case Management.

Extending the reach of business processes to mobile devices and ensuring multichannel and omnichannel interactions are covered and explained.
Finally, business activity monitoring, event-driven architectures, and complex event processing in relation to business processes are explained. Here, it is not only shown how to monitor KPIs and do process analytics, but also how to enable integration with events and Internet of Things devices.

The design principles and best practices are demonstrated in a practical way on a rental car use case, called Rent Your Legacy Car, which is used for demonstration purposes through the chapters. Each topic is explained in detail and supported by examples that allow you not only to understand the principles and practices, but also to learn how to use and apply them in practice. Each chapter builds on the previous chapter, assuring a continuous flow and coherent reading experience. The reader will not only learn design principles and best practices, but also learn how to apply them through the Rent Your Legacy Car use case, which is used in all chapters. In the book, the latest Oracle BPM and SOA Suite 12c are used. The principles are, however, relevant for all process-driven architectures regardless of which implementation platform is used, on-premise or cloud based.

In this book, you will learn the following:

- Design principles for modeling business processes and business architectures
- Best practices to produce executable business processes in BPMN
- Principles for designing reusable services and composite applications
- Advanced approaches to human interactions in business processes, including patterns and advanced case management
- Business rule management and principles for rule design and implementation, including using rules in the BPMN and BPEL processes
- Preparing process application for mobile and multichannel/omnichannel
- Business activity monitoring and principles to define and monitor key performance indicators and using these for process optimization
- Extending the processes to Internet of Things devices and processing complex events
What this book covers

Chapter 1, Business Process Management, Service-oriented Architecture, and Enterprise Architecture, discusses the importance of business processes and their relevance to IT, application systems, enterprise architecture, reference models, and modeling principles. It describes the business architecture and enterprise architecture, describes their relation to business processes, and digs into business process management and its life cycle. It discusses the key concepts of process modeling, adaptive case management, and process execution, monitoring, and analytics. It explains process optimization. Finally, it explains how SOA and BPM fit together and discusses new frontiers for SOA.

Chapter 2, Modeling Business Processes for SOA – Methodology, describes strategies and methodologies that can help us realize the benefits of BPM as a successful enterprise modernization strategy. It discusses a set of actions in the course of a complete methodology in order to create the desired attractiveness toward broader application throughout the enterprise. It describes organizational and cultural barriers to apply enterprise BPM and discusses ways to overcome them.

Chapter 3, BPMN for Business Process Modeling, introduces the fundaments of business process modeling using a standard-based approach. It explains the concepts of using Business Process Model and Notation 2, a standard developed by the Object Management Group (OMG). BPMN 2 has gained the capability not only to model, but also to execute processes. From strategy modeling all the way to modeling executable business processes, this chapter discusses best practices to model business processes on different levels of decomposition.

Chapter 4, Process-driven Service Design, discusses best practices and patterns to design services. It explains the guidelines for service design in relation to BPM and demonstrates key service design principles. This chapter discusses service granularity, categories, data in the context of services, and how application services, cloud or on-premise, can be incorporated into your executable process.

Chapter 5, Composite Applications, explains key ways to architect and design the backend services that comprise the composite application. In this chapter, best practices for integration to backend systems, complex orchestrations, data validations, and calculations are explained. Also, some tools, such as templates, are used that help to create a successful solution that follows consistent architecture and design standards.
Chapter 6, *Process Execution with BPMN and BPEL*, outlines a set of guidelines to actually implement business processes and the associated services. This chapter describes how to define an implementation roadmap, explains how to implement the Service Facade and Delegation patterns in BPEL, discusses the right level of variances, highlights the degrees of coupling between technical components, and sets out a series of best practices, for example, naming conventions for composite partitions.

Chapter 7, *Human Interaction with Business Processes*, explains in depth what the integration of human actors means for process-driven applications, especially about the challenges from an architectural, conceptual, and technical perspective. Considerations with reference to UI design principles and user experience during development of user-centric processes and the corresponding inbox applications are discussed. The impact of the task-based approach on end users is also considered.

Chapter 8, *Business Rules*, outlines how a transparent enterprise decision management can be supported by IT by automating business decisions. It describes the characteristics of business rules and their relation with BPM; explains how to identify and assess business rules, that is, business rules candidates; discusses how to design, organize, and implement business rules; and depicts the Oracle Business Rules design time as well as the runtime architecture and its concepts. Finally, it discusses best practice approaches to implement business rules.

Chapter 9, *Adaptive Case Management*, gives all the overview information that is needed to fully understand the ideas behind ACM. It explains the characteristics of ACM, its relation to business analytics, shows how to model a case, how to build your own UI on top, and presents best practices.

Chapter 10, *Mobile and Multichannel*, explains the integration of mobile and BPM, provides best practices to design the architecture to enable multichannel access, accesses human tasks and BAM from mobile, and explains best practices for multichannel.

Chapter 11, *Event Processing and BPM*, explains what event processing is, why it is of interest to combine it with BPM, and how this can be achieved. It answers questions such as what is fast data and what is event processing, explains key elements of event processing and the different types of event processing, and compares event processing with Business Rule Management Systems (BRMS). It provides conceptual architecture for event processing and explains how event processing fits into a modern architecture.
Chapter 12, Business Activity Monitoring, shows how Business Activity Monitoring (BAM) can help an organization gain insight into business operations and outcomes in real time, thus enabling optimization, efficiency gain, and ongoing improvement to the business processes. This chapter defines the capabilities that a BAM platform should provide and the different types of insights that it delivers. It clarifies the difference between BAM and traditional Data Warehouse-based reporting solutions and describes how BAM integrates into SOA and BPM to monitor business processes and services in real time. It also presents a methodology and best practices to design and build BAM dashboards and the supporting data objects.
Service-oriented architecture (SOA) is a fundamental building block of a successful business process-driven architecture. To build agility and business alignment into our processes, we must ensure that the functionality that the processes implement is also designed in an agile, modular, and business-aligned fashion.

The IT architectures of many organizations have grown in an organic way through the adoption of new packaged applications (both commodity applications and industry-specific applications), custom-built applications in areas of differentiation, and hosted partner or cloud services. Integration of these systems becomes an increasing challenge with the divergence of the definition of business terms, technical protocols, and location of the systems. Whether these systems provide key engagement with customers, partners, and employees or are just a system of record, they are still required to interact with the organization's business processes. To give the business the control over the process that they require, the functions as well as the processes themselves can no longer be embedded within the systems. This becomes increasingly important as processes cross boundaries of ownership, for example, across organizational silos, partner organizations, and devices (mobile, tablet, desktop, and so on).
To ensure that business processes are not constrained by the IT architecture that supports them, a further layer of business functions must be provided for the business processes to interact with—one that speaks the business language, can interact in a consistent way with all the organizations or partner IT systems, and can provide information, actions, and functions across multiple devices. This is the role of the SOA layer. In this role, SOA is more than just a set of standards to allow systems to interoperate; it is a design approach that, if implemented correctly, will be a key enabler to a process-driven architecture. The design of the service layer must ensure that the change in IT systems has limited impact on the process layer and that business change can be facilitated by the IT systems with similar limited impact. One of the key drivers of BPM is to accelerate business change without too much IT involvement; this is only possible with the correct service architecture and approach in place.

If business process change requires constant IT refactoring of the services and functions that implement the process, then it will never truly deliver the business agility desired.

This chapter will outline an approach and a set of guidelines to design your services in such a way that they facilitate usage by the business process layer. We will do the following:

- Describe the role of a service in a process-driven architecture
- Explain the importance of service design in the context of business processes
- Set out a series of guidelines that should be adhered to in designing services
- Discuss the role of data in a service-oriented architecture
- Outline the impact of the lack of design of the service layer
- Discuss what should come first: the business process design, service design, or data design

**Service design guidelines**

As you roll out business processes to your organization, it will become increasingly important that a good set of service design guidelines are defined, communicated, and governed against. The definition of a service in your organization and the contract between the consumer and the provider will enable the separation of concerns between the two parties. It will allow the consumer, that is, the business process, to concentrate on delivering the outcomes they are responsible for without being concerned with the functions and activities over which they do not have control—which is now the responsibility of the service provider.
A business service is a contract between the provider and multiple consumers. It provides a guaranteed and repeatable behavior that allows the consumer to concentrate on the usage of the service rather than the details within the service.

Benefits of service design for BPM

BPM promises a fast rollout of processes that align closely with business needs. However, this cannot be achieved by process modeling alone. Implementing a good phase of service design within a BPM initiative will give the following benefits:

- **Response to business change**: A well-designed SOA will allow business processes to respond to changing business conditions in a fast, flexible manner, thanks to the ease with which services can be revised and reused. If application functionality is bound directly to the business process, then this can allow the first process to be implemented very quickly, but any change in that process will be constrained and delayed, and inconsistency will be introduced as new business requirements are responded to.

- **Adaptability in process execution**: With a good service architecture supporting the business processes, there will be more confidence in supporting variations in processes. Very often, changes to rules, parameters, or additional data being presented to the user may be implemented solely within the service layer, minimizing the impact of change of the process layer. In addition, if the service is designed for multiple usage scenarios, then changes in the process behavior could be implemented by just changing the inputs to the service layer. Customization of the process should result in minimal or no change at the service layer if the services are designed correctly.

- **Consistency in data**: SOA allows the enterprise to share data, information, and knowledge more readily through open standards and common protocols. SOA supports more effective communication both within an enterprise and between an organization and its supply chain since communications are not hampered by incompatible systems. This helps us to implement processes that cut across data domains and organizational boundaries.
Process-driven Service Design

- **Simplification of the process layer**: SOA simplifies the development, maintenance, and integration of enterprise applications through standard, reusable components. This building block approach means that an enterprise can add, remove, and swap out components without the pain that typically results from reprogramming large applications. Since applications can be reconfigured without rewriting the underlying code, changes can be made to any function by simply plugging in a new component. This allows new processes to leverage coarse-grained business functions as new systems are implemented, cloud applications are adopted, and partner providers are enrolled.

- **Secure business processes**: SOA supports security-enhanced environment and identity management. SOA allows administrators to define security policies for legacy, web, or cloud applications as well as for an entire organization. That way, employees gain access only to the data they're allowed to see, and any policy changes take effect almost immediately across the entire enterprise, preventing the business process from being concerned directly with the securing of functions or the trusting of identities.

- **Support and maintenance**: A good service design will reduce the overhead in supporting and maintaining business processes. It allows the business process to track the steps the business is performing and reporting on and does not dilute that layer with technical and functional details. With a more direct integration approach, it becomes difficult to trace the downstream impact of process steps, issues take longer to resolve, and they involve more teams, thus becoming a costly task.

**Key service design principles**

In Thomas Erl's book *Service-Oriented Architecture: Concepts, Technology, and Design*, he outlined a set of service-oriented principles that should be adhered to in building a service-oriented architecture. These principles give the service architect a good checklist to ensure that the services are working towards a solid service architecture that will satisfy the strategic goals of the enterprise. These principles are very relevant for service design that supports a process-driven architecture:

- **Service contracts**: A fundamental component of service-oriented architecture is a service's contract. Every service must have an accompanying contract. The service expresses its purpose and capabilities via this service contract. It is the contract that informs potential consumers of that service and about what they can expect from it (and hence forms the basis for enabling service reuse—a key goal of SOA). This includes both what functionality it implements and what nonfunctional service levels it will meet. The latter includes how secure it is, what response times and throughputs it can achieve, and what level of availability it will guarantee.
A service should be defined based on its external value and not its internal implementation. This is why services suit BPM so well—the business process can concentrate on the outcomes of the service rather than its implementation.

The service contract does not just define the interchange of data, but also its runtime behavior, activities, and outcomes. A service contract expresses how interface and data types are defined, and any policies and quality-of-service factors are attached to the service, but the contract also must communicate the outcomes expected when invoking these services. That is why a WSDL itself is often not enough to define the behavioral contract of a service—this is where a specification of the services and associated service tests come to the fore. Service tests are a crucial part of a well-defined service architecture and build up the trust that is required when reuse is promoted. A process analyst will be more inclined to leverage and reuse existing services if they can run simple service tests to validate that a service will give the outcomes required in the process they are defining.

The focus within the service design stage of any process implementation is to ensure that the service contracts are optimized at the correct level of granularity for the business processes and standardized to ensure services are consistent, reliable, and governable.

- **Loose coupling:** Coupling refers to a connection or relationship between two items. Within SOA, there will always be many interactions between services. When designing these interactions, it is essential that we make the linkage as generic and as loose as possible. This is to ensure that the services are shareable between different consumers or solutions, while also being easy to change without impacting any of the consumers of the service.

Where practical, we wrap the service with a virtual service (or proxy). There are two drivers for this. Firstly, we want to virtualize the endpoint—we don't want to hardcode into our service that we are calling service X on server Y. This would mean that every time X is moved to a different server, we would need to change the process. Similarly, if we replaced X with a new service, our service would need to be modified. It is imperative that the business process does not need to know about where the service implementation is running.

Secondly, we want to exchange messages in a generic format—we do not want our service to handle messages (data) in formats that have been defined outside our canonical or business data model. If we pass data to/from service X in X's own format and then we later upgrade or replace X, then our service will need to be changed—even if the change affects a data item that we have no interest in.

This technique will promote loose coupling between services. This is an essential principle of designing solutions for change.
• **Abstraction**: Through abstraction, we are able to protect the consumer from the details of the underlying service implementation. This is done by abstracting logic into more generic services that can be reused independently and mapped closer to business terms than IT terms. This is an important principle in ensuring that the services satisfy the requirements of the business process but are also designed to offer wider capabilities for the business and are not tied to a service implementation that is just specific to one given process step.

All business services should deliver a high level of abstraction and should not be dependent on a specific implementation. This provides maximum reuse by allowing access through multiple types of interfaces. It also provides greater versatility in how they are deployed. Business services should only contain essential information in terms that can be understood by all consumers without containing any implementation-specific detail or being tied to any particular underlying technology, delivery channel, or physical location. Changes to the business service implementation should have minimal impact on the service consumer, and there should be no dependency within the implementation on the identity of the consumer that invokes it.

• **Reusability**: Reusability can be seen as one of the key principles and drivers behind SOA. Service reusability focuses on ensuring that services can be reused across the organization without the need to recreate the business logic each time. If reusability is not considered during design, the reuse potential of services would be minimized. What is more worrying for a process-driven architecture is that inconsistency in the behavior of functionality will be introduced where similar services may diverge over time, providing unintentionally different outcomes.

There is an important handoff between the process designers and the service governance team through the design of services within a process-driven architecture. Each service requirement that a process introduces must be cross-referenced against the existence of an additional planned or implemented capability in the service catalog.

• **Composition**: SOA-based solutions are built by *wiring together* discrete services in such a way that data (or messages) flows between them in order to achieve the desired business objectives. From a solution designer's perspective, SOA is like designing for previous generations of technology—the problem domain needs to be divided into a hierarchy of *modules*. SOA differs from previous efforts due to its near-universal support; the modules you wire together can be implemented in a multitude of differing technologies—both modern and legacy.
The prevalence of SOA and standardized APIs also means that designers no longer reinvent everything from scratch. The first step now is always to try to find existing chunks of functionality that can be composed to meet new requirements. However, the state of the existing implementation will need to be verified to ensure that it meets quality criteria, and that it is in a technology that is not currently scheduled for retirement.

- **Autonomy:** The ability of a service to exercise control and governance over its execution environment is the key for it to provide reliable runtime performance. If a business process is reliant on a set of services that may be owned by different domains, it is essential that their execution is predictable. All services must have independence from outside influences with a clear functional boundary and little or no overlap between services. This is essential for their reuse in the BPM layer; otherwise, the business process would need to know about dependencies to facilitate the execution.

- **Discoverability:** A key consideration during design and governance of solutions is how to discover what assets currently exist. It is important to understand the need to identify where opportunities arise for potential reuse. This could be at a high level for the reuse of a business service across multiple business processes or at a lower level for reuse of individual artifacts within the service architecture. The business catalog within Oracle BPM will be populated as services are made available for use in the process layer. To ensure that an enterprise-wide view of the service architecture is given, the services created must be visible to other consumers. Then, services will not be built just to satisfy one business process step but will be able to be discovered and reused in the design of other business processes.

All services used externally to a given SOA domain are published and consumed via a service catalog, such as Oracle API Catalog. The SOA governance board will ensure that new candidate services adhere to the service governance life cycle, which will be implemented in Oracle Enterprise Repository.

- **Statelessness:** The management of excessive state information can compromise the availability of a service and undermine its scalability potential. Services are, therefore, ideally designed to remain stateful only when required.

The business process should store any state required to link steps in the process together but no more. Business services should be designed to be short running, where possible, and repeatable. Any long waits should be controlled by the process layer to allow the provision of human interruption, escalations, and reporting.
• **Encapsulation:** Each module that we need to set up in our hierarchy must be treated like a black box. We do not want to know what goes on inside it—just what effect it achieves. In other words, each service must provide a clearly defined set of operations passing clearly defined data in and out, with all external dependencies (for example, database tables accessed, flat files read/written, and other services invoked) noted.

It is imperative that we document the capability of a service in a standalone manner for every service—without this, reuse of these services in future business processes will be severely hampered, and changes to these original business processes will be lengthened as the process designers try to understand the actual implementation of the service to verify what capability it serves. Service unit tests should be designed to demonstrate the capability of the service, focusing on what the service delivers rather than how, to allow full encapsulation of the functionality.

**Service granularity**

Services can be either coarse-grained or fine-grained. A coarse-grained service performs a big task. To achieve the same outcome, the alternative would be multiple fine-grained services (with a thin layer of logic to link these invocations together). The first factor that drives the choice of granularity is why that service is being made available. If the service is being made available to remote consumers (trading partners or one of your international data centers), then a coarser granularity will reduce the time spent on network latency and hence improve overall performance. If the service is for local consumption, then finer-grained services are more preferable. By splitting a service into smaller subservices, you increase the likelihood of services being reused. Also, you reduce coupling (the consumer doesn’t have to deal with parameters that aren’t needed in the one bit of the service it wants to use). This can also improve performance since the consumer may already have some of the data that the service needs; thus, it can skip the service that would have reread that data and go straight to the service that uses it.

It is, of course, perfectly acceptable to expose both the inner workings as subservices and then provide one big service that remote consumers can avail of to execute the whole—getting the best of both worlds. There is always a tradeoff within a service architecture between reusability of fine-grained services and the management overhead added, so the right balance is needed. But within a process-driven architecture, the granularity that is exposed to the business process should be defined by the granularity of the business task at hand. The process should not need to call more than one service for the one business outcome that it requires.
Service categories

As your service architecture evolves and matures, the designers will build up a catalog of services, with each fulfilling a different purpose and each having different life cycles. Some will be shared externally with partners, some used directly within the enterprise business processes, and some intended to be used internally within one project only. As services get built and published beyond one team, a design hierarchy of service layers is required, and it is important that each service is categorized within that hierarchy. A layered architecture helps to assist with meeting the key SOA goals, such as loose coupling, improved reusability, abstraction, and improved agility. The layering is a design pattern that will lead to successful SOA implementations, especially with the introduction of BPM. However, it should not be mandatory to use all layers in all cases. Breaking a service down into smaller components adds flexibility if done correctly but also adds an overhead (at design, development, and runtime), and so must be challenged on each occasion to ensure that each component adds the right value. If a layer is in place that is serving no purpose, then it should be removed.

Implementing a reference service architecture and classifying the services into categories, each with differing roles and rules surrounding their usage, will ensure consistency throughout your design, development, and operations of the SOA environment. It will also allow multiple teams to contribute to the service architecture, allowing teams who know and understand the existing applications to provide services to teams who know and understand the business processes or user interface layer.

Without a governed reference service architecture and classification method, your architecture will deviate away from the structure that allows the benefits of SOA to be realized. An entangled, unstructured set of services would ensue, resulting in an architecture where the impact of change is wide and reuse is minimal. The maintenance of such an architecture will become more difficult than when using point-to-point integrations, and the business process layer will become tightly coupled to a chain of services, losing the clarity and simplicity that it would have if it were built on a well-defined and governed SOA.

Never mix application logic with business logic in a service; they change at different rates with different stakeholders. Keeping their life cycles separate minimizes the impact of change and increases reuse and agility.
Different organizations will choose different ways to categorize their services. If an organization is heavily reliant on packaged applications, such as Oracle Cloud Applications or Oracle Fusion Applications, for their core master entities, such as customers, products, and invoices, this may lead to the adoption of a service architecture model and classification closely aligned with that. It is the application integration architecture (AIA), in Oracle’s case, which is built on top of the trading community architecture (TCA) business data model.

An organization that works predominately to offer out services to partners and wants to closely integrate its business processes with its partners will focus its service categorization on public-facing services and design for multiple protocols with a data definition that can be easily understood by their partners (in an industry-specific vocabulary).

In this book, we will present a service classification model that will allow services to have a well-defined business entry point that allows a catalog of services to be produced and consumed by the BPMN business process. Within these classifications, there will be other metadata that will define the further characteristics of a service, such as whether the service externalizes business logic as rules to allow easy change. This includes whether the service is data-centric or function-centric so that the consumer can understand the downstream impact of invoking the service.

The following reference service architecture model shows one way to categorize your service to ensure there is clarity and governance as to which services should be used in the business process and by its supporting user interface. This model takes into account the requirements of various service consumers. This allows services to be exposed in different protocols with different levels of security as boundaries to consumer change and with variants of data being shared (perhaps based on consumers privileges). The model also presents a series of cross-cutting concerns that must be applied across all service layers to enforce consistency and ensure guidelines are followed in the design and creation of services.
Presentation services

This service layer is responsible for ensuring that the user of the service can consume it in their own preferred way from the operations, data, and protocol perspectives. For instance, a presentation service can publish a REST interface for usage by a partner who wants to visualize data or present functionality via a mobile device. It can present an event interface for consumers who want to decouple their application from the producer's services. It may present a well-secured SOAP interface for a consumer who requires to access sensitive data or functionality. Presentation services should not contain any business logic themselves; they should just facilitate access to business process services or business services to different consumers.

Presentation services enable specific service functionality to be grouped together to provide ease and efficiency for consumers. The purpose could be to minimize calls from a portal's browser to the middleware, to minimize the volume of data returned when providing an interface for a mobile device, or to provide specific naming, structuring, and protocol translation to ease the integration of your service into a partner's environment. It is important that these services are named for public consumers to understand and that the message structure is simplified from the canonical form. For example, the complex types referenced in a set of public-facing presentation services should be flattened into one schema definition file, the consumer should not be responsible for checking whether multiple schema definitions are imported into their projects, and it should be made as simple as possible to consume the presentation service.
For use across domains within an organization (for example, within a business process layer or other departmental systems), these services will be implemented on a service bus, which gives specific protocol support, ease of data translation, caching benefits, and provision of additional security. For external consumers (for example, partner APIs or mobile interfaces), Oracle API Gateway will be used to satisfy the nonfunctional security and reliability requirements associated with public-facing services, supporting system-to-system, web UI, and mobile interaction protocols.

**Business process services**

A business process service consists of the functional elements of the business processes, with no technical or application-specific knowledge residing at this layer. It will be based on the process models that the process designers have produced written in BPMN. These services should be written in the language of the business. The steps in the business process should mirror how the business sees that process, wherever possible, with the responsibility for the technical details of how that is made possible passed to the business service layer. The business process service should also manage the interaction between automated steps and manual or off-system tasks. The business process service should be aware of the input and output from the off-system tasks and any dependencies between automated steps and the off-system task.

An organization arranges business processes in a hierarchy to manage their complexity, so the business process service should be reusable itself where appropriate. A business process service is typically long-running with multiple participants (defined in swimlanes) and will generally include user interaction and workflow functionality to facilitate the inclusion of manual tasks within the business process.

In order to achieve the loose coupling and flexibility that BPM promises, it is critical when designing the business process service that you focus on what a process does and the steps required to reach the desired business outcome are, rather than how it does it. The *how* will be covered in the implementation of the process steps, which will be delivered via other services within the model. If the distinction between the business process services and the implementation of the functionality in the other services is kept strict, then the business process should only change when the business model changes or when process improvement is carried out. Likewise, any changes to the services that implement the business process should have no impact on the services until there are new features that the business want to leverage.
Enterprise business services

One of the primary aims of SOA, especially in the context of a process-driven architecture, is to abstract functionality away from the underlying IT systems and surface it in a layer more closely aligned with the business. Enterprise business services are a key component in delivering against that aim. They will act as the service façade to a more complex implementation of the functionality exposed. There can be different types of enterprise business services, but all should provide a set of business functionality to deliver business outcomes that can play a part in the business process architecture and communicate in a common business dialect. Enterprise business services will be shared on the service bus and made public across different SOA domains. The interface of an enterprise business service needs to be motivated by the customer's perception from a functional and data perspective—there should be no technical or implementation-specific details exposed. These services should always be designed contract-first, with the implementation added to meet the contract requirements. Therefore, if the implementation changes, such as routing to a new application provider for the service, the interface for the public consumers can remain stable.

These types of services will form the business catalog that a process analyst will utilize in building the BPM process and give a business abstraction of more granular services that deal with the logic and that produce the results that the process layer requires. Their responsibility is to route requests to the correct service that will give the functionality required within the process and will simplify the process analyst's interaction with SOA. They can be further classified into the following types:

- **Business activity services** provide business logic to control the steps to reach a business outcome. BPEL is often used to deliver these services as an orchestration of subservices. Business activity services are normally defined by a noun operating on a verb, such as `issueInvoice` or `checkCreditHistory`.

- **Business entity services** wrap entities from master data sources for a consistent view and maintenance of key business entities. These services should be defined against a business definition of the entity (customer, product, and so on), and it is important to distinguish whether the entity is within the context of a department, a business domain, the enterprise as a whole, or even wider than that, such as leveraging an industry standard canonical model and including partner concerns within the entity.
Process-driven Service Design

- **Business decision services** encapsulate frequently changing business rules that should be maintained by specialists within the business department, such as underwriting rules, price determination, customer classification, and so on. A business decision service should still have clear input and output that will form the facts for the decision, but the rules that determine the results should be stored and maintained separately from the service itself as part of Oracle Business Rules.

- **Human task services** present information to users to link off-system tasks or allow users to participate in the business process with knowledge-based decisions. Human tasks can be embedded in the business process if their usage is entirely for that one step of the business process. However, if there is scope for the tasks to be incorporated into other business processes, they should be created as reusable services. If more data in addition to that already used in the business process is required to complete the task, then the retrieval of this data should be included in the human task service.

Enterprise business services form the basis of Oracle BPM Business Catalog and so should be named with the business stakeholders in mind. If the name and description do not make sense to that audience, then maybe they are not true enterprise business services—a further level of business abstraction is needed for them to make sense in that context.

**Application services**

Application services play a crucial role in ensuring that the logic held within the business process and enterprise business services is not polluted by logic specific to an application. These services will change only as the applications they connect to change their interfaces or when the business services want to leverage more functionality from the services. These services are responsible for the translation of data between the application's data representation and the business schemas used by the process architecture. Application services will have the name of the application within their name to denote that they are tightly coupled with the applications interface. If the business functionality is migrated to another application, then the application services should be replaced too.

**Utility services**

Utility services are commodity infrastructure services that provide functionality that is not tied to any business entities, project-specific logic, or applications. These will include services for error reporting, auditing/logging, PDF generation, and so on. These should be standardized across business processes and will be used by multiple business services.
Service design – an enterprise concern

As a service architect, meeting the requirements of the initial design use case is only one of the concerns. It is essential that the wider SOA landscape is considered when producing the blueprints for a set of services within a project or process implementation. We could build up a micro service landscape with each service just meeting the need of the immediate requirement from a project. However, very soon, we will build up services that overlap functionality and that will lead to data quality issues, consistency problems, ownership debates, and ultimately a high cost of refactoring the architecture. If services are not layered correctly, it may seem simpler in the short term, but there will be a large management overhead for the services.

Security should also be the concern of the SOA framework. It is imperative that secure content can be included in the business process, but it should not be the process designer’s job to implement the security. Security policies are required that can be attached at the services layer, hiding the complexity of working with these policies from the process designer.

To reap the benefits of an SOA in supporting your BPM initiate, you will need to get your monitoring in place early and at the right level. There is IT operational monitoring that needs to be considered as well as business monitoring. IT monitoring will assist in keeping the BPM execution running smoothly, ensuring that dependent services are responding correctly. Get the SOA monitoring right and the business process monitoring can concentrate on the KPIs and leading indicators that will enhance business operations rather than monitoring IT variations and faults.

One other, very important enterprise concern is the enterprise data model that, from a BPM perspective, should be synonymous with the business terms that support your business architecture. Get this defined right in a collaboration between the BPM designers and the SOA architects, and the processes and services will be built on consistent foundations.

The more fine-grained the services are, the bigger the chance of overlap and increased management overhead. The coarser-grained they are, the more the impact of change. A balance is needed between the two, and service governance is required to maintain the balance that is right for your organization. This governance is of equal importance as you strive for a process-centric view of your service architecture.
Data in the context of SOA

The success of a service-oriented architecture is often in its logical data design. There are subtle differences between modeling a physical data store and a logical model to support the usage of data in services. Physical design concentrates on the system of record, optimizing the storage of data and access to that data in the most efficient way. On the other hand, services and processes deal with the usage and communication of data in the form of messages. This data should be structured in a way that the consumer will understand and include metadata about the message itself (such as the message sent time or source system) rather than about the physical storage of the data (such as last update details).

Physical data models focus on the storage of data to ensure consistent use across the enterprise, ensuring attributes are stored in one and only one place. Logical service data models focus on the usage of data, ensuring the meaning of the data is included in the service contract and that the consumer is abstracted away from being required to understand the physical model.

The logical data model for SOA is often referred to as a canonical model. The role of a canonical model is to give an abstraction from the physical data model and allow the consumer of services to concentrate on the business definitions of entities without being required to understand the details of data that is proprietary to a specific application.

In building a canonical model, a general model, such as the OASIS canonical model or a more industry-specific model can be utilized. If you have a large application footprint predominately from one vendor, then it is common to base the canonical model on theirs, such as Oracle's trading community architecture (TCA) model. Organizations can also build their own canonical model that is closely aligned with their business model—this facilitates the use of a common language when multiple business domains are crossed by a business process.
Objects will be used in service design to model the structure of messages that will be passed around the service infrastructure. They represent the organization of the messages that will facilitate the service-oriented architecture. These are hierarchical in nature and shouldn't be tied to a physical data model. Messages will be used to define the data passed between the services and processes – these messages will reference complex types that are grouped together in the object definitions. Both objects and messages will be defined in the form of XML schema documents (XSDs), so it important to distinguish between them with naming conventions. It is also normal to distinguish between an object/message that is tied to the data structures of a specific application and those that relate to the canonical or business model.

- **Enterprise business objects (EBOs)** give a business representation of an entity. These objects represent the business catalog of data that should be used referenced by business services and processes.
- **Application business objects (ABOs)** give the structure of an application entity that will only make sense within the context of that application.
- **Enterprise business messages (EBMs)** are instances of data that will be transmitted between services and processes. These will reference common elements defined in the EBO. These messages will form part of the service contract for public services and processes.
- **Application business messages (ABMs)** are application-specific data representations that will be mapped to the enterprise data structures via application services.

XSLT and XQuery will be the primary tools used to map messages from one service request to another. XSLT is very good at transforming one XML representation into another representation. It has techniques such as templates, allowing the reuse of logic across multiple nodes in the XML document. It is very suitable for data mapping of one structure to another and is very common in integration scenarios. XQuery’s strength lies in extracting sets of nodes, performing processing actions on those nodes, and building up the new document based on the results of these actions. Many of the concepts of XQuery will be familiar to an SQL developer, such as querying hierarchical and tabular datasets, restricting and manipulating these to form result sets, and so on. If your requirement suits the method of stepping through each node while mapping the data to a new structure, then XSLT is likely to be the simplest. If the mapping requirement requires more processing on specific parts of the document, then XQuery is likely to be more suitable. Within SOA Suite 12c, both tools are available in BPEL and the service bus, so a mixture can be used for their best purpose.
When designing the service interfaces for process-driven services, the language represented by messages is crucial. You must ensure that the messages are understandable to the layer that is consuming the service. Business services should ensure that the process designer has all the data required to populate the input message of a service and returns all the data required for subsequent steps.

Remember that all data items included in the interface message exposed by a business service in the business process catalog are to be populated by the BPM developer as a part of the composition of the business process, so make sure they are structured suitably and in a dialect that the process designer can understand with limited scope for variance in interpretation.

Service virtualization

SOA promotes loose coupling of components in an attempt to break the dependency and impact of change. Therefore, we should apply this principle of loose coupling within the service design of our process architecture. Services and components of services should not be bound directly to each other. All services should be virtualized to allow routing to the physical service to be changed at runtime. Any service endpoint changes should only result in a small configuration change for its consumer rather than any disruptive redeployment. All public services should be deployed to the service bus to facilitate the mediation of messages between the consumer and the provider. The service bus can also allow the same service to support diverse protocols that a consumer service may require. This virtualization should never incorporate any business logic but should have knowledge of the different versions of services that can satisfy the requests. It can also perform the role of caching or throttling data to protect more complex business service from undesired load.

For routing requests between services within a service domain or between components in a composite service, the mediator can be used. However, if the service is used by service domain boundaries or by different teams, then the service bus should be used. All services should be versioned to ensure backward compatibility of functionality, that audit trails of previous versions should be stored while the new service is transitioned, and consumers are allowed more scope to move to new services at their pace rather than the producer's pace. Services should be versioned with a major.minor.build structure:

- **Major** changes denote that the service contract has changed. These are disruptive and effort is required by the consumer to work with the new version of the service. Previous versions are normally kept operational in parallel until all consumers have moved across to the next major version.
• **Minor** changes denote that the functionality or contract structure has changed but that the interface and outcomes are backward compatible with the previous minor version of the service—allowing consumers to pick up the latest minor version without change. As a service consumer, it is important that automated tests are in place to verify that minor changes have not broken the usage scenarios that the consumer relies on the service for.

• **Build** versions are internal iterations of the services and may not be publically deployed and distributed.

Major changes will affect the service contract and require that the consumers rebind to the service to take advantage of the new version. It is crucial that the service deployed on the service bus supports multiple major versions of the service to allow consumers to use upgraded major versions at the stage in their delivery cycle that is appropriate for them. However, there is a tradeoff between being customer focused and offering wide choice and independence to the consumers versus the maintenance overhead that multiple service versions introduce (regression testing as well as the resources that each deployed service may consume). Therefore, a clear and published service decommissioning strategy is required, whereby the consumer knows how long superseded versions of a service will be supported and at what stage they would be required to upgrade.

Service decommissioning strategies often relate to the number of versions of the service deployed (for example, version n-2 always gets retired when version n of the service gets deployed). This is acceptable for services within a domain, but when a service is offered to another domain of ownership or external party, then more planning is needed for the consumer to upgrade to any new version of the service. In this case, the strategy should focus on supporting previous versions of a service for a period of time after a new major version is deployed, where the timeline is agreed with consumers upfront and included in the service contract. The period is intended to give enough time to allow the consumer to make the changes required bind to the new version and have confidence that the new version is stable. Typically, with this approach, public consumers are given 6–12 months before old major versions of services are no longer supported.

Ensuring that the correct version is invoked should be the role of the service bus. The service designer should decide whether service invocations should be routed to the latest major service (with the consumer not specifying any version), a specific minor version, or the highest minor version of a service for which the consumer specifies the major version. This can then be managed operationally at runtime with consumers given the flexibility of which version of the service they utilize.
Service design methodology
There are multiple approaches to defining the portfolio of services that will be delivered as part of an SOA. There is no one-size-fits-all for an approach to identify the right services needed to support current and future requirements. Service should be cataloged into service domains that will define the ownership, funding, usage rights, and categorization of the services.

Service domains are often driven by projects—to understand the ownership of services and how they can facilitate cross-organization business processes, these need to be realigned with business domains. However, business domains get restructured, so these domains should not be confused with an organizational model, which is normally a management structure to facilitate the people organization within departments of a company. SOA domains should be based on people, processes, and outcomes that are required to meet the organization's goals. With this in mind, the collection of services into domains should only change as the operating model of the organization changes and evolves.

Top-down portfolio-driven service design
This approach decomposes the business operating model and supporting business architecture into domains and identifies services to satisfy the functions within these domains. All services are defined upfront, with services implemented as they are required. A central authority is required to govern consistency across service domains, and any new request for services must be ratified from the top-down perspective to ensure it meets requirements set out in the operating model. This method of service design often produces services that map entities and functions to organizational structures, which should be avoided. It is important that the domains, functions, and entities map to the business model and that ownership is defined separately to the organizational silos.
Bottom-up application-driven service design

Often driven by integration projects or package implementations, this approach builds up a catalog of services from how the applications are designed, with the services composed together for usage in different contexts. It is very easy for this approach to produce a wild west SOA with a proliferation of services that make sense in one context, but do not support reuse and do not map to the business processes of the organization. It is acceptable to identify what services are available when a new application is implemented, but these services should be classified as application services and treated as such rather than forming part of the business service design.

Use case-driven service design

This approach arises when project requirements dictate the service design. Each use case in a project is considered to produce service requirements, and services are built directly to satisfy the project. In a mature service-oriented organization, this approach can work well, but normally the governance aspects of SOA, that is, ensuring that reuse ensues and that design is consistent across projects, are compromised for project deadlines. This can often result in a not built here attitude to the trust of services delivered by other projects and lead to projects repeating work and diluting the link between the service architecture and the business model and processes.

Process-driven service design

This is the preferred approach to build a service portfolio that supports a process-driven architecture, with the business process design dictating the requirements of the service architecture. This approach draws in features of the other design approaches, with services being cataloged into domains and checked against existing services as process requirements arise. They will also utilize application services via the service categorization model (abstracting the application integration away from the business process). Services will still be defined as use cases but within the context of the processes that require the functionality.
Service design should start at the abstract business process, identifying the capabilities required to satisfy each process step and the contract that should be agreed with the service provider to deliver that capability. This contract provides a demarcation between the process definition and the service implementation. There may already be a service defined in the service catalog that will satisfy, fully or partially, the capability required. There may also be an existing application that delivers a functionality that can satisfy the implementation requirements. The following diagram outlines the steps to follow when defining services to support process definitions:

1. **Design abstract business process**: No system terms should be included in this process service, each step should map to an activity to be completed, and the implementation of the step should be of no concern at this level.

2. **Identify abstract services**: Define the contract of the service operation first before any implementation details are considered. Operations, style (asynchronous/synchronous/one-way/event-initiated), business outcome required, data input and output (as business messages), nonfunctional requirements, and service domain/owner.
3. **Refine based on the SOA portfolio**: Group the service operations into service contracts that can be packaged together. Operations will be grouped if they operate on the same objects, perform related activities, or are a part of the same change life cycle. Review the service contract definition and requirements against the current and planned services. Identify change requests to already implemented services that have been initiated by other projects. Submit a request for the new service if there is no existing or proposed contract that satisfies the business process step requirements. If an existing service satisfies the requirement or comes close to the requirements, then initiate tests against the service and identify gaps for which a change request can be raised and the impact assessed. Finally, submit any new service contract or change it from an existing service contract to the catalog for review and reuse within other processes.

4. **Identify implementation details** to define the business logic, application integrations, and mappings required to satisfy the use case and add physical details to service.

5. **Apply service principles** to ensure the SOA reference architecture layering standards are adhered to and identify the composite service required. Classify these as being within the service categories. Break down the service functionality into these layers to ensure that the impact of change is minimized and reuse is maximized.

6. **Identify the use of application services** and map to the application portfolio, defining which application services should be grouped together. Wherever possible, application services should satisfy one outcome, and if multiple calls are required to the application to meet one outcome, these should be combined as a BPEL composite service. If APIs are available for applications, then keep extensions to these at a minimum within the application. If you are building the application service implementation in the source application, then ensure that the service principles are adhered to and that each implementation maps to a specific business outcome in the application, thus minimizing the number of calls into this application.
Applying service design to RYLC

The first task of a service architect within a process-driven architecture is to review the business process design to ensure that the design meets the principles laid out for process services. There should be no implementation details in the process. The process steps should be written in the business dialect, and there should be no splitting of the granularity of business steps due to the process designer’s assumption of what services are available. This review of the process model is useful in bringing together the process analyst and the service designer to focus on one common architectural goal. The initial review of the business process that will be executed and the set of business services that will support that execution should be a collaborative process.

Rationalizing the RYLC process into abstract services

In the RYLC example, the service architect walks through the process looking at each step and considering these as abstract services, questioning whether multiple steps in the process could actually be satisfied by one business step, whether a step in the process includes a reference to IT applications that shouldn't be assumed, and whether there are individual steps that are providing multiple business outcomes that you may not always want to be tied together as the process is implemented.

The service designer should pay particular attention for a common process anti-pattern, whereby consecutive steps that do not handoff the process flow to different roles or between manual/automated tasks and determine if there is a business reason for this split. For instance, within an initial RYLC process that the service architect was presented with, the first two steps in the BPMN model performed by the customer relationship manager do not have any true message flow between them and stay within one swimlane. As service architects, we must determine whether these should be implemented as one abstract service or two. The following diagram contains a model that shows an example of this:
The two steps for **Get customer information** and **Get customer status** are implemented for the same business outcome in this case—to determine the history of rentals from the customer, the classification of the customer from a reward perspective, which will be used when pricing offers, and some general details that will be used when communicating with the customer. The process designer may have split these two steps as they know that traditionally, different systems master the customer contact information, history, and rewards.

However, in the definition of services to support the business process, there should be no application knowledge involved, so granularity of the business service should just be determined based on the process requirements. So, you would only split these steps into two services if the business wanted to report on one step in isolation of the other, if the business wanted to reuse one of the steps multiple times in the process with different input and output, or if the business specified that one of these steps may be owned by a different role in the future.
None of these are applicable here, so the service would create the `CalculateCustomerStatus` abstract service operation that may be part of a composite `CustomerManagementEBS` service, which would take in a `CustomerIdentificationEBM` message and return the `CustomerDetails`, `CustomerHistory`, and `CustomerStatus` complex types—all adhering to the `CustomerEBO` model. This business service can then orchestrate the calls to the customer relationship management and the customer benefits system applications to retrieve the customer information and customer rewards status before returning control back to the process. The following diagram shows the layer of services to be implemented as part of the **Rent a Car (calculate customer status)** process step:
With this design, the multiple steps of accessing this information from different application systems is abstracted away from the business process to keep it easy to understand and simple to change. As application services change, the service contract for the business process should be kept stable. Only changes that result in additional functionality that the RYLC business process would like to leverage (such as a Twitter handle added to customer details) should result in a change to the business service contract.

This approach of contract-first development of business services is accelerated by the service bus that allows the creation of abstract services prior to concrete details being added as the composite service is built to support the functionality. This allows the creation of service catalog entries and tests for those services to be created in advance of the service implementations. The Rent A Car service has a key role as it will be published publically for use in a mobile and web user interface. Therefore, this is implemented with two service interfaces, SOAP and REST. This is achieved by creating a service bus pipeline that will expose the interface through two proxy services, RequestVehiclePS and RequestVehicleRestPS.

The SOAP interface will be represented by a shared WSDL and XSD. The following code shows this:

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<definitions targetNamespace="urn:RentACarBPS"
xmlns="http://schemas.xmlsoap.org/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/
xmlns:weo="http://www.rylc.org/core/carrental">
<types>
```
The XSD structure is kept as simple as possible for the outward facing consumption, with all complex types defined inline in the schema. This is a simplified version of the vehicle rental structure that will be used within the business process itself:

```xml
<?xml version="1.0" encoding="windows-1252" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://www.rylc.org/core/carrental"
    targetNamespace="http://www.rylc.org/core/carrental"
    elementFormDefault="qualified">
  <xsd:complexType name="tRequestVehicleInput">
    <xsd:sequence>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```
<xsd:element name="VehicleType" type="xsd:string"/>
<xsd:element name="CustomerName" type="xsd:string"/>
<xsd:element name="StartDate" type="xsd:date"/>
<xsd:element name="EndDate" type="xsd:date"/>
<xsd:element name="Location" type="xsd:string"/>
</xsd:sequence>
</xsd:complexType>
<xsd:element name="RequestVehicleInput" type="tRequestVehicleInput"/>
<xsd:element name="RequestVehicleOutput" type="tRequestVehicleOutput"/>
<xsd:complexType name="tRequestVehicleOutput">
<xsd:sequence>
<xsd:element name="requestResult" type="xsd:string"/>
</xsd:sequence>
</xsd:complexType>
<xsd:element name="RequestVehicleError" type="tRequestVehicleError"/>
<xsd:complexType name="tRequestVehicleError">
<xsd:sequence>
<xsd:element name="ErrorMessage" type="xsd:string"/>
</xsd:sequence>
</xsd:complexType>
</xsd:schema>

The REST version of the service still uses the same data structure and offers the same operation, but its specification is shared in the WADL format, as follows:

```xml
<?xml version='1.0' encoding='UTF-8'?>
<application xmlns:soa="http://www.oracle.com/soa/rest"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:weo="http://www.rylc.org/core/carrental"
xmlns="http://wadl.dev.java.net/2009/02">
<doc title="RequestVehicleRestSvce">RestService</doc>
<grammars>
<xsd:schema xmlns:tns="urn:RentACarBPS"
xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/
xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/
xmlns:weo="http://www.rylc.org/core/carrental">
<xsd:import schemaLocation="../XSDs/requestVehicle.xsd"
namespace="http://www.rylc.org/core/carrental"/>
</xsd:schema>
</grammars>
```
Process-driven Service Design

<resources>
  <resource path="/requestVehicle"/>
  <method name="POST" soa:wadlOperation="RequestVehicle">
    <request>
      <representation mediaType="application/xml"
        element="cns:RequestVehicleInput"
        xmlns:cns="http://www.rylc.org/core/carrental"/>
    </request>
    <response status="200">
      <representation mediaType="application/xml"
        element="cns:RequestVehicleOutput"
        xmlns:cns="http://www.rylc.org/core/carrental"/>
    </response>
  </method>
</resources>

The two service interfaces exposed rely on the same service implementation that will be implemented as a composite of other services from the service catalog.

Building the RYLC service catalog

RYLC wants to ensure that the services that are created to meet the business processes are aligned with their business model and have the correct ownership and change is managed by the business areas responsible for the functions. Therefore, they define a set of service domains with representatives from the business engaged as domain owners. At the start of their journey towards a service-oriented architecture, the involvement of the domain owners is low; they will review the structure of the initial services and ensure that the role of the services that their domain offers to the process architecture is complete.

As the architecture evolves, the domain owner's role will become a key part of business governance over SOA, managing change to the services, assisting reuse of the services, and reviewing the usage and SLAs of the services. They will work closely with the process owners to ensure the correct value is being obtained from the business service layer.
The following are the service domains and responsibilities of RYLC:

<table>
<thead>
<tr>
<th>Business segment</th>
<th>SOA domain</th>
<th>Responsibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td>Car rental</td>
<td>This is responsible for the customer-facing rental services, including all customer channels, which are the branch front office, Internet, mobile and call center. A customer experience representative is appointed to ensure that the processes and services that back the user interfaces lead to a joined-up experience for the customer.</td>
</tr>
<tr>
<td>Core</td>
<td>Product and marketing</td>
<td>This contains all the services that facilitate or result from campaigns, promotions, and prospect or customer segmentation and targeting. All decision services relating to pricing a rental package will be included in this domain. A representative from the marketing department has been selected to become an SOA stakeholder and review the marketing capabilities that are surfaced to the process layer and how these will interact with the marketing, car rental, and customer relationship management business processes.</td>
</tr>
<tr>
<td>Core</td>
<td>Customer management</td>
<td>Customer experience is paramount within this process and will be focused on all services that produce an outcome for the customer and not just the user interface elements. A representative from the customer account management team will ensure all CRM services meet the need to manage the customer experience effectively and will review the processes to ensure that the end-to-end customer experience is satisfied, suggesting new services or refinements where gaps are found.</td>
</tr>
<tr>
<td>Core</td>
<td>Fleet management</td>
<td>The fleet team is responsible for the logistics of ensuring that the vehicles are in the right place for each rental, adding to the stock, and monitoring the use of the vehicles for replacement. They will need to make sure they get the information they require on the return of the vehicle to keep the fleet management efficient and work towards the KPIs and goals set out for fleet management in the business architecture.</td>
</tr>
<tr>
<td>Core</td>
<td>Repair</td>
<td>This is responsible for the condition evaluation, repair, and valeting of the returned vehicles. It will be essential that a stakeholder from the repair team understands the effect of the outcomes and SLAs of the repair services on the new business processes such as the Rent A Car process.</td>
</tr>
</tbody>
</table>
Process-driven Service Design

<table>
<thead>
<tr>
<th>Business segment</th>
<th>SOA domain</th>
<th>Responsibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support</td>
<td>Handling and financials</td>
<td>The role of the finance domain owner of SOA is to ensure that the Rent A Car process utilizes the financial services in the correct way to expedite any invoice processing and to ensure accuracy and minimize faults in the financial processes. Very often, invoice errors will be down to data issues in the upstream process, so it is imperative that there is a finance input into the RYLC Rent A Car process and services.</td>
</tr>
</tbody>
</table>

We will package the business services together into the web services listed in the following table. However, we will leave the application services finer-grained as the rate of change of these services is dictated by the source legacy systems that they operate on, where JCA adapters or APIs from these source systems will be encapsulated in a service that will be invoked from the more coarse-grained business services.

Taking fleet management as one of the key business services and expanding the hierarchy of services within it, we can define the following service catalog. The business service contract should be base-lined early with the SOA developer and SOA testers, so it is imperative that a collaborative review of these definitions is done with the team. However, the more granular layers of the service will be expanded by the SOA developer, and you are likely to see extra services and components added purely for implementation reasons, such as error and compensation handlers, delegators and audit components. These services should not appear within the business catalog and, therefore, would be invisible to the process designer as they extend existing processes or make use of the business services in new processes.

The following table represents an extract from service catalog for fleet management:

<table>
<thead>
<tr>
<th>Catalog SOA domain</th>
<th>Service operation</th>
<th>Service category</th>
<th>Service description</th>
<th>WSDL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fleet management</td>
<td>requestVehicle</td>
<td>Human task service</td>
<td>This requests a rental on the entered vehicle and date criteria</td>
<td>RentACarBPS.wsdl</td>
</tr>
<tr>
<td>Fleet management</td>
<td>findAvailable VehicleTypes</td>
<td>Human task service</td>
<td>This returns a list of available vehicle models</td>
<td>FleetManagementEBS.wsdl</td>
</tr>
<tr>
<td>Fleet management</td>
<td>findAvailable VehiclesAndPrices</td>
<td>Business decision service</td>
<td>This retrieves a list of vehicles and calculates the rental price for each based on the terms of input given to the service</td>
<td>FleetManagementEBS.wsdl</td>
</tr>
</tbody>
</table>
### Chapter 4

<table>
<thead>
<tr>
<th>Catalog SOA domain</th>
<th>Service operation</th>
<th>Service category</th>
<th>Service description</th>
<th>WSDL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fleet management</td>
<td>bookVehicle</td>
<td>Business activity service</td>
<td>This registers a rental period for an existing customer</td>
<td>FleetManagement EBS.wsdl</td>
</tr>
<tr>
<td>Fleet management</td>
<td>undoBookVehicle</td>
<td>Business activity service</td>
<td>This cancels a rental period for an existing customer</td>
<td>FleetManagement EBS.wsdl</td>
</tr>
<tr>
<td>Fleet management</td>
<td>checkoutVehicle</td>
<td>Business activity service</td>
<td>This gives a notification that the vehicle has been handed out to the customer</td>
<td>FleetManagement EBS.wsdl</td>
</tr>
<tr>
<td>Fleet management</td>
<td>checkinVehicle</td>
<td>Business activity service</td>
<td>This gives a notification that the vehicle has been returned from the customer</td>
<td>FleetManagement EBS.wsdl</td>
</tr>
<tr>
<td>Fleet management</td>
<td>releaseVehicleForRental</td>
<td>Business activity service</td>
<td>This makes the vehicle available as part of the pool of rentals</td>
<td>FleetManagement EBS.wsdl</td>
</tr>
<tr>
<td>Fleet management</td>
<td>evaluateVehicleCondition</td>
<td>Business decision service</td>
<td>This is a rule-based service to evaluate the vehicle condition</td>
<td>FleetManagement EBS.wsdl</td>
</tr>
<tr>
<td>Fleet management</td>
<td>bookVehicleCBS</td>
<td>Application service</td>
<td>This books and unbooks the vehicle into the CBS booking system</td>
<td>Booking Information CbsReqABCs.wsdl</td>
</tr>
<tr>
<td>Fleet management</td>
<td>updateFleetData</td>
<td>Application service</td>
<td>This updates the fleet information in the CRS fleet management system</td>
<td>updateFleetData.wsdl</td>
</tr>
</tbody>
</table>

The service identification process would continue to observe each step in the abstract business process, allocate it to the correct SOA domain, and discuss the purpose of the service with the owner of that domain to make sure the service reflects the business capability required and not just the immediate use case instance of that service. This role is often fulfilled by the SOA architect as the organization's SOA matures, but even in these early stages, involvement should be sought from the relevant business stakeholders in the definition of the enterprise business services and their allocation to domains. Once the service architecture is defined and each business service has its operations, types, and messages defined, these can be exposed in the business catalog and added as implementation details to the process. The specification of these business services and the supporting application services can then be passed over to the SOA developers for implementing.
**Service architecture for the Rent A Car process**

The following diagram represents a service architecture map for the Rent A Car process with the process, business, and application services identified and allocated to their owning SOA domains. This diagram, along with a service specification, normally including a service definition, use case, and test scenarios, will be passed on to the SOA development team.
Summary

In this chapter, we introduced the importance of defining your services based on the business architecture as abstract services and adding the implementation details in a layer of services subsequently. The role of data in an SOA that will support business processes has been understood with a critical success factor being the definition of the business data model that, along with services, will form the connection between the process layer, user interface layer, and the service layer. We have understood how important it is to separate application logic from service logic and process logic to ensure that the benefits of a process-driven architecture are realized.

There are a number of key takeaways from this chapter, the first among them being to ensure that the design of the services is considered in the business context to ensure that the service layer does not become a one-to-one mapping of each process step. Then, public services should be designed with the customers in mind, ensuring that the service granularity, protocols, data structures, and naming approach are simple and intuitive to consume. Finally, a service versioning strategy should be defined and communicated with service consumers, with minor service changes not impacting the public service interface and major service changes adhering to a clear rollout and decommissioning strategy.

A service catalog that publishes a list of services with a defined capability in business terms is critical to the success of a consistent process-driven architecture.
Where to buy this book

You can buy Design Principles for Process-driven Architectures Using Oracle BPM and SOA Suite 12c from the Packt Publishing website.

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

Click here for ordering and shipping details.