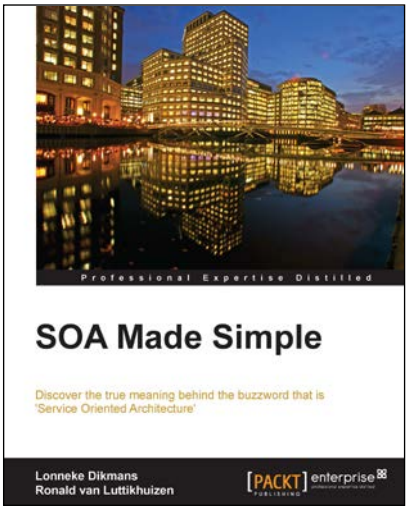
[PACKT] enterprise
professional expertise distilled
PUBLISHING

SOA Made Simple

Lonneke Dikmans
Ronald van Luttikhuizen



Chapter No. 4
"Classification of Services"

In this package, you will find:

A Biography of the authors of the book

A preview chapter from the book, Chapter NO.4 "Classification of Services"

A synopsis of the book’s content

Information on where to buy this book

About the Authors

Lonneke Dikmans lives in the Netherlands with her husband and two children. She graduated with a degree in cognitive science from the University of Nijmegen in the Netherlands. She started her career as a usability specialist but went back to school when she lived in California to pursue a more technical career. She started as a JEE developer on different platforms such as Oracle and IBM, and specialized in integration. She now works as an architect, both on projects and as an enterprise architect. She has experience in different industries such as financial services, government, and utilities. She advises companies that want to set up Service Oriented Architecture and Business Process Management. Lonneke was one of the first five technical experts to be recognized as an Oracle Fusion Middleware Regional Director in 2005. In 2007, the program was renamed and is now known as the Oracle ACE program. Lonneke is a BPMN certified professional and was awarded the title of Oracle Fusion Middleware developer of the year by Oracle Magazine in 2007.

Lonneke is the managing partner of Vennster with Ronald van Luttikhuizen. Vennster is a knowledge-driven organization. Vennster’s single most important ambition is to help her customers improve their products and services by improving the quality of the information flow. This is accomplished by offering services in the areas of User Experience, Business Process Management, and Service Oriented Architecture.

Lonneke has contributed to the *Oracle SOA Suite 11g Handbook, Oracle Press* by Lucas Jellema that was published in 2011. She publishes on a regular basis in magazines and on the internet, participates in podcasts, and speaks at international conferences about Service Oriented Architecture and Business Process Management.

I would like to thank the people that I have worked with over the years that helped shape my thoughts about Service Oriented Architecture. It would take too much space to list them all. Everyone contributed in different ways and were from different fields: technical people, enterprise architects, project managers, departmental managers, product managers, and so on. I would like to thank the reviewers Derkjan Zweers, Anant Kadiyala, and Howard Edidin for their valuable input. Their perspective, remarks, questions, and suggestions were very valuable. Last but not least I would like to thank my husband Hans and our children Mathijs and Anne for their support, encouragement and patience. My final thoughts are for our neighbor Dafnis, who died earlier this year at the age of 13. His courage and determination have become an example for me. We miss him!

Ronald van Luttikhuizen lives in Nijmegen, the Netherlands with his partner Susanne. He has over 10 years of experience in IT. Ronald studied Computer Science at the University of Utrecht and University of Wisconsin – Madison and received his MSc degree in 2003. Ronald creates valuable solutions for the business using a structured approach to Service Oriented Architecture. He takes into account both technical and functional aspects of a process to come up with a feasible solution. Ronald worked in projects for government, financials, energy, logistics, and services.

Ronald has experience in various roles such as architect, project lead, information analyst, software developer/designer, coach, trainer, team lead, and consultant in a wide variety of enterprise applications. He started his career as a specialist in analysis and design, application development, and application and process integration. The main technology focus in these projects were UML, Java, and XML. In later years, Ronald focused on architecture within service-oriented environments and other types of EAI environments, describing the to-be architecture, defining roadmaps, guiding implementation, and building parts of the solution.

Ronald is a speaker at (international) conferences and regularly publishes articles on Oracle Technology Network, his blog, Java Magazine, Optimize, and participates in OTN ArchBeat Podcasts. In 2008, Ronald was named Oracle ACE for SOA and middleware. Ronald was promoted to Oracle ACE Director in 2010. Ronald wrote several chapters for the *Oracle SOA Suite 11g Handbook, Oracle Press* by Lucas Jellema and served as a technical reviewer for the book. The book was published in 2011.

I would like to thank everyone that helped me in my professional career and my personal life. Without them I wouldn't be able to do the job I do today! A big thanks to my friends and family for supporting me and putting up with all the time I spent on the book and not with them; especially Susanne.

Last but certainly not least I would like to thank the reviewers Derkjan Zweers, Anant Kadiyala, and Howard Edidin and the people at Packt for their valuable input, suggestions, improvements, help, and patience! Without them this book wouldn't exist.

SOA Made Simple

A lot of organizations are implementing, or want to implement, Service Oriented Architecture to support their goals. Service Oriented Architecture is a natural step in the evolution of Information Technology; we started out with big systems in universities and banks, and moved to desktop computers in the workplace and at home. We are now moving to solutions in the cloud, offering services to consumers and businesses alike, adding mobile computing to the mix. So what is a service? A service is something that has value. Service orientation is not a difficult concept to grasp, everyone knows services and uses them daily; think of a hotel that offers a shuttle service to the nearest airport. Or the hairdresser that cuts your hair. This book describes how you can accomplish service orientation successfully in your organization and in IT, using a practical and simple approach. It is done without overly complex abstractions, but with examples from different industries and hands-on experience of the authors. The approach is independent of the specific technology or programming language you apply in your organization.

What This Book Covers

Chapter 1, Understanding the problem?, discusses the challenges that organizations face with respect to information technology and is illustrated with examples. Architecture is explained as a means to solve these problems structurally and in compliance with your organization's goals.

Chapter 2, The Solution, explains how applying SOA can help your organization to solve the problems that were discussed in the previous chapter. In this chapter, the concept of services is explained as well as Service Oriented Architecture.

Chapter 3, Service Identification and Design, describes how services are the base of a Service Oriented Architecture. The process of identifying services and designing their interface, contract, and implementation are important activities when realizing a Service Oriented Architecture.

Chapter 4, Classification of Services, covers the different types of services. You learn in this chapter how classification can help you in your SOA effort. The chapter explains different ways of classifying your services and the reason to choose a particular classification. Classification based on service composition is discussed in detail.

Chapter 5, The SOA Platform, identifies the different components of an SOA platform and explains the use of these components, keeping in mind that to realize an SOA in your organization, you need a platform to build it with

Chapter 6, Solution Architectures, tells us about how you can go for a best-of-breed solution to realize your SOA, or use a product suite. The solution of the big software

vendors Oracle, IBM, and Microsoft are discussed in terms of the components you need for an SOA platform.

Chapter 7, Creating a Roadmap, How to Spend Your Money and When?, explains how to plan your endeavor. In this chapter, creating a roadmap for the realization of your SOA is discussed.

Chapter 8, Life Cycle Management, explains how to maintain services. Requirements may change, services may become outdated, and new services may be needed. This chapter discusses life cycle management of services, and tooling that supports registries and repositories.

Chapter 9, Pick your Battles, talks about how during the realization and operation of an SOA you will run into issues with stakeholders. A common pitfall for architects is to be too strict and unrealistic about what can be achieved. This chapter discusses some common issues you will run into and discusses how to handle them.

Chapter 10, Methodologies and SOA, talks about how there are existing methodologies in IT that you are probably using right now in your organization for project management, demand management, and so on. This chapter discusses the impact of using SOA on these existing methodologies.

4
Classification of Services

In the previous chapter you learned how to identify and design services. This chapter starts by revisiting the classification of services that was explained in the previous chapter, the aspects on which it is based and how the classification can be applied. The chapter briefly discusses why it is important to classify services based on your needs and what aspects can be used to create a classification of services.

You will then learn how services can be combined into larger services, which is called **service composition**. Composition is one of the building blocks for the classification scheme.

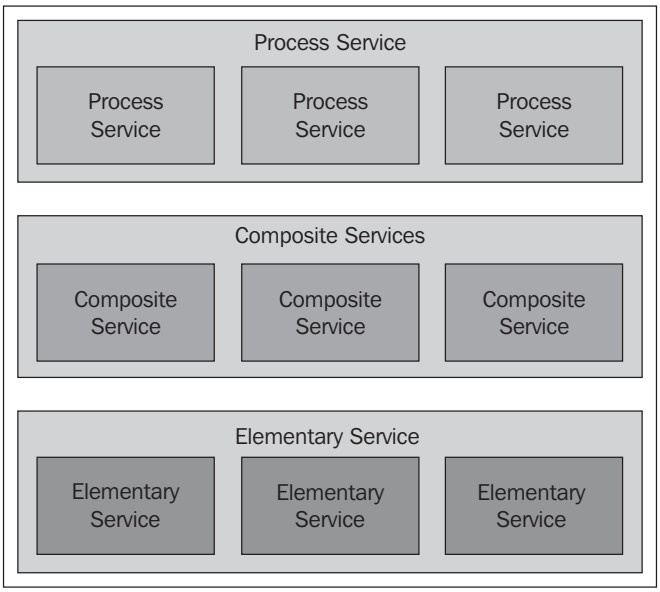
Next, the different elements of the classification scheme will be discussed in detail. Finally, the chapter revisits the design principle of isolation and explains why composition and isolation is a good match.

Service classification revisited

You have learned in the previous chapter that services can be divided into **elementary services**, **composite services**, and **process services**:

- **Elementary services**: The smallest possible components that qualify as services. For example, a service that can be used for zip code lookups.
- **Composite services**: Services that result from combining two or more other services into one service that provides more value. Composite services are executed in a single transaction and their execution time is relatively short. An example is a service to book a hotel and flight together.
- **Process services**: Longer running services that can take a couple of hours, days, or even more to complete and span multiple transactions. An example is a service for ordering a book online. The entire process (order, pay, ship, deliver) involves multiple transactions and takes a couple of hours at least.

The following figure shows this classification of different services:



Example – insurance company

Let's take a look at an insurance company. The company offers a `ClaimToPaymentService`. We saw in *Chapter 1, Understanding the Problem* that this is a process service that consists of the following steps:

1. Receive claim.
2. Review the claim.
3. If the claim is valid, pay the claim.
4. Otherwise refuse the claim.

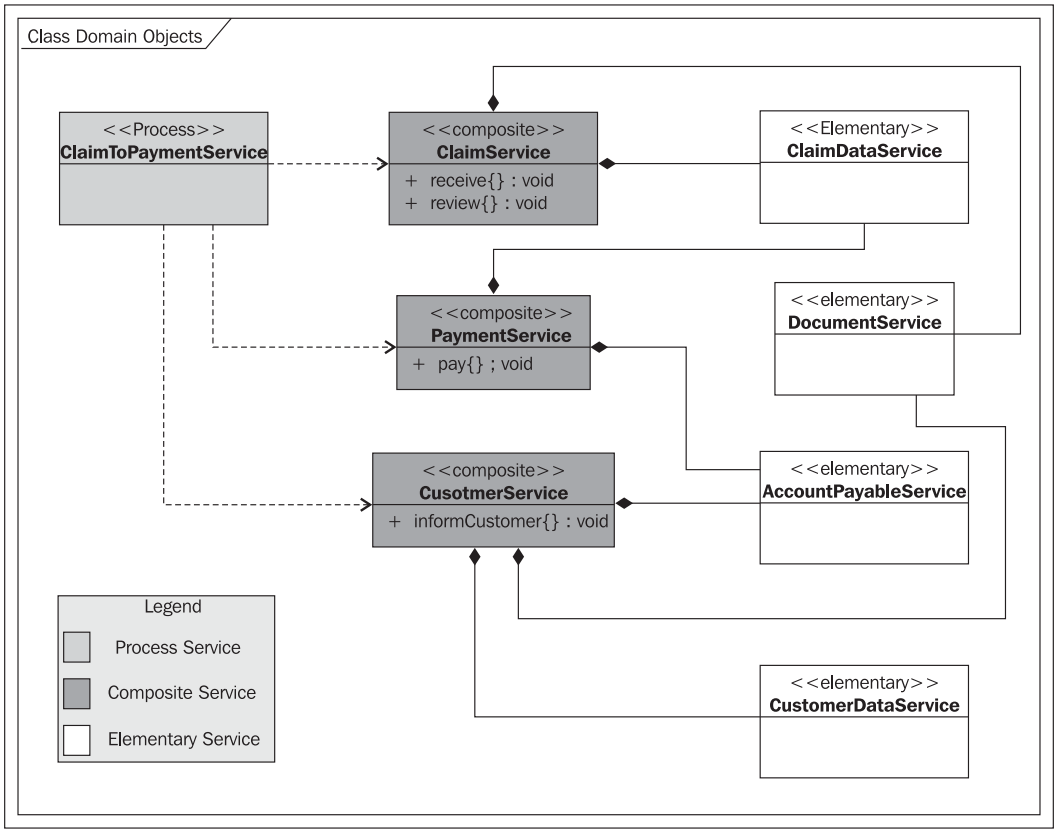
The steps are realized using the following composite services:

- `ClaimService`, operation `receive` and operation `review`
- `PaymentService`, with operation `pay`
- `CustomerService`, with operation `informCustomer`

The composite services are composed of the elementary services:

- `ClaimDataService`
- `DocumentService`
- `PolicyService`
- `AccountsPayableService`
- `CustomerDataService`

The following diagram shows the relationship between these services:



This diagram shows that **ClaimToPaymentService** depends on the composite services to execute. The composite services are composed of several elementary services. As you can see, an elementary service can be part of more than one composite service.

The classification is based on the *granularity of the service* and the ability of *services to combine* into larger ones. Composite services can be created from elementary services and process services can be created from elementary and composite services.

Reasons for choosing this classification scheme are:

- Combining components into larger objects is a natural way of thinking. Consider the assembly of car parts such as tires and engines from small components, and the assembly of entire cars from these car parts. This is a natural way of differentiating between screws and bolts, a cylinder, and a Volkswagen Golf.
- Guidelines and implementation choices differ between these service types. For example, different error-handling mechanisms are required for long-running process services than for short-running composite services. Another example is the programming language used to implement services. It is not a good idea to program an elementary service in BPEL or XPDL; a more fit choice is Java, .NET, .PHP, or any other generic programming language.
- Its simplicity. Classifications can be elaborated into more dimensions, more layers, and more details that altogether provide more information, but more clutter as well. This is needs-based service classification.

There is no one true classification scheme to structure services. You can choose other aspects to classify services based on your needs – what do you want or what do you need to highlight using a classification? Perhaps you are the organization's security officer and security is the main concern you need highlighted in the classification model.

Other classifications

The following list names aspects that are frequently used as discriminators in classifications:

- **Granularity**: We discussed this already
- **Actor type**: Human, IT System, or both
- **Channel**: Telephone, web, and so on
- **Security level**: Public or confidential
- **Organizational boundaries**: Departmental, enterprise, and external
- **Architectural layer**: Business, information, and technical architecture

Actor type

If you base your classification on actor type, you divide services into two categories based on the question, *Who executes the service?*. The answer can be humans, IT systems, or both. Although you might think of services as automated building blocks without a user interface (for example, a web service), services can just as well provide their capabilities through a graphical interface next to (or beside) an automated interface. A `ZipCodeService` and `DocumentService` are examples of automated services executed by IT systems alone. An example of a service that is implemented by humans alone is bank employees who advise clients on the benefits and risks associated with buying a mortgage product. A service for invoice handling in which invoices are automatically stored and processed, but approved for payment by controllers is an example of a service in which the execution is mixed. Both IT systems and humans implement the service.

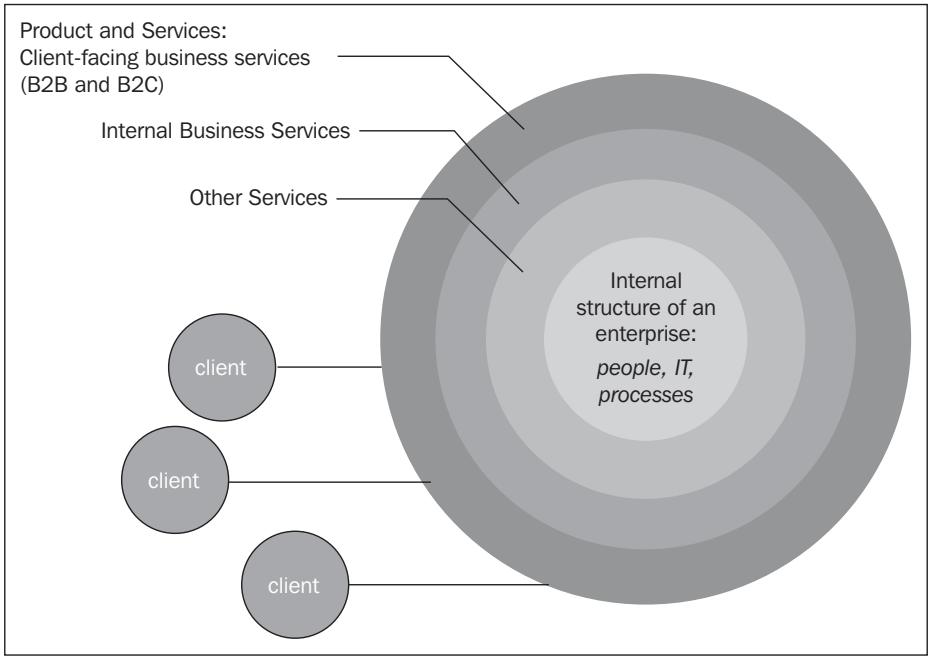
Channel

An organization has different channels to offer services to their customers or business partners. Examples are: mail, e-mail, telephone, internet, intranet, mobile applications, face-to-face conversations, third party resellers, and so on. An example is `BookService` which can be accessed from two channels, the telephone and via the Internet.

Organizational boundaries

The classification based on organizational boundaries is useful to get insight into who owns the service and who uses the service, external services are services that are owned by another organization; an example is a `StockQuoteService` offered by a stock exchange to banks. Internal services are owned by your own organization. These services can either be used by external consumers or by stakeholders from your organization alone.

The following figure depicts this. An organization creates client-facing business services by using its internal services. These business services are either offered to other businesses which is called **Business-to-Business** (**B2B**), or offered to individual consumers which is called **Business-to-Consumer** (**B2C**). An organization deploys people, IT, and processes, which are exposed as services to realize the required functionality.



Security level

Another common classification is based on the sensitivity of the functionality and data that services operate on. This is expressed in terms of confidentiality, integrity, and availability requirements.

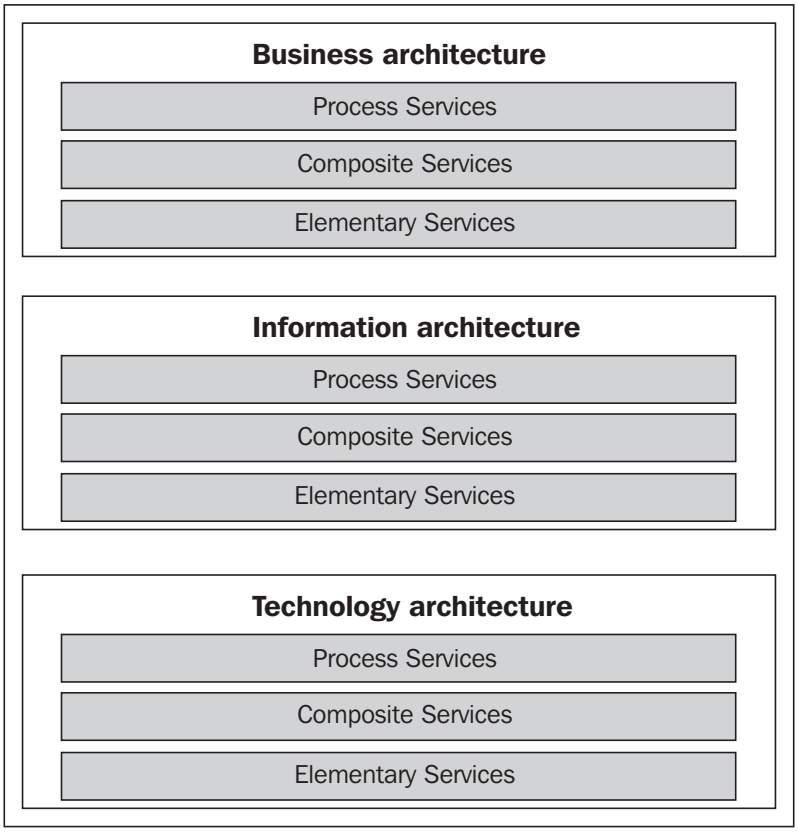
Architectural layer

This classification is based on the architectural layer to which services belong, that is business, information, or technology layer:

- **Business architecture layer**: The services you have identified in this layer are called **business services**.
- **Information architecture layer**: The services that are identified in this layer are called **information services** or **application services**.
- **Technical layer**: The services that are identified in this layer are called **technical services** or **infrastructural services**.

Combining classifications

The classification of services based on granularity can be combined with the classification of services based on architectural layer. The following diagram shows this:



The diagram shows that services in the business layer can be of type – process service, composite service, or elementary service. An example of a process service in the business layer is the `BreakfastService`. This service takes some time to complete, there are different transactions involved and it uses several other composite and elementary services like the `OrderService`, `PaymentService`, `CookingService`, `RefillService`, and `BuffetService`.

The `BreakfastService` is supported on the information layer by several services as well, such as `SupplierService`, `CheckService`, `HRService`, `RosterService`, `PrintService`, and so on. These can be classified as process services, composite services, or elementary services.

Finally on the technical layer there are services that support the information services. These can also be classified as process services, composite services, or elementary services. Examples are the `CashRegisterService`, `NetworkService`, `AlarmService`.

Why classify your services?

Classifying services has several benefits for you and your organization:

- Classification schemes help to focus on important aspects of services and filter out irrelevant details for stakeholders. A classification scheme divides services based on certain criteria. By choosing criteria that focus on important aspects, you can filter out details of services that are irrelevant to you.
- The importance of criteria differs per stakeholder. If the amount of value that services deliver is most important, you can categorize the services accordingly. This will yield a different classification than choosing ownership or the runtime platform of services as criteria in your classification.
- Classification of services also helps organizations to know what (type of) services they own and can be used to support decision-making processes, the architecture process, and new projects that want to reuse existing services.
- Classifying services into categories helps to decide what guidelines, best practices, and technology to apply per type of service. This helps an organization to create better services. Some guidelines apply to all services in general such as isolation and idempotency. However, there are guidelines and best practices that only apply to a certain type of service.

It is feasible to work with services without classifying them if you're only dealing with a dozen services or so in your organization. But as your SOA effort grows, you will find that classifications are helpful to inspect, analyze, and evolve your service's landscape in a controllable way. When you design solution architecture, the classification based on granularity is most suitable. It is based on service composition. Let's investigate that before we delve into the specifics of elementary, composite, and process services.

Composability

Service composition means the creation of new services from combining existing services. Following the design principles from the previous chapter improves the ability of services to be composed into larger blocks, thereby enabling rapid (re) use of functionality. Services need to be composable so that smaller services can be combined into larger services that provide a specific value. Think of it as building blocks that can be assembled into larger structures. A bank's `MortgageService` that provides loans to its customers for buying real estate is not comparable to the bank's `ZipCodeService` that validates zip codes in respect to the type of value they offer. Note however, that the `ZipCodeService` can contribute to the realization of the `MortgageService`.

Aggregation versus orchestration

Sometimes the terms **service composition**, **service aggregation**, and **service orchestration** are used interchangeably. While service composition denotes the fact that services are combined, it does not elaborate on the way in which services are combined. Aggregation and orchestration are specific ways to combine services:

- **Aggregation**: Combining two or more services into larger ones is fairly straightforward (for example invoke the service operation A, then service operation B, then service operation C). The `CaseDocumentService` that was discussed earlier on in this chapter is an example of service aggregation. Service aggregation involves short-running services.
- **Orchestration**: Combining several services into a flow or process that is more complex in nature and contains decision logic and different execution paths. Orchestration involves a central component such as a BPM platform that manages the orchestration of services. Service orchestration involves long(er)-running services. An example of orchestration is the order-to-cash process from *Chapter 2*, *The Solution*.

Example – DocumentService as a composite service

Another example of service composition is the combination of a `PDFService`, `SigningService`, `EmailService`, and `CRMService` into a `DocumentService` that is capable of creating documents in an automated fashion, storing these in the Document Management System, and e-mailing them to the customers. This new service can for example be used in an order-to-cash business process to automatically create invoices and send them to customers. Through composition, rather than development, you have created a larger service that makes it possible to completely handle a business function for consumers. This is a faster way of creating new services than developing similar functionality over and over. It is also less error prone since you reuse already tested and used software.

It is now time to investigate the different layers of the classification in detail, starting with the elementary services and then work our way up to the process services.

Elementary services

Elementary services are the smallest possible components that qualify as services. Splitting elementary services in smaller components yields components that violate the design principles you learned about in *Chapter 3*, *Service Design*. For example, splitting a `DocumentService` that is capable of storing and retrieving documents into a separate service that only stores the document, and another service that only stores the metadata violates the principle that services should be isolated or autonomous.

Realization

Elementary services in the information layer can be implemented in several ways:

- Existing applications such as packaged applications, custom developed systems, and legacy systems that are exposed as services using connectors, adapters, and so on. This can be coined as service-enablement of existing systems. An elementary service is implemented by one system, although a system can implement more than one elementary service.
- New functionality is created as a service right away. In other words, software is developed in an isolated manner and deployed as a separate service instead of a monolith that offers several services. Popular implementations for automated elementary services are .NET, Java, PHP, Groovy, among others. These platforms offer web service frameworks that can be used to easily expose software components as web services.

Composite services

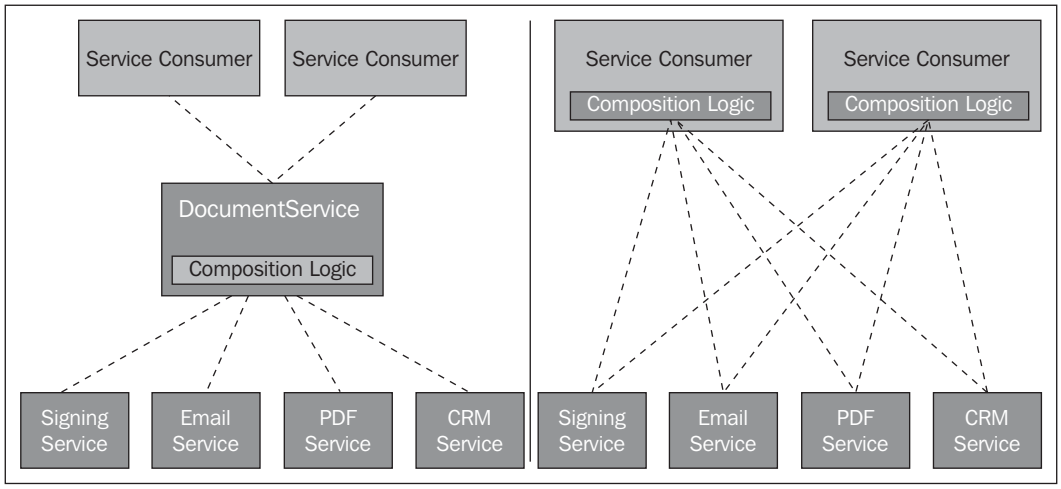
Composite services are services that result from combining two or more services into a new service in a straightforward manner, and provide more value. Composite services avoid the need to implement reoccurring composition logic in clients.

Where to put the composition logic?

There are two options for the location of composition logic:

- **Service consumer**: The client invokes several service operations to accomplish a certain result.
- **Service**: A new service is created that invokes the different service operations and exposes this functionality as a new, single operation.

The following figure shows the difference between these variants using the example of the `DocumentService`. On the left-hand side, a `DocumentService` is created in which the `generateDocument` operation contains the logic to combine several underlying services. Service consumers invoke this new operation of the composite service. In the right-hand side service consumers implement the composition logic.



It pays off to create a new service that contains the composition logic in case multiple consumers require the same composition logic:

- Changes to the composition logic can be applied more easily since it is implemented in a single location instead of in different service consumers. Similar business logic is reused in the service that implements the composition.
- When you compose services into larger ones, the knowledge and complexity of how to combine these services and how they interact is hidden in a new service itself (**implementation hiding**). This makes it easier for service consumers to use the functionality.

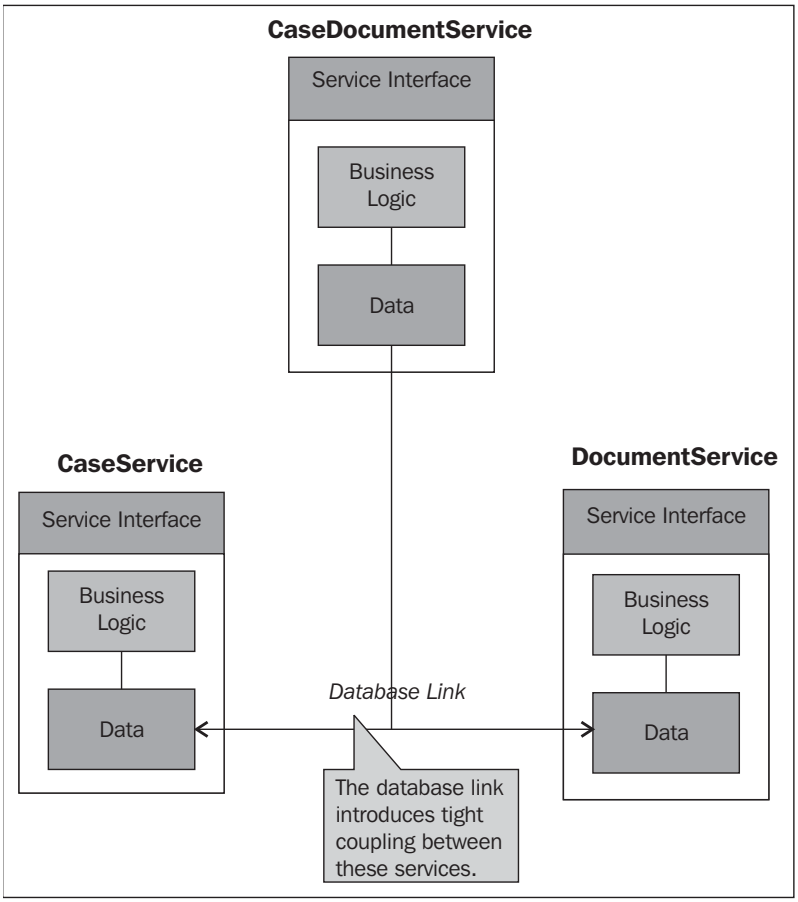
When services are composed it is important that the entire service fails or succeeds. You do not want to end up in a situation where the document is generated and sent to a customer but not stored in the Document Management System due to some error. This means that a number of aspects need to be addressed in the composite service that relate to the behavior of the combination of the elementary services, not the elementary services in isolation. Such aspects are fault-prevention and handling, security, and transactionality.

Implementation

Composition of different building blocks can be implemented in several ways. Let's look at two examples. In both examples we use an insurance company that uses a case management system and a document management system. When claims are received, both systems are used to register the claim. For every new claim, a case is created using the `CaseService`, the accompanying documents are stored using the `DocumentService`, while the relation between case and documents is stored in the `CaseService`. Retrieving the documents belonging to a specific case requires two invocations: retrieving the document references from the `CaseService` and retrieving the physical documents from the `DocumentService`, using these references as input to the `getDocuments` operation. When such interactions occur frequently, you can compose this reoccurring logic into a new `CaseDocumentService` that offers a single operation for this functionality.

Example 1 – database link

The following figure shows `CaseService` that stores case information in its database. The case-related documents are retrieved using a database link that provides access to the `DocumentService` database. For the case data the data is retrieved using a database link to the datastore of the `CaseService`. For the business logic of the `CaseDocumentService`, it looks like the document data is in its own datastore.

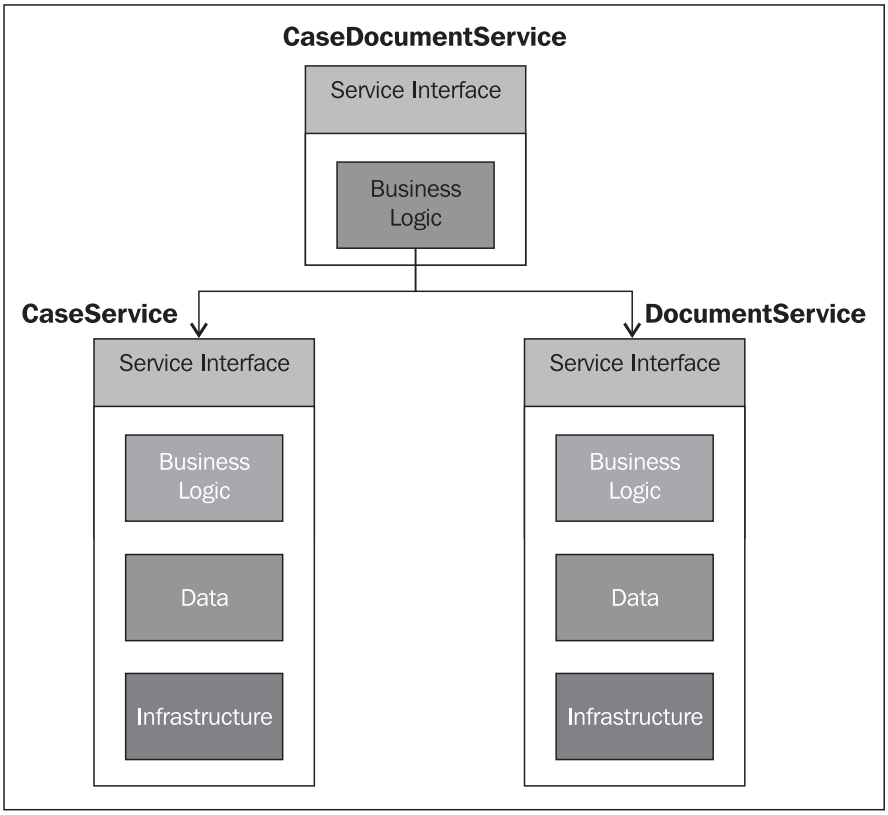


The figure shows `CaseDocumentService` that consists of a Business Logic layer and a Data layer. The `CaseService` and the `DocumentService` are composed of the same layers. The Business Logic of `CaseDocumentService` needs access to the data in `DocumentService` and `CaseService`. The database link offers this access, by creating a route directly to the database tables via the database layer of the `CaseDocumentService`.

When using a database link to directly access the `DocumentService` you need to have specific knowledge of the internal data structure and technical details. Whenever the internal implementation of the `DocumentService` changes, it will most likely impact the `CaseService`. In other words, the database link introduces tight coupling between the services.

Example 2 – service invocation

Not every change to `DocumentService` has an impact on its clients. It could be that a bug fix to the service affects the internal data model, but not the functionality that its clients need. Consumers should be unaware of such changes. In the previous scenario this would not have been the case since the service implementation was accessed directly. A better scenario would be to invoke the `DocumentService` using its service interface. The interface only exposes what the clients need, it hides the complexity of the service and its internal implementation, and shields service consumers from changes in the service that don't affect them.



The figure shows the situation where the composite service `CaseDocumentService` combines the logic of the `CaseService` and the `DocumentService`. The `CaseDocumentService` contains business logic that was located in `CaseService` in the first example. The `CaseService` only contains business logic that is relevant for case management and `DocumentService` only contains business logic and data that is relevant for document management. You can see that there is no direct link anymore between `DocumentService` and `CaseService`.

Whenever one of the services is changed, this change is advertised through its interface in a technology-neutral way. This way it is easier to cope with changes to underlying services and their implementations. It could even be that changes in the implementation have no impact to service consumers at all. Composing services through their interfaces instead of implementation makes services work together instead of tightly-coupling them together.

Process services

Process services are services that use other services to execute their flow, like composite services. However, they take longer to complete and involve multiple activities and transactions. Process services invoke automated and human steps. The logic to combine these operations is more complex and contains more conditional logic than in composite services.

Examples of process services are invoice handling, the on boarding of new employees, a mortgage service, and so on.

The difference between process services and composite services is somewhat of a grey area. The main differentiators between the two types of services are:

- **Duration**: Process services are long-running processes that can take hours, days, weeks, or even longer, while composite services are short-lived services that take at most a couple of minutes to complete. A mortgage process that includes preparation, application, and underwriting, for example, can take weeks.
- **Transactions**: Process services can span multiple steps, or transactions; composite services are executed within a single transaction. A process service that spans multiple transactions needs other mechanisms to be undone. A common mechanism is compensation in which transactions are undone by executing an opposite action. An example is online booking of your holiday wherein you book a flight, a hotel, and a rental car. If you abort the holiday later in the process while the money has already been charged from your credit card using a service call, compensation means the money is deposited on your card by invoking the same operation with the opposite amount.
- **State**: Due to their complexity and duration, process services are required to maintain state between the different service operations that are invoked. For example, state can be used in a process service as part of a decision logic later on in the service. Composite services are stateless or volatile.

Implementation

Often, a business service is realized by orchestration of other services into a process. Every step in the process contributes to the end result of the business service that is offered to clients. A Business Process Management tool can be used to implement this orchestration. Most process services include both automated and human-centric services. A mortgage product for example is achieved by human steps (bank employee informing the client of the different mortgage products, a controller that validates the client's financial history and calculates risk) and automated steps (automated underwriting, automated debit for mortgage payments).

The term **Straight-Through Processing** is used to denote process services that are realized entirely in an automated fashion. An example is an `InvoiceService` that processes inbound invoices (scanning, extracting metadata, storing financial data, order matching, payment) without human intervention.

That finishes the explanation of the different layers in the classification scheme. The chapter continues by revisiting isolation and composition in light of the classification scheme that was introduced.

Isolation and composition – a contradiction?

The design principle **isolation** that was introduced in the previous chapter and **composability** that is introduced in this chapter might seem to contradict each other. How can you have autonomous services if you compose larger services out of smaller ones? Wouldn't that mean that the composed service is dependent on the smaller services?

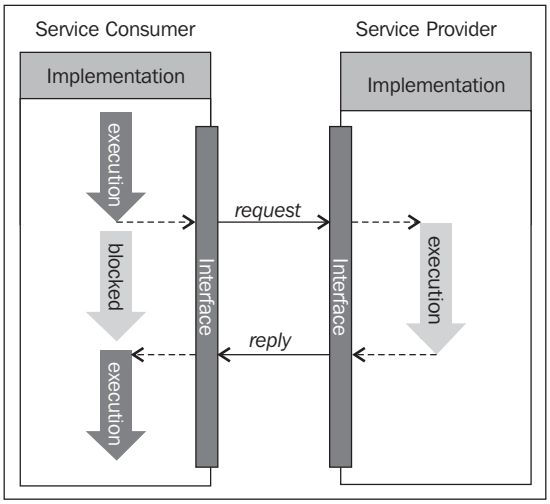
The answer is that service composition promotes isolation of services. Isolation is viewed from the perspective of consumers, a consumer should be able to invoke a service independently without needing to know if other services need to be invoked before or afterwards. Service composition wraps such dependencies and knowledge of how to combine them into a separate service, thereby hiding this from the consumer and offering a new autonomous service.

Isolation is also known as **loose-coupling**, too many dependencies between services make them tightly coupled and therefore harder to change. As you know from the previous chapter, changes to services should have minimal impact, to maintain flexibility in SOA. Only create *direct* invocations from services to other services that are *in the same or higher level* of the classification when really needed. For example, it is better that the elementary `DocumentService` should not directly invoke the composite `CaseDocumentService`. Introducing this dependency would mean that changes to the `CaseDocumentService` can impact the `DocumentService` and vice versa. When avoiding the invocation from the composite service to the elementary service, changes to `CaseDocumentService` do not directly impact `DocumentService` anymore.

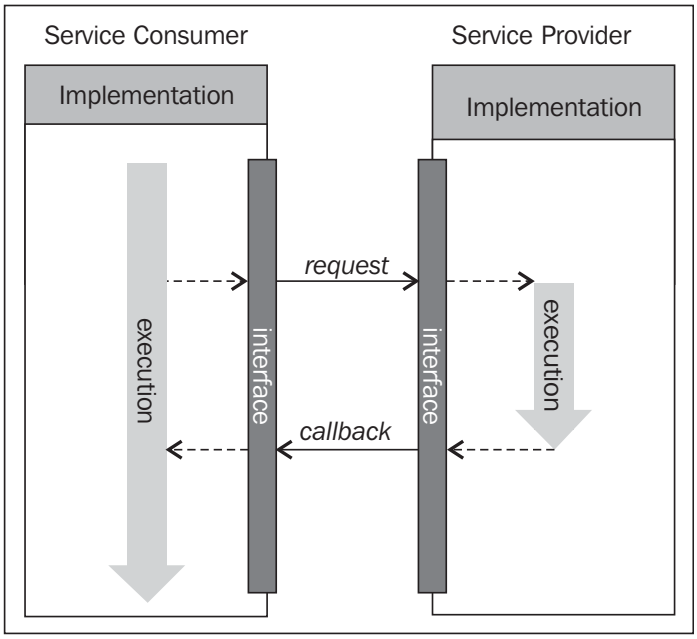
Passing information from smaller to larger services

Information and data need to be passed from elementary services to composite services or process services. The following list names different mechanisms that can be applied for this:

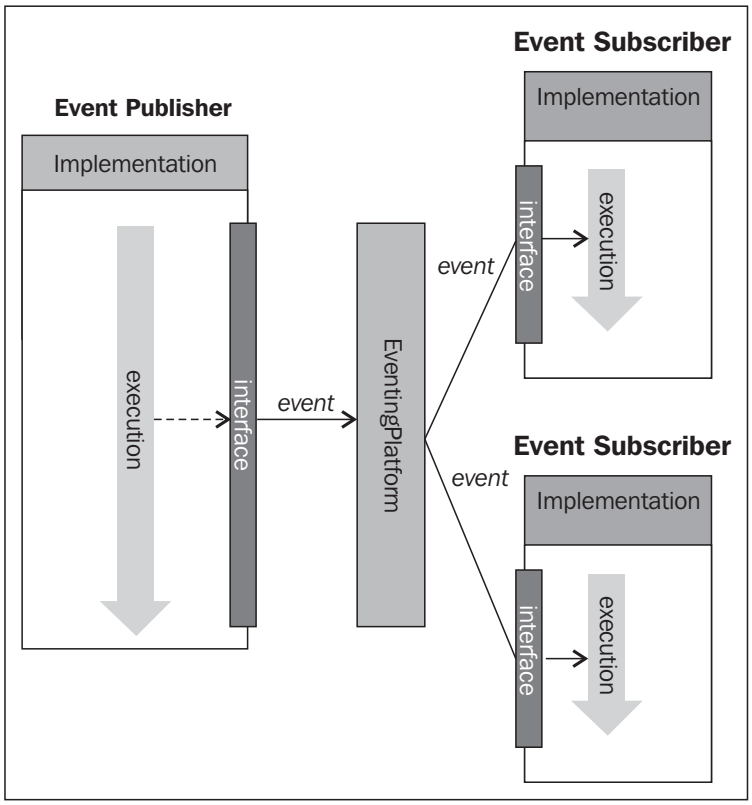
- **Request/reply**: In this message exchange, a service consumer invokes a service by sending a request message; for example, a composite service that invokes an elementary service. The invoked service responds by sending a reply message that can contain information for the service consumer to act upon. In this case, the data can be used by the composite service. This type of invocation is synchronous, meaning the consumer is blocked between the time the request was made and the time the reply was received. The following figure shows a request/reply message exchange:



- **Callbacks**: Next to synchronous request/reply messages, services can also define callback operations in their interface. When using callbacks, the consumer sends the request and can continue with other activities. At some later point in time, the service that was invoked can send the response as a callback message to the consumer. Callbacks are used when consumers invoke long-running processes. Callbacks still introduce coupling between services, although this is made explicit by denoting operations as callbacks. The following figure shows a callback message exchange.



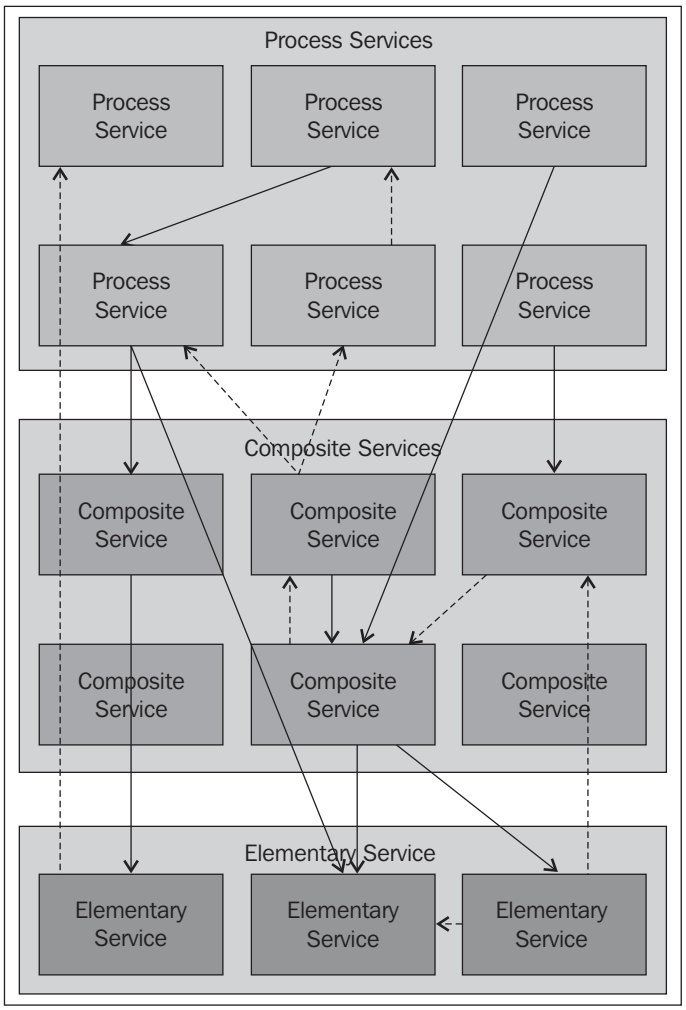
- **Events**: An event is the occurrence of something that is likely to be relevant to one or more consumers; for example the creation of a case, the approval of a loan, a new employee, and so on. In eventing, the originator of the event doesn't know which consumers are interested in what events and what actions these consumers will perform based on the events. The originator just publishes the event to some eventing platform. The platform knows which consumers are interested in what events and makes sure that events are sent to these interested consumers. This increases the decoupling between services. The following figure shows a message exchange based on events:



There are some important differences between these mechanisms in the light of passing information from smaller services to larger services:

- **Trigger**: While request/reply and callback mechanisms can be used to pass information from smaller services to bigger services, the bigger service that invokes the smaller service still triggers the information exchange. In eventing, the trigger of the message exchange is the originator of the event—the service that generates or captures a relevant occurrence. This publisher can be any service, and consumers of these events can be any service.
- **Decoupling**: Message exchanges using request/reply and callbacks require service consumers to know service providers, and the specific operation (that is action) to invoke. In eventing, publishers of events have no knowledge of the consumers and the operations that are triggered by the events they publish.

The following figure shows the communication between the different types of services. The solid arrows indicate service invocations from one service to another. The communication of information between services through events is shown as dashed arrows in the figure:



The figure shows the process services that call other process services, composite services, and elementary services. Composite services can call other composite services and elementary services. Elementary services don't call other services but use events to communicate. To minimize the complexity you can create design guidelines. For example you can limit the number of calls to elementary services that a composite service can make. If you exceed that number, it becomes a process service. Other guidelines are that elementary services are not allowed to call other services.

Events can be classified as well as services. Examples of such classifications are those into **internal events** and **external events**, or a classification into **elementary events** and **complex events**.

Elementary events are those events that in themselves are noteworthy, but when divided into even smaller occurrences are not meaningful anymore. Think of the events that signal the start and end of a telephone conversation. These are meaningful to the network operator to determine the duration and possibly cost of the phone call. The occurrence of the fact that one of the persons having the phone conversation presses the red cancel button to stop the call in itself isn't interesting to the network operator.

Complex events are those events that are combined from smaller events to create a new and more meaningful event. An example is the occurrence of an ATM withdrawal by a particular client in New York at 13:00 GMT and an event indicating an ATM withdrawal by the same client in Amsterdam at 14:00 GMT that same day. These events can be combined into a complex event for the bank that issued the ATM card indicating the possibility of fraudulent activity. Complex events often involve a time aspect. If certain events occur within a specified timeframe, a complex event is generated that correlates these smaller events.

Summary

In this chapter you have learned how services can be categorized into a classification. This chapter introduced a classification into elementary services, composite services, and process services by looking at their granularity and the way services can be combined into other services; this is called composition.

In the next chapter, you will learn about the concrete building blocks that are used in SOA landscapes. These among others include business rules, user interfaces, BPM tooling, and Enterprise Service Buses.

Where to buy this book

You can buy SOA Made Simple from the Packt Publishing website: `http://www.packtpub.com/service-oriented-architecture-made-simple/book`.

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

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

www.PacktPub.com