Design Principles and Patterns

WHAT’S IN THIS CHAPTER?

➤ Exploring services and SOA
➤ Understanding communication and integration patterns
➤ Working with business process patterns

This chapter describes a number of principles and patterns regarding Service Orientation, Integration, and Business Processes. You will see how these principles are related to WCF and how you can use WCF to implement the patterns.

WHAT IS SOA?

SOA or service-oriented architecture is a style of programming, an architectural approach in software development, where an application is organized in functional units of code with a given behavior called services.

Services are a group of methods that share a common set of requirements and functional goals. They are called by other parts that need to execute its logic, depending on the outcome (such as data, results of calculations, and so on). The functions have a clearly defined and public signature which is published so other code (service clients) can use the functions in the service as a black box. The service operations are invisible — there is no direct interaction with a user and the work is executed as instructed by the given input parameters. SOA allows distributed applications to be organized. This means that the service consumers are running on different machines than the services. This allows the business logic and run user interfaces to be centralized or other consumers to be decentralized across the network. To make this happen in SOA, structured messages are sent between machines containing data.
The main idea behind SOA is to create a loosely coupled system where the only element a consumer of a service and the implementation of the services have in common is the list of public service operations and the definition of the structures of the parameters.

The client only knows the signatures describing the name, names and types of input parameters, and return type of the functions of the service. There’s no other dependency. The application platform and programming language can be different on both client and service.

It’s clear that this dependency is a functional one and is not based on technical infrastructure. This makes it possible to interact easily from different platforms with a service. A technical cornerstone in the SOA paradigm is the use of the SOAP standard. SOAP is the XML language that defines the content of messages sent to and received by a service operation.

Messages are formed out of the value of the parameters or return values and the data is formatted by SOAP. Every development platform has a SOAP stack, so working with service is supported in many environments. Supporting multiple environments is the goal of working in a SOA style.

This approach makes it possible to create systems that are built out of services. The services make up the building blocks for the application which can be composed out of the service operations. Either an end user application or another service can make use of these building blocks. SOA makes it possible to define a workflow of a business process in which they make calls to service operations.

Implementing an application in this architecture is the way to make code and the functional behavior of it reusable for unknown uses in the future. As the business logic is not coupled to some kind of user interface technology, it’s possible to access these functions from clients that use newer technologies for creating user interfaces.

Separation of concerns is also an advantage. When structuring a development team for a project, different sub teams or individual members can be assigned to both sides of the service boundaries.

One team concentrates on building only the user interaction experience without concern about the code dealing with the business logic and data access. The UI team receives the service interface and can start coding against this interface. Meanwhile, another team works out the implementation of the defined service interface without the need to build a user interface. This means a developer is no longer responsible for the code end-to-end, including user interface, business logic, and data access for a given requirement. This results in assigning developers which can focus their knowledge of technology for one layer of the complete application.

Separation of concerns also means that the development of UI and services can be started at the same time directly after the publication of the agreed service interfaces. This is a huge advantage that allows the creation of the UI to be outsourced and keeps the creation of the real business logic in-house.

SOA is a way to build distributed systems where autonomous logic is called using loosely coupled messages via a well-defined interface.
Having a stable definition of a service interface is absolutely needed. The advantages of SOA are only present when the service contract is agreed on by multiple parties and is not subjected to changes during development. It’s the business, defining the requirements, that has to have a clear view of the needed functionality. This is done in combination with a functional architect that defines the interface on a technical level. Of course this is not always easy or even possible as business requirements tend to change a lot in most environments. To solve this contradiction, it’s wise to have an iterative development process that typically lasts 1 to 4 weeks. The service interface is not changed and the modifications to the interface are clearly discussed and reported to the development teams at every new iteration.

As application and software systems get bigger and more complex, a strict development architecture is needed that supports a great maintainability with a reusability of components. In the more distributed environments we see today — with applications implemented on different platforms — the need for a simple development approach that supports interconnectivity is a big value.

Implementing a SOA architecture is needed to solve problems that object orientation alone cannot solve for very large systems with integration between multiple parts. Integration of existing components needs a well-thought-out and industry-wide paradigm in the form of SOA.

FOUR TENETS OF SOA

To have a good and deeper definition of SOA, some principles need to be described in more detail. Tenets in the software industry are the way to do this. In SOA, the four tenets of service orientation are discussed.

These tenets include the following:

▶ Boundaries are explicit
▶ Services are autonomous
▶ Services share schema and contract, not class
▶ Service compatibility is based on policy

Boundaries Are Explicit

When working in a SOA approach, the boundaries a consumer needs to cross to reach the implementation should be defined explicitly. Services run in a process and memory space separated from the clients using them. Up-front thinking is needed to define the boundaries and should be communicated to each possible participant. The boundaries are defined by means of the contract and the address where a service can be reached. This information should be considered important and be easily accessed.

It is impossible to execute the logic in a service without having a contract and address. The logic is allowed to execute only in one way. This is by calling the contract, which is considered as the boundary. Boundaries are explicit, which means the client only needs to be aware of the existence
of functions in the service that can only be executed via the contract. This tenet also means that all possible exceptions must be described and a method can only stop executing by either giving the needed answer as an explicitly known data structure or as a structure containing the details of the exception. No data enters the service operation and no data leaves the service operation without a clear allowance to do so.

**Services Are Autonomous**

Services are considered standalone pieces of code that do not rely on the behavior of other services. Services are considered available without the need to explicitly instantiate it. They should be deployed and versioned independently from each other, and installing a new version of a service should not influence the behavior of other service operations. Services should not be coupled to each other as classes are coupled in a compiled executable. Services should instead use a loosely coupled architecture.

**Services Share Schema and Contract, Not Class**

A schema is the definition of a service operation and describes the signature in a platform-neutral way: the name of the functions, types of parameters, and the type of return value. A contract is considered metadata for the service being a black box with only this well-described interface. Schemas are the definition of the structure of the parameters. This tenet clearly indicates that the class itself (in code or as UML notation) is not shared across services and their clients.

As SOA is aimed at cross-platform interaction between different parts, it is not useful to bring the code for a class to other parties. In a lot of cases that code is meaningless. The inner workings (behavior of the code) of the class are not relevant to the consumer. The application (or other service) that uses a service is only interested in the outcome of the service operation. Clients send messages to part of the schema operations that conform to this contract to obtain this outcome.

Clients should interact with a service and back through an explicitly defined public interface including faults. Each version of a service should have one version of the interface. After an interface is in production it should not change, as changing the interface would be the result of modifications in the behavior of some service operations and should result in a new version of the interface.

**Service Compatibility Is Based On Policy**

This tenet means that a service decides which conditions process the message. A policy is used to negotiate elements in the communication, such as message format and security requirements. The policy is there to further indicate the semantics of the service and its expectation of the behavior on client side.
ANATOMY OF A SERVICE

A human body and a service both have an anatomy containing different parts related to each other that form a whole. Every part has its own purpose and behavior in the system. Describing this anatomy helps you understand how services work and is an introduction to the technical details of an SOA implementation. See Figure 1-1.

A service containing methods uses a channel to communicate and to be reachable by service consumers. The service consumers also use a channel, considered to be compatible with the service channel, to actually call the methods and send the needed data to the service. A channel is the combination of the schema, contract, and the policy on one hand and the used protocol at runtime on the other. Messages are sent through this channel in either direction. The channel is bound to a protocol defining in which way and how the service is reachable. A protocol, such as HTTP or MSMQ, is there to transport the data and needs to be supported by the operating system platform on which services are implemented. A channel is a kind of pipe in which messages flow. Messages are put on the channel by clients (or other types of service consumers) and taken out of the channel by the hosting stack of the platform that publishes the services.

A channel is bound to the schema it contracts. The channel is not complete without the definition of the metadata of service operations contained in schemas and contracts.

The channel also knows the policy which a service consumer has to implement.

Service Ecosystem

From a higher point of view a service is living in an ecosystem where a few concepts form part of the SOA paradigm. This ecosystem describes the place of these concepts and how they are related to each other. See Figure 1-2.
Applications Are Composed of Services

The heart of the ecosystem is the service itself. Services are the building blocks out of which applications can be composed. Applications can be end-user tools with clear user interface parts or business processes which access the services in a predefined sequence. As the SOA approach allows you to see the services as units of behaviors, you can build applications by picking the ones needed to compose another part of the software solution.

Services Manage State

The responsibility of a service and its operations is often to persist data to a database and read the data again at a later time. A service is actually managing a state. Although services are stateless, meaning they don’t remember this data in memory, they are calling to a database to persist the state. Services will not call to a database directly, but in a good architecture they will depend on other layers to communicate to the database.

Services Enforce Policies

A service has the right to define policies concerning the usage of its logic. Policies describe prerequisites about how the service consumer should behave. You can think of a policy being an agreement on a condition that has to be fulfilled before a client is allowed to communicate with the service. Most often this is an agreement on security.
Policies Enforce Operational Requirements

By defining policies, a service can enforce the operational requirements of the calling platform. The policies can be organized in such a way that a certain measure of security must be implemented on the client.

Services Are Bound by Contracts

Services only exist if a contract is present which describes the signatures of its operations. This contract is the agreement between client and service. The contract has to be defined clearly and bound to the service at runtime. The contract is needed to create a proxy class for the client tier used to program against the service as if it is a local class. Without a contract the service cannot be consumed by a client application.

Contracts Describe Message Exchange Patterns

A message exchange pattern is the definition of how and in which sequence messages are sent from one party to another. The patterns influence how the service can be called either synchronously or asynchronously and defines whether answers are expected from the operation or not. Message exchange patterns can be the following:

- **Request-response**: Most-used pattern, every call is returned by another message directly.
- **One-way**: There is no answer coming from the service and the operation can thus be called asynchronously.
- **Duplex**: A service operation can call back to the client during the method call. In this way the operation can ask more information from the client before returning the final answer.

A message exchange pattern is visible on a functional level where developers implement service operations. On a deeper technical level of the protocol, a message exchange pattern is present but mostly invisible. When using a `wsHttpBinding` for calling an operation, client and service will not only exchange the message with the data. The functional call will be preceded and followed by some other technical messages. We can consider these messages as handshaking messages that are part of the WS-protocol. These messages can negotiate a security context. They are also needed to ensure reliability that the service has received the data and that the service can be sure the client has received the answer as expected. This additional communication is done by the protocol and is not to be implemented by the developer.

Contracts Contain Schemas and Schemas Define Message Structure

A schema is the definition of the structure of the parameters for an operation. It describes these parameters in terms of an XSD file. XSD is the metadata language that defines messages going into a service operation or the result of a service operation. Schemas are used so service operation consumers can format data. It can be interpreted by the service when called so the service can access the data again. This is called serialization and deserialization.
A Message Exchange Pattern Is a Set of Messages

The combination and calling order of messages can be described by a more complex exchange pattern. So a message exchange pattern can define which operation must be called first and which one last, or if a complete workflow of the operation can be defined.

Services Exchange Messages

Exchanging messages is the most important part in a service ecosystem. Exchanging messages means calling an operation and receiving answers from it (or raising exceptions). Exchanging messages is simply a way to allow that operations are called on another machine. A message transports the input parameters from client to service and another message transports the answer back to the caller.

ORCHESTRATING SERVICES IN BUSINESS PROCESSES

As services are building blocks of applications in a service-oriented architecture, the operations in it can also be called by a workflow. This workflow is a definition of the order and dependencies between incoming and outgoing calls. The workflow orchestrates the interaction between consumers and services to fulfill a business process. Workflows know the contracts and schema of the services they are using or implementing.

A workflow is like programming sequences and branches around receiving operations and calling other operations in other services. Instead of programming these sequences and branches in a programming language, the flow is defined declaratively as a meta-language. This meta-language can be interpreted to form the integration between multiple parts of the application at runtime. In this way a workflow describes a very functional and complex message exchange pattern. As the main goal of SOA is building reusable components for an application, the use of workflows is an enabler to define the logic of integration and business processes.

The requirement of a business process forms the base of the definition of the contracts and schemas. Looking at these requirements should be the start of the analyzing process.

Figure 1-3 shows an example of an orchestration.
TECHNOLOGIES BEHIND SOA

To implement an SOA application, you need to have technologies and protocols. As SOA is intended to create distributed and cross-platform applications, these supporting technologies and protocols need to be industry standards. The following sections describe some of the most used standards in the SOA world.

SOAP

Simple Object Access Protocol (SOAP) is an XML specification for exchanging data as structured information in messages. SOAP standardizes how data is exchanged on the wire. As it’s based on XML, it is platform agnostic. A SOAP message simply carries the data as a message. A SOAP envelope contains a (optional) header and a (required) body element. The header can contain information needed for the underlying technical infrastructure to support the communication and is not related to the business functionality. The body element contains the functional data as payload. Each parameter for the service operation is present in the body as the serialized representation of the data. This is an example of a SOAP message:

```xml
<soap:Envelope
    xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
    soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
    <soap:Body xmlns:m="http://www.example.org/stock">
        <m:GetStockPrice>
            <m:StockName>XYZ</m:StockName>
        </m:GetStockPrice>
    </soap:Body>
</soap:Envelope>
```

WS-* Protocols

SOAP is only a specification describing how functional data in the body and technical data in headers is formatted. SOAP itself does not always define a meaning to the header. WS-* are protocols that standardize how to implement certain needs and behaviors when working with distributed messages using SOAP messages. These protocols describe how the plumbing needs to exchange messages in a secure, transactional, and reliable way by using the headers in the SOAP message. WS-* is a set of protocols where each protocol has its own purpose.

WCF can implement the WS-* protocols by using a binding called WsHttpBinding. This binding makes use of some of the WS-* protocols and adds the needed behaviors, such as transactional message calls, reliability, discovery, and addressing.

WSDL

WSDL is a XML-formatted definition of the contract. It contains all metadata for the interface of the service including function names, parameter names, and their types and the types of return values. The purpose of a WSDL file is to define this contract in a cross-platform way as the types are expressed in XML types. The WSDL file can be used by non-.NET development environments to
create classes in their programming language (such as J2EE) to act as proxies for the real centralized implementation class.

See Figure 1-4 as an example of a WSDL file. This file can be shown in a browser when browsing to a dedicated URL on which the WCF service exposes metadata.

![Screenshot of WSDL file](image)

**FIGURE 1-4**

**CONTRACT-FIRST PRINCIPLE**

Designing services for an application starts with analyzing the requirements. It's not a good idea to open Visual Studio first and start coding immediately. Analysts should start by working together with the people needing the application and carefully interpreting what they say the requirements are for the services. Meetings with all stakeholders are needed to define what the services should do and what their logic is.
The first thing that should be clearly defined is the contract of a service. Analyzing a service-oriented application is not about drawing screens for the user interface or defining tables and building a relational diagram for them. It’s about defining the contract first.

Of course, at the end of a functional analysis there are three basic layers in an application:

- **UI**: Contains the screens, validation logic, and interaction between user controls
- **Logic**: Implements the requirements, business rules, calculations, and reporting
- **Database**: Stores data and referential integrity across tables

In a SOA approach, it’s clear that UI and database are not the first layers being analyzed. SOA is focused in the business aspect and deals with separating the logic from the look and feel of screens and how the data is stored. Neither UI nor database is really related to the business requirements. In most cases the business is not interested in how the data is stored or how it’s represented to the end user.

You want to make the logic contained in services accessible to multiple user interfaces. The user interface is developed by another team that is only responsible for the UI and has no relation to the business. Remember that services are black boxes; they are used to get results from them.

You want to make the logic independent of the database store. It’s up to the DBA administrators to come up with a design for the tables unless the database is already in place.

This leads to the contract first principle. The first thing you should analyze is the contracts. You should have a clear view of what to expect from a service, what methods are part of the service, and what the structure of the parameters for these methods are. The design of this contract is very much influenced by the business and should be driven by it. It’s up to the software and business analysts to define a contract.

**HOW WCF AND .NET SERVICES IMPLEMENT SOA PATTERNS**

Patterns are a description of a reusable solution to a well-known problem in typical situations. The idea of a design pattern is to describe the solution for the problem in an understandable language before it’s translated to code. One could say that a design pattern is a blueprint or template for a possible solution. It is used for discussing the problem before programming it in a language. It is agnostic of the development stack or of the used technology.

**Patterns**

Patterns are languages used by architects or the designers of the needed integrations where people can start discussing and using the patterns in their communication.

Phrases such as “I would suggest using an enricher pattern before the messages are sent to the broker . . .” and “for this functional requirement I don’t see the need for a correlation pattern here . . .” are typical in conversations when doing an analysis for service-oriented or integration-based solutions.
These phrases are clear enough for the process designers and architects and are a valid way of communicating before development starts. After designers agree on the design they can draw it in a schema. Design patterns often have a counterpart as a schema with a big diagram describing the whole picture of communication and integration. This schema contains a combination of the patterns and is the ideal base for developers to create code and system engineers to set up the implementation.

Decoupled Contract: Interface vs. Implementation

WCF supports the separation and implementation of interfaces by using interfaces as part of the C# or VB.NET languages. All operations, their types of parameters and type of return values for a service, can be expressed in interfaces. These interfaces can be created in a separate class library project. The data contracts describing the structures of parameters or return values are also defined in classes.

Typically these interfaces and classes are stored in a class library which contains no implementation. The implementation of the services is done in a different class library. It’s just a matter of referencing the needed interface library from within the implementation project and implementing the interfaces in the classic way that is supported by .NET languages.

This separation allows sharing the interface with other projects that need them but cannot provide the implementation as with the client applications.

Interfaces and their operation signatures are attributed with metadata attributes so WCF can recognize them as contracts that are part of the service ecosystem. These attributes are placed on top of the code elements and they form an additional layer of metadata which is understandable by the WCF runtime to give the code meaning in terms of contracts and schemas. The metadata for defining these contracts and schemas contained in the attributes is thus separated from the interface, and the implementation only knows the interface. This makes the contract decoupled from the implementation. In the metadata the name of operations and XML structure of the SOAP message can be defined without the implementation having to know these.

Those attributes are [OperationContractAttribute], [DataContractAttribute], and [DataMemberAttribute].

Proxy Pattern

The proxy pattern is used by WCF by allowing building and using a class in the project which consumes the services. This class acts like it is the implementation but it’s actually a class that implements the same shared interface as the implementation of the real logic. As the interface together with the extra metadata for the SOAP messages defined in attributes is also available in the client project, this proxy class can mimic the implementation by implementing the interface.

WCF provides the ClientBase<T> class for these proxy classes to inherit from. This base class has the needed logic to set up communication with the service and transform the operation calls. The consumers execute against the proxy into SOAP messages and send them to the service using the defined binding.

ClientBase<T> is a generic class that needs to know the type of the interface. This class has a protected property, only visible in the classes that inherit from it. This property is the channel and
is of the generic type used by the proxy. This means the channel has the same list of methods as the actual implementation of the logic on the server side.

**OperationContext Pattern**

The goal of the OperationContext pattern is to decouple the functional input parameters from technical information the method needs to execute. WCF provides a class called `System.ServiceModel.OperationContext` that is useable during the execution of the implementation methods to obtain the execution and message context for the current method. This class can provide the executing method with information about the call, like a session ID, the incoming message headers in the SOAP envelope, information about the identity of the caller, and how the authorization of the call was executed. The context contains a lot of information that is nonfunctional and therefore should not be part of the parameters of the `DataContract` of the method being called. When working in a duplex communication mode, the OperationContext provides the channel to be used to call back to the client during the execution of the call. WCF 3.5 also supports an additional context called WebOperationContext that provides a method with more information about the request in terms of Http properties.

**Concurrent Contracts**

WCF supports concurrent contracts by implementing multiple interfaces simultaneously combined with allowing a service to have multiple endpoints configured. Each endpoint configuration has its own address and is referred to one of the interfaces. This results in the possibility for a single service implementation to implement a set of operations coming from multiple interfaces in which some operations are reserved for one particular interface in combination with operations that are defined in multiple interfaces. The list of operations that is accessible by an interface is defined by the endpoint configuration as this configuration is aware of the interface it’s exposing.

**Data Confidentiality**

Security and data confidentiality is implemented extremely well in WCF. WCF supports cross-platform security by using the WS-* protocols stack for message level security and can additionally use transport security on a higher level in addition to the security applied through these WS-* enhancements. The level of security is defined by the selected binding.

**Atomic Service Transactions**

WCF supports transactions by implementing the WS-Atomic Transaction (WS-AT) protocol. Most often transactions on the service level lead to transactions on the database level. WCF communicates with the distributed transaction coordinator of the database to set up the transactions. The WS-Atomic Transaction specifications define mechanisms to execute transactions across service boundaries. It allows transactions to span multiple calls from a client so the transaction is distributed beyond the facade of the session. This makes it possible for a client to start the transaction, call different operations on the service, and consider all the operation calls as one transaction. The transaction flows from client to service and the operation in the service participates in (sometimes expressed as “follows”) the transaction started by the client.
This means that the service keeps the transaction alive until the client decides the transaction is complete. When this happens, the service signals to the distributed transaction coordinator to commit the transactions.

Distributing transactions also means that when something goes wrong at the service level during one of the calls instantiated by the client, the transaction is considered ended and thus rolled back.

How a service behaves concerning transactions can be defined by the `[TransactionFlow]` attributes on the operation in the interface and by `[OperationBehavior]` attributes on the implementation.

**Session Facade**

The session facade in WCF is built up by the implementation classes. These classes form the first wall of code a client reaches when executing an operation in a service. It’s up to these implementation classes to call the business logic in other layers of the application. This could be the business logic, a workflow, or the data access layer.

**Exception Shielding**

WCF automatically shields away all exceptions occurred in or behind the facade and does not show the details of these exceptions to the calling clients. When exceptions occur at service side, the client is only informed about the fact that something went wrong. By default the detailed information about the error, the inner exception and stack trace, are not visible for the client. This client only sees a very generic error. Receiving this exception causes the channel of the client to come into a faulted state.

This is not only done from a security perspective — you would not want to give information about the stack trace to a potential hacker, for example. But also because sending the details of a .NET exception to a client in a SOA architecture would be useless as the clients are not always .NET clients. As SOA has this cross-platform approach, it’s possible that the clients are not running in the .NET environment but possibly in a J2EE or other framework where receiving a .NET exception would be useless and could not be interpreted.

WCF supports a more cross-platform way of carrying information about exceptions in a SOAP fault. Transporting a SOAP fault and serializing/deserializing the content is defined by the SOAP standard and is cross-platform and more SOA-oriented.

Using SOAP faults in WCF is done by defining a data contract like any other `DataContract` used by the interface. This data contract can have a customized structure containing only the data about the exception and what the service wants to reveal and thinks is useful to its clients and the end user. The operations that will send this custom fault contract need to know this contract by adding the `FaultContract` attribute to the operation.

The methods in which the .NET exception takes place need to catch this exception and translate it to this `DataContract`. It’s up to the client to react to this exception.
COMMUNICATION AND INTEGRATION PATTERNS

Communication and integration are the most important aspects of WCF. These are the areas of software development where WCF really shines and where it has great value — what WCF is really built for. Developers can take advantage of a rich set of features in WCF that support them by building distributed applications that communicate data and create integrations between each other.

WCF is an acronym for Windows Communication Foundation, meaning the core WCF is really targeted to communication. Here you have the background of the need for integration, the different ways communication is used, and the patterns of how messages are exchanged for integration are described.

The need for communication and integration comes from the fact that enterprises are typically comprised out of a large number of different applications. Each of these have their own functionality, built in different architectural styles, using different technologies, and implemented on different operating systems. It’s not uncommon that enterprises have a broad set of applications, either custom-made or bought as a package running on different platforms and locations with each fulfilling only a small part of the business requirements. Solid integration infrastructure is needed as well as a platform-agnostic way of communicating that provides an efficient, controllable, and maintainable exchange of data between multiple applications.

All communication leading to the exchange of data is based on sending messages. This data is either a set of parameters for a question to obtain information or is information that is the answer to these questions. These messages are formatted and serialized according to an industry-wide standard. This standard is SOAP, which is based on XML. The purpose of formatting messages in the SOAP standard is interoperability. SOAP is understood by many technology stacks used for communication and integration. SOAP and the additional WS standards support not only cross-platform formatting but also additional functionalities such as security, reliability, and transactional behavior.

Communication and integration is the technical part of applications that allow functions in a distributed environment to talk to each other and that know how to exchange the data they need. In almost every environment, lots of smaller applications or services, each executing a single responsibility, form a much bigger application.

In many cases applications are distributed by nature. Services are implemented close to the database, applications are run on desktops, and data has to be moved between multiple branches of a company. In between them, the data has to be enriched, aggregated, or somewhere a decision has to be made about where data is to be delivered. Most often data is sent dynamically.

Integration Styles

Throughout the history of software architecture, a number of integration styles were introduced and each of them was based on the needs and requirements for that time — the availability of technology stacks and the standardization of protocols and existence of the communication infrastructure.

We recognize four styles:

- Import/export files
- Shared database
Remote procedure calls
Message Bus

Import/Export Files

At first, when no communication infrastructure was present, the need for integration was low. There were not many applications to integrate and all environments had a closed architecture — the only possible integration was to export data into a file from one application and import the file containing the data into another application. The file could be a delimited or positional flat file.

This was originally a manual process where people could export and move the files on disk or use a FTP connection to a place where the receiving application could reach the file again to import it by a manual action. This process was done later in a more automated way using scripts run by an operating system or custom application that could execute macros in the application and automate a FTP connection.

A lot of these applications still exist in many enterprises keeping this first wave of integration styles alive. It’s clear that this integration is a pure peer-to-peer integration and that the process doing the integration is strongly coupled to the application at both ends of the communication channel.

This style is always communicating data in a delayed fashion where imports and exports were scheduled during night. This was okay at those times but nowadays, a faster, real-life integration is needed in our set of applications.

Shared Database

Another style of integration was built on top of a shared database. Instead of exporting data to files, the data was copied to a central database form where it could be picked up by other applications having access to the same database. This was possible as the access to databases was standardized and most databases were open and reachable by middleware (think ODBC) on different platforms. Integration in this style also became more focused on the multiuser aspect of applications where different applications were accessing this shared data simultaneously and data was exchanged nearly immediately instead of by scheduled imports and exports.

This style also resulted in a coupling between applications and a relation structure somewhere in a database. Changing the data structure for one of the integrations could impact the correctness of other integrations. It uses the same structures in the database for their integration needs.

Remote Procedure Calls

Styles evolved and it became clear that applications should be able to talk to each other without the need for import/export procedures and that a shared database was hard to maintain in large integration scenarios. So the remote procedure call style was taking over.

In a RPC (remote procedure call), style applications can send data to other applications by simply calling their methods and crossing memory and physical network boundaries. Some technologies such as CORBA were created for this style of integration. But a problem with these first-generation RPC technologies is that their implementation is somewhat proprietary and not available on every platform.

SOAP later became a solution, not only in the SOA area but also by providing its value in the integration area. But again RPC, even with an open technology, has its drawbacks. The fear of
coupling applications too tightly with RPC calls is present. Another drawback is the creation of lots of connections between multiple applications where every new integration results in creation code at the client side and code to receive it on the other side.

**Message Bus**

The last style is the Service (or Message) Bus. In this style a new infrastructure is introduced as a bus where messages can be sent by an application and picked up again by other applications. This could be done by publish and subscribe patterns. Data is published onto the bus and other applications subscribe to data on the bus. Even when no subscribers are available the bus can take the initiative itself to resend the data later. On the other hand, the bus can also act as a reader to read external data and resend it to the subscribers.

Publishing data and subscribing to data is based on schemas declared by data contracts. Most often an XSD file is used to not only declare the structure of messages but also as the unit of subscription. Subscribing in a Service Bus is done by expressing interest in a certain namespace or the root element name defined in XSD scheme. All data flowing into the bus in a XML message conforms to this schema and is automatically pushed toward the receiver. The advantage of the bus is that new applications interested in receiving data already published into the bus for another integration scenario can simply be plugged into the bus without the need to change the code of the sender and redeploy the client application.

The main reason to use a bus is if the data is already published on the bus for another application, the new application can also profit from this. It could subscribe to the same data. Even when the data that is already being sent into the bus does not perfectly fit the needs, because the receiving application needs less or possibly other data, it is possible to work with this approach. Data can be filtered and or aggregated with other data flowing into the bus to form a new stream of data onto which the subscriber can plug in to also consume this data.

A Service Bus is there to allow other integrations in the future without impact on already existing integrations.

---

**INTEGRATION OF LEGACY APPLICATIONS WITH A NEW ENVIRONMENT**

Legacy applications are software written years ago, using technology present at that time. It was not meant to be re-engineered or thrown away when new requirements arrived, because the investment would be too large to migrate them to new technologies.

They tend to have a closed and monolithic approach, sometimes tightly coupled to the dataset or user interface. Most often these types of applications are stable, doing their job perfectly, and are the core of the enterprise.

When new requirements are needed, these applications remain unchanged but need to integrate with the new applications or services built for these requirements.

Integration is needed to exchange data—the legacy application exports data and the new service needs this data to fulfill the business requirement.
Message Exchange Patterns

Message exchange patterns (sometimes called MEPs) are descriptions of common ways to have two parts: communication and message exchange.

Request-Response

The most commonly used pattern. The client requests information from the service by sending a request message and expects a response message from the service. This means the service has to be reachable and the client waits for the answer for a defined timeout. The client has a clear definition of what the structure of the answer will be. In case of an exception occurring while processing the request, a message containing the fault is sent. See Figure 1-5.

In most cases the request-response pattern is used to organize communication between applications. Almost every communication has a client requesting data from a service which sends a response. WCF supports this by simply allowing the developer to define a method in C# or VB.NET where the methods have one or more input parameters and can have a return type. These parameters can be of any type, such as primitive strings and integers or more complex types defined in classes.

The approach of WCF is that these method signatures can be defined in an interface, which is a C# or VB.NET language element. This interface has the signatures of the methods together with attributes to indicate that the method is reachable from a remote client. By having the operating class implement this interface, WCF supports the separation of the metadata indicating it’s an operation from the actual implementation. The classes are just implementing the interface, they are not aware that they are called by a remote client. These attributes are [ServiceContract] for the interface and [OperationContract] for the method signatures.

The attributes on the interface indicate that the methods are implementing the request-response pattern. The attributes have additional parameters to further define the details of the request and the response, like the namespace for the contract and the SOAP action and reply action values.

One-Way

Messages are sent from client to server in one direction. The client sends the message but does not expect an answer back. In many scenarios it even forgets about sending it. The server simply processes the message and does not calculate an answer or send acknowledgements back to the client. It’s clear that not all communication can be done by this pattern. Whenever an answer is needed directly, it’s not useable. See Figure 1-6.
The biggest value of this pattern is the ability to work asynchronously, allowing the client to continue its work while the receiver processes the message at its own pace. This results in the illusion that communication is very fast as the client is responsive again immediately after sending the request, but it’s up to the service to process the request. Processing at the server side can be done in a multithreaded approach so the service can consume multiple messages simultaneously.

Another, perhaps most important, value is the ability to create a guaranteed delivery environment. By using a one-way pattern it’s possible to send the message through a queuing system such as MSMQ. A queue at the client side receives the message and tries to send it to the receiver. When the receiver is not available for one reason or another, this mechanism resends the message and delivers it when the other end is back online. The queuing system can also be configured to persist the data when delivery is not possible so the messages survive a reboot.

Having a method executing in the one-way pattern is defined partly at the level of the operation itself by indicating that the return type is void or defining the method as a subroutine instead of a function in VB.NET.

This has to be done in combination with specifying the use of the one-way pattern on the interface by setting the IsOneWay parameter to True on the OperationContract attributes.

### Duplex Messaging

In the duplex pattern, client and server exchange messages without a predefined pattern. The client sends an initial request to the service waiting for a response but allows the service that sends the response to call back to the client to ask for more information. The service can call back to the client dynamically and possibly multiple times before sending the response to the initial request. See Figure 1-7.

In this pattern the client becomes a service for the code while executing. The code which was considered as the service can now act as a client to call a function during the processing of the call.

A distinct communication mechanism is used when a duplex communication calls back to a client during execution. It also needs the signatures of the operation that is called by service to reach the client available in both service and client. First the service needs to use a duplex-enabled way of communicating; there is no way for a service to call back to the client without this.

This is done by selecting one of the duplex bindings supported by WCF and indicating this in the configuration. The signatures of the methods that are implemented in the client need to be defined at service level and also be present at the client to have them implemented.

WCF allows defining a callback contract as a normal interface and indicates at the service interface that it could call one of its methods in the callback interface. This is done by specifying the type of callback interface at the [Service Contract] of the service.
In this way, the definition of the callback contract is part of the WSDL file. The client can implement a class based on this interface. The service knows exactly what methods it can call back to the client as it knows the signatures of the methods available on the client.

Duplex simply means that the service is operating temporarily as a client and needs to have a channel back to reach the client. During the execution of a method the service has access to a context which has a reference to this callback channel. Calling methods on this channel result in calling methods on the client.

Streaming

In the streaming pattern the client initiates a request to get a very large set of data. The service chunks up the data in smaller pieces and sends them to the client one by one. The data is so big that the service has to read this data from a file system or database in chunks. The data chunks are sent to the client in an ordered way as the receiver wants to consume them in that same ordered way. This would be the case in streaming video.

In this pattern there is one request asking for the data followed by a large set of answers which each contain a subset of all the data as a result of the call. It is up to the sender to indicate in the last message that the datastream is finished so the client doesn’t expect more data to come. See Figure 1-8.

Pub-Sub

Pub-sub is the pattern where you see applications publishing data on a regular basis and a number of other applications interested in this data and subscribing to this publication. For example, a stock quote system that publishes quotes frequently is received by interested parties.

In the pub-sub pattern the publisher is typically unaware who the subscribers are and what they are doing with the received data. The publisher doesn’t even care whether there are subscribers. The publisher just cares about sending data out of its system. The pattern indicates an event-driven approach; the publisher raises an event into the client applications. These client applications act to this event by consuming the received data. This consumption can be different: they can show the information to the end user, they can store it in their persistence environment, or they can decide to send the data to other applications. See Figure 1-9.

In the pub-sub pattern there is always an explicit action, called the subscription, where an application expresses interest in the information from the publisher. Most often a subscriber is not interested in all the information the publisher sends. By subscribing, it can express a condition for the data as a filter. The publisher keeps a list of references to the subscribers together with the
filter conditions, and every time it has new information to be sent it iterates through all the interested subscribers, checks interest in the information, and publishes the data.

In many cases the information provider itself does not do this subscription management and broadcasting of the data. Instead the provider sends its data to a generic service which is responsible for the publication of the data. This additional service manages subscriptions and can publish the received data to all subscribers.

WCF 4.0 has no direct support for the pub-sub pattern. The pub-sub pattern can be implemented by using the same construction for the duplex pattern. The implementation of pub-sub could be made by having a subscription service that maintains a list of references to the callback channels of the subscribers in memory.

When the service needs to send data to all subscribers, it can iterate through all these subscribers and call a method to broadcast this data. There is no polling involved in this scenario as the subscribers are acting as a host waiting for methods to be called by services.

.NET services (as part of the Azure AppFabric) has support for the pub-sub pattern without using duplex communication but by using a communication mechanism implemented by the eventRelay bindings.

Clients can register their own address to the .NET Service Bus in the cloud. The AppFabric records where the client is. When data needs to be published to all subscribers, it’s only a matter of sending a message to the Service Bus. The Service Bus relays this message to all subscribers as it knows all subscriber addresses.

**Implied Order Invocation**

When using multiple operations in a contract in some sequential order to fulfill one logical unit of work at the service, it’s useful to define the order that operations can be called. By defining an implied order invocation the client is aware of this order, and this results in the fact that some operations cannot be executed before others are executed first.

This could be the case when operations rely on a state that must be created by operations called earlier. Implied order invocation allows describing which operations can be used to start this logical group of calls and which method could violate this logical order when called.

It also allows defining methods that are considered last in the logical order. Calling these terminating methods indicates to the client that there is no use calling the other methods as the logical unit of work is considered finished.
This pattern is not a pure message exchange pattern as the calls to the methods participating in the ordered invocation can have their own implementation of another pattern. It’s more a pattern that groups multiple operations into one logical unit of work. See Figure 1-10.

WCF provides the `OperationContract` attribute where you can specify the constraints for invoking operations in order. It allows defining a method that can be used to start the conversation by specifying the `IsInitiating` parameter as false. Specifying the `IsTerminating` as true means that calling this method indicates that the conversation is over.

By default, when the parameters are not specified explicitly, every method has `IsInitiating` set to true and every method has `IsTerminating` set to false. This means every method is allowed to be called first and no method terminates the conversation. It’s up to you to decide otherwise and specify which methods cannot be used as the first (`IsInitiating = false`) and which methods will close the conversation (`IsTerminating = false`).

The important aspect here is that the client also knows this order invocation as it’s defined on interface level. So the proxy will not call methods that it’s not allowed to call. Either because it’s calling a method as the first, which is not initiating, or calling a method after it has called a terminating method.

Another way to declare an order in the invocation of methods in a service is by implementing them as a workflow service. This has a more flexible model that defines the order. A workflow service is a declarative definition of a sequence of operations in the form of a flowchart.

In this definition you can clearly define the order of the methods. You can define an expected number of methods out of the interface that are called in a certain step in the process by placing them inside a listen activity. A listen activity groups operations together and stops the process until one of the methods is called. This call executes the workflow continuously with the rest of the process, meaning the other methods in the same listen activity are no longer available.

This definition of the order is not known by the client. This would be too complex for the proxy and is useless as the workflow can behave very dynamically. It could branch to another set of methods in a listen activity based on input from a previous call. When a client calls a method that is not currently available, the client receives an error message indicating that the requested operation is not found.

In workflow services, operations can be defined as either instantiating a new process or participating in existing and already running processes. Typically, the first operation in the workflow instantiates the process, and the other methods follow.
Messaging Patterns

Messaging patterns can be used in a variety of ways, from enriching content to routing messages to intercepting filters and more. Their versatility makes use of the WCF runtime architecture. The following sections will explore messaging patterns in more detail.

WCF Runtime Architecture

To dive into more messaging patterns, you first have to know how the WCF runtime architecture looks. This runtime has a layered architecture where messages travel from the client application down to a stream transporting the message. This message travels up to the implementation of the service on the other side.

Meanwhile the message is passed by the proxy layer on the client and the dispatcher layer on the service. This multilayered runtime architecture allows different messaging patterns to be implemented. It offers lots of extensibility to hook custom code into places to add logic or to change the behavior when calling operations.

Simply put, the proxy translates calls to methods into WCF messages, sends them through the channel, and the dispatcher picks up the WCF messages from the channel and translates them into calls to the code. See Figure 1-11.

In this architecture you see a proxy that is pushing the message on the channel. When the message is on the channel it is passing other layers such as the protocol layer and the encoder layer before it’s actually sent on the wire to the services. At the service side, the message is received by a corresponding channel stack that sends it through the corresponding layers upward. It passes the encoder and protocol layer and is received by the dispatcher. This dispatcher examines the message and figures out which methods to call.

Content Enricher

The content enricher is a pattern which operates on a functional level. In cases where applications send data which does not contain all information that the receiver is expecting, a content enricher is needed.

The reason for this lack of information can be historical — maybe at the time of writing there was no need to have that information as part of messages as it was always available in the database. But in a distributed environment, not all services have access to the database. Between sender and
receiver, another service needs to pick up the message, examine it, and add the needed information to it. Looking up that extra information can be done by querying a database, executing business rules, or simply hard coding it.

Enriching data is something that an implementation of a service operation executes. Enriching the content could potentially mean that the datacontract is changing dynamically. The behavior to enrich data could result in adding an extra property to the datacontract and giving this new property some content and then sending the changed datacontract further to other services.

To do this in a generic approach means WCF needs a way to receive messages for which the datacontract is not defined in the signature of the operation and needs to send the message further in a datacontract that is also not known by the code. Most often content enrichers are just generic functions receiving data in a generic format (XML), changing this format and sending this data further to another service.

WCF supports receiving untyped data by the message class. A function receives one parameter of type message and answers with a message to the call or will not send something back in case of a one-way void method. The message class is a kind of catchall datacontract. In an operation with the message class as inputparameter, every message can be received that is not already mapped to another operation. So creating a generic content enricher is done by creating a service that has one operation with the message parameter.

Of course this code does not have direct typed access to the datacontract as it does not know the details of it. But this code can read the data on the XML level and change the content or add XML nodes where needed.

**Message Routing**

In many cases the sender of a message does not always know where to send the message. The actual receiver is not known by the client so a routing mechanism determines the real address of the endpoint. The route can perform different kinds of processing during the routing.

This decision can be influenced by the content of the message in combination with a set of business rules. Or the actual endpoint can just be hardcoded. In all cases the idea is to decouple the senders from receivers across the organization in a flexible way.

When the actual endpoint is determined by a value or combination of multiple values in the content of this message, it is called *content-based routing*.

It’s often a business requirement that data be sent to one of possible multiple receivers which is interested in this data and can fulfill the request while other receivers cannot.

Since .NET 4.0, WCF has great support for implementing message routing. It supports this by allowing the host of a service to use a class which has no functional behavior but has only the purpose of routing messages to another service somewhere else that needs implementation.

This class can implement interfaces in the `System.ServiceModel.Routing` namespace to define the message exchange patterns needed. This router service determines what the target service is by the evaluation of the incoming message against a set of message filters.
This routing service is hosted and configured as any other service. This means it has multiple endpoints defining an address: binding and a contract. You can configure this service by adding a service behavior that manages the routing table. This routing table is a list of references to client endpoints in the same configuration together with a list of references to filters. So you configure the exact reason for the routing based on the filter criteria. These filters can work on the WS-addressing values to determine the route, work on endpoint names, or use an XPath expression to evaluate an incoming message.

This routing mechanism also supports protocol bridging by simply specifying the needed bindings in the configuration of the target endpoints referenced by the routing table.

**Protocol Bridging**

When messages flowing into the router are formatted in a protocol which cannot be understood by possible receivers, it is the router’s responsibility to switch to the needed protocol.

This is done by reading the data out of the message in the source protocol and copying the data into a message of the destination protocol. Bridging can be done by reformatting data out of a closed proprietary format to XML, according to a schema. But also on the transport level, bridging can switch from one transport protocol to another.

The router functionality of WCF 4.0 can also act as a bridge between protocols. The use of a certain protocol is defined by the binding configured at an endpoint together with the address to reach the endpoint. To change the protocol and send the message with the request to another endpoint with a different binding, the routing system in WCF can be used. WCF automatically translates the protocol.

**Message Filters**

Message filters are the key to configuring message routing. They allow expressing conditions in a declarative way to support the routing mechanism that makes decisions about where to route incoming messages. With a message filter you can declare that for some SOAP WS-addressing actions the message must be routed to other services. So it’s possible to have a dedicated service for each SOAP action. Not only can message filters be based on SOAP actions but also on other information identified for the request. XPath queries looking into the content of the message or to the SOAP headers are also useful as a message filter.

**Backup Endpoint**

Routing can also be useful for implementing a backup endpoint. In this case you have a routing table configured with multiple endpoints which are not related to any message filter or used for protocol bridging.

But the routing table defines which would be an alternative endpoint if the first endpoint is not available. This provides an escape route if the targeted service going down the routing mechanism decides to send the message to an alternative route.
Multicast Behavior

With this behavior, the routing mechanism delivers the message to multiple receivers. This can be done by overlapping message filters. The incoming message is compared to all filters and is sent to the endpoint for each matching filter found. It’s clear that this multicast behavior is only applicable for the one-way MEP, as routing the question to more than one receiver could result in multiple answers to the client.

Discovery

Discovery can be used when the runtime location of services is dynamically changing as new services join the network and others disappear. Discovery can also be used when the implementation of the hosting of service is not under the control of the client and their network administrators.

When using discovery, clients can look up the needed address themselves by sending a probe for service endpoints on the network. This probe knows what the needed criteria are for evaluation. Finding a match results in the discovery of the service and its network location in the form of an URI. Most often these criteria are the contract. This means the contents of the WSDL for a service is used to look up a service which can fulfill the required contract.

When the address of the endpoint is figured out, the client can talk to this service as it has the correct contract.

Besides this ad-hoc way of working where a polling broadcast is sent across the network to find the correct endpoint, there is a more network-friendly way for services starting up to announce themselves to possible listeners.

WCF implements the SOAP-based WS-Discovery protocol and can make a service discoverable by configuring the behavior for the services. This results in the service being discoverable over UDP on the local subnet. Clients can use the WS-Discovery protocol to find out the actual address of the running service.

Clients can do this by using the DiscoveryClient class. This class provides a method that specifies the criteria to find a service on the network. This will most often be the type of interface the client wants to use to talk to the service. After a corresponding service is found, the client has access to the address and can use it to specify the address of the real proxy that is used for the conversation. Clients can narrow the results by specifying a more detailed scope. Services can also announce whether they are active and have joined the network. This signal can be heard by clients that implement the WS-Discovery announcement protocol. To do this the client has to instantiate the AnnouncementService which has events to detect when services get online or go down. WCF also provides a discovery proxy to build a central service as a hub between multiple services and multiple clients.

Intercepting Filters

Using the intercepting filters pattern allows you to add logic that executes right after receiving the call, and just before the operation in a service is executed. It could also be immediately after the end of the execution and before the answer is sent back to the client.

The logic intercepts the call and does some preprocessing and/or post-processing. In this processing you have the opportunity to check authentication, decide on creating a session, decode data, start
transactions if needed, and provide defaults for missing data sent by clients with older versions of a
data contract.

The purpose is to centralize and reuse this processing logic for multiple operation calls in a
service and to not have the technical code for this behavior mixed with the functional code of the
implementation of the service operation. With the use of intercepting filters, which are executed at
server side, neither client nor services are coupled to this technical logic. Neither client nor services
are aware of the processing logic being executed.

WCF supports intercepting filters implemented in the \texttt{System.ServiceModel.Dispatcher}
namespace. This namespace contains classes and interfaces to modify the runtime execution of the
WCF application at the dispatcher level. The dispatcher gets the message from the channel stack and
calls the methods. This execution can be influenced by custom code.

This code can intercept the call and executes the desired behavior before the implementation if the
operation is executed. There are extensibility points to execute parameter inspections, transform
messages, set up output caching, authorize, and perform message logging and many other patterns.

Creating an extensibility point is most often a matter of creating a class that implements one or
more of the interfaces found in the dispatcher namespace and writes custom code implementing the
interfaces.

\textbf{Context Objects}

The context object pattern describes that there is data available with technical details of the call and
the service instance during the execution of the call. This means that the implementation cannot
only rely on the incoming functional parameters but that it also has access to an object, called the
context, where more technical data is stored.

This data is about the call in general or the running instance of the service. By accessing the object
context the operation has access to information about the associated channel referring back to the
client, possible security information about the user calling the operation, header information in the
message, session ID, and so on.

\textbf{BUSINESS PROCESS PATTERNS}

When trying to describe a business process in a written or schematized form, some recurring
scenarios are discovered. These typical scenarios can be considered common patterns which are
applicable in lots of scenarios and are present in nearly all business processes.

\textbf{Process Manager}

This is the high-level pattern on business processes. It describes the infrastructure and concepts
needed to execute the defined steps in business processes, allows applications to participate in the
process, exchanges data with the process, and reaches the goal of the process.

It’s clear that processes need to be managed. This means there is an environment that runs the
processes, allows starting instances of a process, deals with exceptions, stores the state of a process,
and can put the process into an idle mode.
Processes are a declaration of a list of steps to be undertaken to fulfill a goal. This goal gathers information, makes decisions on the information, and distributes the information.

The start of a process is triggered by some event in the business. For example: a process has to be started when a new employee starts working at the company. The goal of the process is to fulfill a number of registrations in different applications, assign a new employee to a manager, give the new employee access to intranet applications, and allow the employee to order a car.

A number of actions occur when the employee starts working. In some cases the actions have to occur in a predefined order where it’s not possible to continue before the previous step is finished.

The execution of the process is also influenced by the availability of information and is not able to continue before this information is known to the process. Allowing the process to wait for information before continuing is the most important responsibility of the process manager.

In the declaration of the process it is indicated that more information is needed. When the process detects this situation, figures out what information is needed, and from what source it will come, the process manager puts the process in an idle state and persists all information already known by the process.

When reaching this waiting state the process and all the data it is using is released from memory. The process manager persists this to a database and a pointer to the place where the process stopped is remembered. The process manager is then responsible for listening for incoming information.

When new information arrives from the needed source, the process manager checks the content of the information and selects the correct instance of the persisted processes. It then loads it into memory again, fills the process with all the data it had before it came to the waiting state, and restarts the process from the point where it was left.

In this way it is like the process was never stopped, only delayed while waiting for a piece of data.

The process manager is thus responsible for starting processes, persisting them when needed, waiting for incoming data, and restarting the processes.

The process manager must also act when a process does not get the needed information in a timeframe defined by the process. This means the process manager must not only look for incoming data, it also must frequently check the database with persisted processes to figure out if the idle situation of an instance of the process hasn’t timed out.

**Start-Listen-Action Pattern**

In its simplest form, every process has three stages. It gets started by the process manager and triggered by an application. After this the process comes in the listen state and the process manager does its work to persist the process, and waits for new information before continuing.

When the data arrives, the process manager reactivates the process and the process takes action to process this new information. The pattern can be present in a more complex form where decisions influence where to listen.

You could say that if there is no start-listen-action pattern to be recognized in your requirements, it’s not really a process that needs to be managed by a process manager. See Figure 1-12.
Patterns in Workflows Declaration

To define the declaration of the process you need a typical combination of activities. This combination can also be seen as patterns.

Sequence

One step (considered an operation) is executed after the previous one. It’s the basic and fundamental pattern for organizing processes. It enables a top-down approach where a process starts and continues by doing the next defined step in a sequence.

Parallel Split

In a parallel split the process is split into more than one branch. Each branch runs simultaneously and does its work independently of the other branch. In a parallel split there is no way to know which one of the branches will be completed first. It’s possible that one branch has only a small step that executes fast and a second branch that waits for human interactions. The finishing time of the second branch is not predictable.

Synchronization

This pattern is often found together with the parallel split. A parallel split has a wait-for-everybody-to-complete-before-continuing approach. Every branch does its work and the steps in the sequence after the split are only executed when all branches report that they have finished their job.
This parallel looks like the multithreading features found in an operating system but in fact it is the process manager who does the scheduling of the steps in all branches.

Listen Pattern

Also called the exclusive choice. In this pattern a set of branches is declared of which only one is executed during the process. The pattern is based on a mechanism that can decide which one of the branches needs to be executed.

This mechanism is either a delay that times out or an incoming message with the information the process instance was waiting for. In the listen pattern a few conditions are described that can become true while the process is in an idle state. Being in idle state means that the process cannot decide for itself what to do next. It will wait for a signal before continuing.

As the content of the signal is important for this decision, multiple branches are described in the listen pattern. Every branch has this condition as the first element and has a sequence of steps just after this condition. When the signal comes in, all conditions are evaluated. When a match is found for the branch the process continues with the steps defined in that branch. It only executes the step defined in its branch; the other branches for which the condition is not satisfied are discarded and the steps in these branches are never executed.

Listen for Incoming Messages during a Predefined Time

In all situations where the listen pattern is used there will be at least one branch which defines a delay. This indicated process will listen for the defined condition fulfilled by incoming messages for only a specified amount of time. If this timeout is reached before any condition is met, the branch for the delay is executed. The branch will most often execute a backup scenario informing the logging system that the needed trigger, with possible answers for the condition of the other branches, has not been executed.

With this branch, the process can take action and reminds other applications and humans using them to take action. The process can decide to continue in the listen shape and wait again for the input needed to execute one of the other actions.

Convoy Patterns

In a convoy pattern, multiple messages are related to each other and must all be received by the process to achieve a result. This indicates that the process does not have all the information it needs just by receiving one message. In convoys there is always a first message that starts the convoy and the expectation by the process to receive other related messages. The process waits for more messages which are related to the first one and can only continue after all information is received. The combination of all the messages is what the process is looking for. Either it needs multiple messages to work with the set as one big piece of related data or it needs at least a number of messages to make a decision from different sources to continue. The data to fulfill the request is split into pieces which are given to the process one by one.

The reason behind this chunking could be technical, with some kind of memory limit enforcing us to split data into multiple parts. But the reason can also be functional because the input for the process is just an aggregation of data coming from multiple applications which can only send
the data they own themselves. Aggregation data from various sources is exactly why business processes are needed.

Here are some examples:

- A process fulfills an order and receives multiple messages. It contains order lines because the sending application cannot send them with the order because of technical limitations.
- A process needs a supplier and waits until at least three price quotes are received before deciding on a supplier.

**Sequential Convoy**

In a sequential convoy, different messages have a predefined order. The process needs multiple messages for a result but can only receive one message at a time. This is the message it is expecting according to the place in the sequence of the previous one. This could be the case if data in the messages is needed to make a decision about what messages to expect next. If the content of the messages is indicating some kind of priority needed for executing, a sequential convoy is needed.

**Parallel Convoy**

In a parallel convey, messages do not have a strict order. The process can consume messages in any order. But as in the parallel pattern, the process only continues after it has received all messages of the convoy. In a parallel convoy either the number of messages to be expected is known before the messages start coming in or a property of the messages indicates whether it’s the last message in the convoy or not.