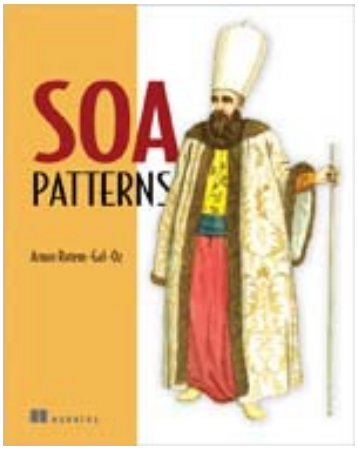
# *Basic Structural Patterns*

Excerpted from




## SOA Patterns

EARLY ACCESS EDITION

**Arnon Rotem-Gal-Oz**
MEAP Release: June 2007
Softbound print: February 2009 (est.) | 250 pages
ISBN: 1-933988-26-6

This article is taken from the book *SOA Patterns*. Service Oriented Architecture (SOA) has become the leading solution for complex, connected business systems. While it's easy to grasp the theory of SOA, implementing well-designed, practical SOA systems can be a difficult challenge. *SOA Patterns* provides detailed, technology-neutral solutions to the challenges by providing architectural guidance through patterns and anti-patterns. This article focuses on five patterns that address matters of services.

Congratulations, you are now in charge of building your first service – now what? The first thing to do, before getting into the fancy stuff like making service secure, scalable, connecting it to a UI and the like, is to take care of the basic concerns about our service – where will it live ? How can it be reliable? How to make it autonomous? Etc.

Simply put, this chapter deals with some basic patterns. Patterns that solve some of the more common issues related to services. These are the patterns you are most likely to use even if you have modest requirements for your service. In Chapter I we talked about the SOA basics- creating autonomous components that publish and accept messages defined in contracts, delivered at an endpoint(s) and governed by policies to service consumers. Dealing with fundamental issues, the patterns in this chapter are relevant to implementing the Services themselves (See Figure 2.1 below). Patterns in other chapters will cover issues that relate to the other SOA components

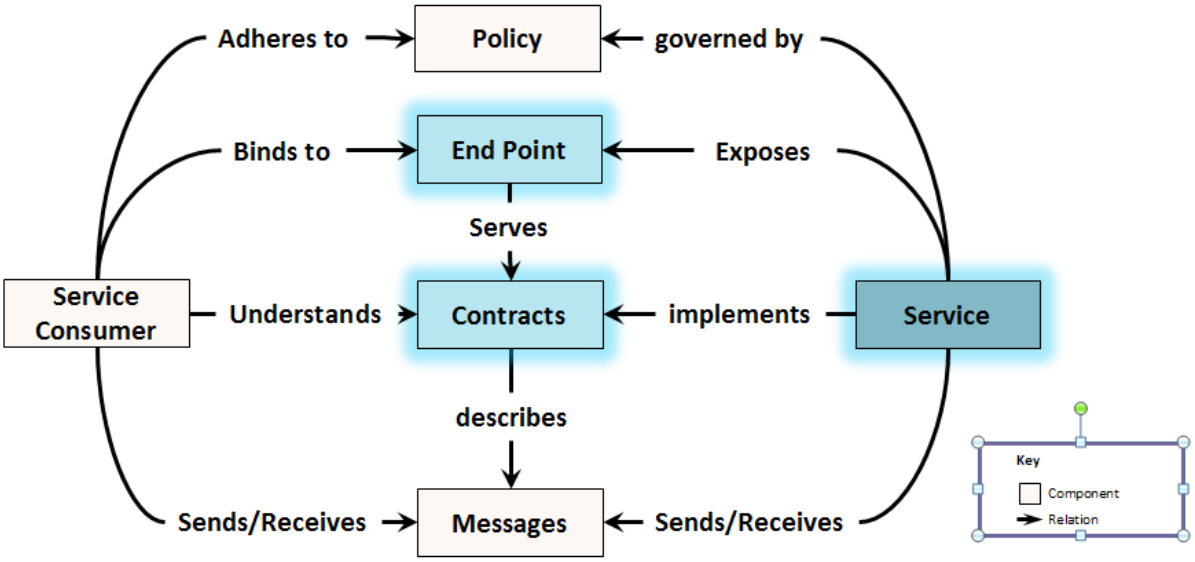



**Figure 2.1 SOA defines six different components. Chapter 2 has patterns that deal with only one of them the services, which is of course the essence of SOA.**

There are five patterns in this chapter; Table 2.1 lists all of them along with the problems they address.

| Pattern name | Problem address |
|---|---|
| Service Host | How can you make your services adaptable to different configurations easily and save yourself the repetitive and mundane tasks of setting listeners, wiring components etc. |
| Active Service | How can I increase service autonomy and handle temporal concerns? |
| Transactional Service | How to handle messages reliably? |
| Workflodize | How to increase the service's adaptability to changing business processes ? |
| Edge Component | How to isolate the business functionality of the service, from the non-related cross-cutting concerns like security, logging etc.. |

**Table 2.1 list of patterns**

Ok, enough introduction already, let's bring on some patterns

## *Service Host*

The first pattern we are going to talk about is one of the most basic patterns if not the most basic one. The ServiceHost pattern deals with the environment which runs the services instances and handles some of rote tasks associated with that.

### *The Problem*

Pick a service, any service (don't tell me what it is ☺). Wait , ok, I see something, You have some code that set up listeners for incoming messages or requests. You have some code to wire-up components, you have some code that initialize and activates the service. You probably also have some code to configure your service Well? How did I do? Chances are you have most of if not all these pieces of code somewhere in your service.

There are quite a few tasks that are repetitive and common. Maybe there is something we can do about it.

> **How can you make your services adaptable to different configurations easily and save yourself the repetitive and mundane tasks of setting listeners, wiring components etc.**

The first option, or actually non-option, is to rewrite the wiring and the rest of the repetitive code for each and every service. Obviously, this is not a good option - for one you are likely to introduce bugs as you write something again and again. The multiple copies of the same code problem is even worse when we consider maintaining this code. During maintenance you not only need to make bug fixes and changes for each service but also ,make sure that you didn't miss any of them and all the services are up to date.

A more plausible option is to create a library of common tasks and have each service work with the library's API. This helps a bit, but you are still left with coding the wiring that is needed to utilize all of the library's functionality.

Another option is using inheritance, i.e. to create a superclass which implements the common functionality and have each service inherit from that class. Using inheritance is also problematic since the service functionality usually won't fit within a single class, due to the granularity of what a service is. Additionally different services handle completely different business areas – otherwise they would be the same service, so it usually doesn't make sense to make them a single class hierarchy.

Using inheritance almost solves our problem as we write the code once and the customization only occurs where the services differ – if we want to get the same behavior without using inheritance we need to use a framework.

### *The Solution*

**Create a common ServiceHost component that acts as a container or a framework for services. The container is configurable and performs the wiring and setup of services.**

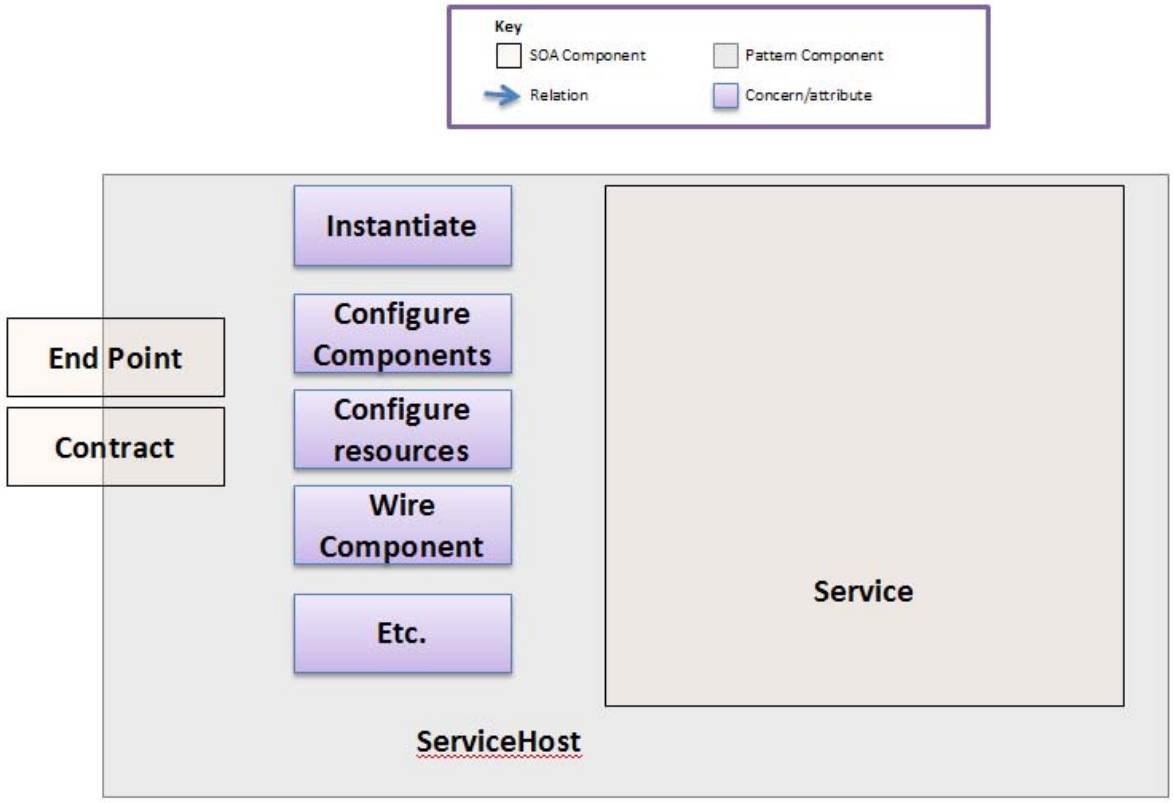



**Figure 2.2: ServiceHost Pattern. ServiceHost is a container that runs the service and performs the wiring and configuration on the service's behalf**

The ServiceHost acts as a mini-framework which can have several responsibilities. The first responsibility is to instantiate the right components or classes that make that the service consists of. The ServiceHost is also responsible to read configuration information for instance, the ServiceHost can read the port where consumers can use to contact the service. Another responsibility is to set up the the service's environment for example setting up listeners on the endpoint. Lastly, the ServiceHost can be responsible for wiring the components for instance binding a protocol to listener to an endpoint or setting up a connection to the database. The common denominator of these responsibilities is that they are all related to instantiating and initializing services and as we've seen in the problem introduction you are likely to encounter them in more than one service. As was mentioned above, the ServiceHost is a framework, which is a little different from the library concept mentioned in option two. basically a collection of utility classes/methods your code can call upon to get some functionality. Frameworks, on the other hand, contain some functionality or flow and calls your code to extend or make the flow a specific one. The principle of frameworks calling your code is known as "Inversion of Control." It is in wide use in object oriented frameworks such as Spring or Spring.Net, picocontainers and the like.

The the ServiceHost pattern has several benefits compared with the other options mentioned above. One benefit was already mentioned – as a framework the ServiceHost performs the work and only calls your code to fine tune the behavior rather than leaving this orchestration to you. Another benefit is that it better addresses the Open Closed Principle (OCP). OCP states that a class should be open to extension but closed for modification which is exactly what we get when we use the framework concept.

A ServiceHost may host more than one service – though this is probably not very common, but there was one system I built where we had to scale a solution down to run on single computer and this was handy. The more likely situation is that a service would span more than one computer and then you would have several ServiceHost instances each hosting here parts of the service rather than the complete service

The ServiceHost pattern is commonly used by technology vendors – we can see that in the technology mapping section below.

### *Technology Mapping*

As mentioned in the patterns structure in Chapter 1, the technology mapping section takes a brief look at what does it mean to implement the pattern using existing technologies as well as mention places technologies implement the pattern.

The SeriveHost is a fundamental SOA structural pattern and as such it is supported by most available technologies. Both JAX-WS and Windows Communication Foundation let you configure your services in markup (XML) and have the web server i.e. the servlet engine or IIS, do most of the wiring for you.

Windows Communication Foundation also has a class called ServiceHost. Microsoft's documentation describes the ServicHost as follows: "Use the ServiceHost class to configure and expose a service for use by client applications when you are not using Internet Information Services (IIS) or Windows Activation Services (WAS) to expose a service. Both IIS and WAS interact with a ServiceHost object on your behalf". Thus, basically the built-in WCF implementation of a ServiceHost is very much in line with the pattern described here.

If you are implementing the ServiceHost pattern by yourself perhaps on top of the technology one, you can take a look at lightweight container like Sprint (or Spring.NET), picocontainers etc. to help you out with wiring and instantiation. There aren't many other technological implications as the ServiceHost pattern is a relatively simple pattern.

#### LIGHTWEIGHT CONTAINERS AND DEPENDENCY INJECTION

Spring and a few other frameworks are known as "light weight containers". The nice thing about these "Light weight containers" is that they allow you to increase loose coupling and testability of your solutions. They perform this magic through the use of the Dependency Injection pattern which is a non-SOA pattern. Dependency Injection, as the name implies, happens when a class lets a 3rd party which acts as an "assembler", provide it all . the interfaces it depends upon. Using Dependency Injection a class no longer has to depend on a specific implementation but rather it only depend on the interface or abstract class. This help increase testability as you can now supply stubs or mocks for the class to simulate its environment. It also helps flexibility as you can easily change implementation of the dependencies as long as they keep their contract

As we've seen The ServiceHost pattern is simple but effective and it is in wide use. You can take a look at the Further reading section at the end of the chapter for links to resources that expand on the technologies mentioned in this section.

### *Quality Attribute Scenarios*

The quality attribute scenarios section talks about the architectural benefits of utilizing patterns from the requirements perspective. As was mentioned in chapter 1, most of the architectural requirement are described from the perspective of quality attributes (scalability, flexibility, performance et.) – through the use of scenarios where these attributes are manifested. The scenarios can also be used as reference for situations where the pattern is applicable.

The main reason to use the ServiceHost pattern is reusability. Reusability is increased by the fact that common tasks that are needed by many services are only written once. A nice side-effect of reusability is also increased reliability as you also only need to debug once. The other quality attribute the ServiceHost pattern provides is portability. Portability is enhanced by the separation of concern effect of the pattern – as was demonstrated in the scale-down example I mentioned above. Another facet of portability comes from the ability to configure the service context in markup.

Table 2.2 below shows a couple of sample scenarios that can point you to consider the ServiceHost pattern.

**Table 2.2 ServiceHost pattern quality attributes scenarios. The architectural scenarios that can make us think about using the Edge Component pattern.**

| Quality Attribute (level1) | Quality Attribute (level2) | Sample Scenario |
|---|---|---|
| Reusability | Reduce Development Time | During development, set the environment for a new service within 20 minutes. |
| Portability | Installation | The system has to support configuration of server per service as well as grid scenarios. During installation, switching from one environment to another should take less than an hour |

Once you have a service up and running you need to think whether the service would be passive and only wake-up when a request arrive or should the service be active and perform some chores such as publish its state, handle timeouts without waiting for its consumers to activate it. As its name implies, the Active Service pattern allows a service to be active rather than passive.

## *Active Service*

As Chapter 1 explain it is important for services to be Autonomous. Autonomy increases the loose coupling between services and spells greater flexibility for the overall solution. But what does autonomous services actually mean? One explanation I've heard was the autonomous means that the teams working on different services can be autonomous. This definition means that there are few dependencies between the services as they only know each other by contract. This means that different teams can work independently each focusing on its own service without stepping on each other's tows.

While, this is a nice “feature” to have, a much more valuable (as in “business value”) definition is that the services are as self sufficient as possible. Let’s explain this using an example.

### *The Problem*

Imagine a journal subscription agency like Ebsco or Blackwell, which needs to create a proposal for a client. The proposal service needs, among other things, to produce a “pro forma” invoice. In order to produce the pro forma, the service must get both the discount the business gives the customer as well as the discounts the business get from the different publishers so that for example, we can check if the proposal is profitable. Figure 2.3 below shows a simple example for such a flow.

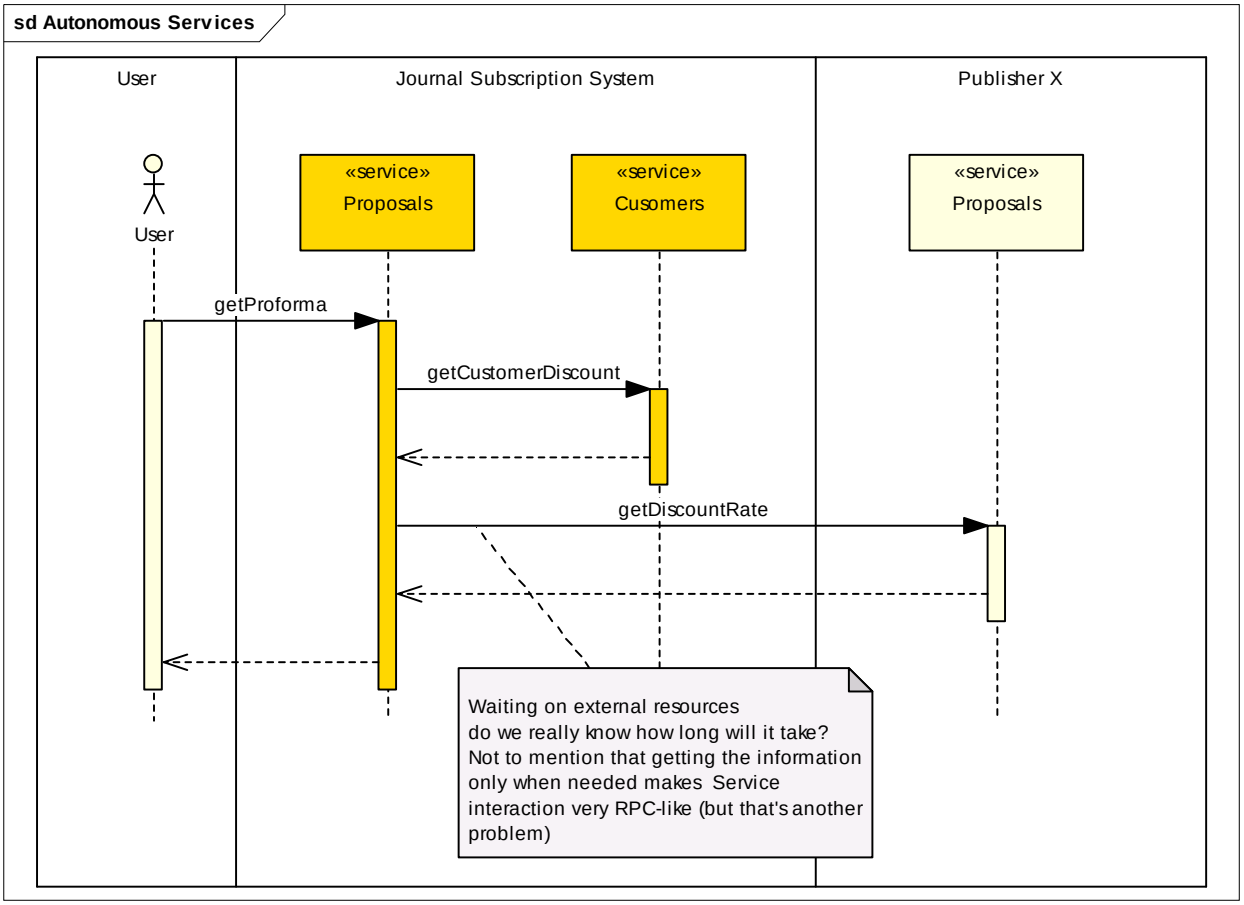



**Figure 2.3. A common inter-service interaction. The proposal service needs to get discount rates from both a customer service which is internal and a publisher’s service which is external. The proposal service needs both these service to be on-line or it can’t function properly.**

In the sample scenario the proposal service has to wait for two other services and while the customer service is internal and is part of the same system, the publisher’s discount service is most probably an external once – what will happen to our proposal service if the publisher’s system is not on-line? The proposal service would be unavailable. Oops, even if we spent gazillion dollars on making sure the proposal service was fault-tolerant we now encounter a situation where it is unavailable. The reason for that is that the proposal service is coupled in time to the publisher service which is external. The proposal service is not really autonomous.

**How can I increase service autonomy and handle temporal concerns?**

As the example above demonstrated a passive service which only works upon request is problematic since the service might not be able to fulfill its contract (or its SLA) depending on how other external services behave.

One option is for the service to cache previous results but this will provides a partial solution as it doesn't take care of data freshness and that occasionally you'd have a cache miss and will need to contact other services anyway. Another problem with this approach when you have a lot of incoming requests and you are "busy waiting" on the thread that handled the request on your service, is that you can get into a resources problem when, as each of these requests is now waiting for external input.

Even if we manage to solve the caching problem mentioned in the previous paragraph we still need to be able to solve other temporal events. Temporal events are recurring or one-time events that are tied to time. For instance producing monthly bills or publishing stock figures or any other recurring reports are all temporal events. One option to solve this is to choreograph the service from the outside (see Orchestrated Choreography pattern) the problem with this approach is that you are externalizing business logic that is really the service responsibility. Remember that encapsulating business aspects is one of the reasons to go with SOA in the first place. We need to have another way to achieve this.

### *The Solution*

**Make the Service an Active Service by implementing at least one active class ,either on the edge, the service or both. Let the active class take care of temporal concerns and autonomy issues**

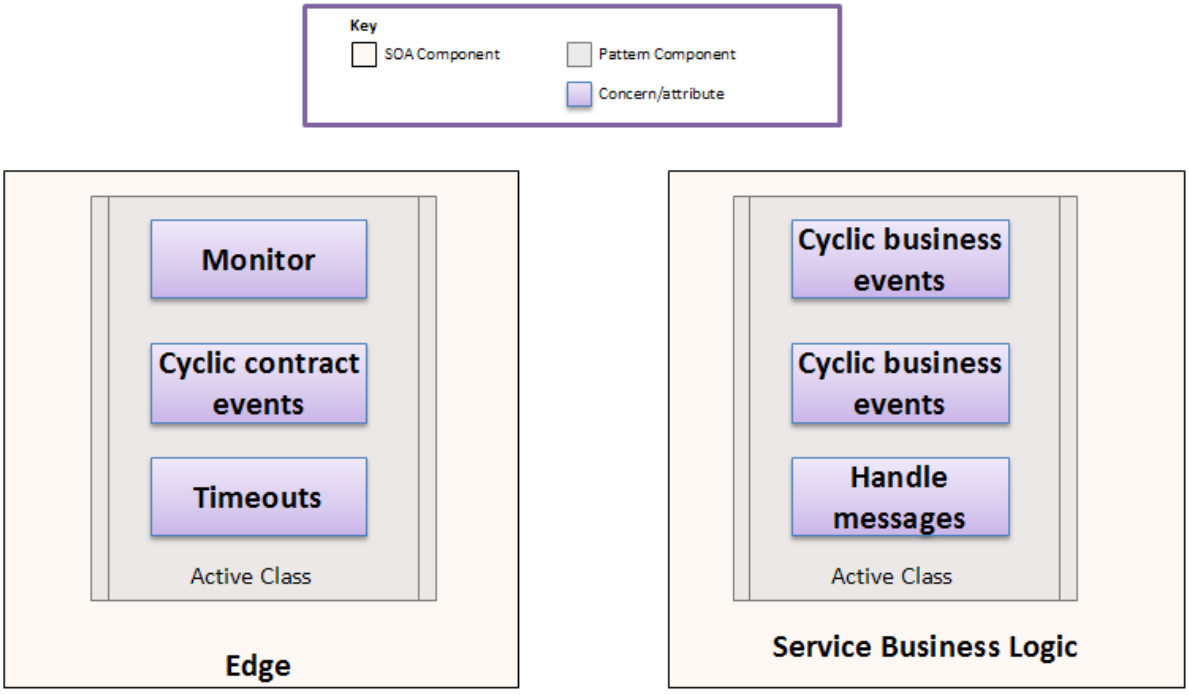



**Figure 2.4 Active Service Pattern. Active Service means to add independent behavior to a service to that it would have its own thread of control. This can be used to handle cyclic events, timeouts, monitoring etc.**

The Active Service pattern basically means implementing the "Active Class" concept on the service level. "Active Class" as defined in the Official UML specification means "An object that may execute its own behavior without requiring method invocation". The same can be applied to services. This means

that the service would have independent threads that are used to handle cyclic events such as monthly billing or publishing status etc. An Active service can also monitor its own health (see Blogjecting Watchdog pattern in chapter 3), handle timeouts it can even be used to handle requests (see for example the decoupled invocation pattern in chapter 3).

So, how can an Active Service pattern help us solve the problems mentioned above? Well, as Pat Morita playing Mr. Miagi said in "Karate kid" – "best defense – no be there" – if you want to avoid waiting for other service, your best defense is not to get to that situation in the first place– you can actively, periodically, go and fetch data from other services to refresh your caches. You can also save other services the trouble and proactively publish your own state changes (see Inversion of Communication pattern). Caching data locally can seem to induce a data duplication problem but it doesn't (see callout for details).

**CACHING AND THE DATA DUPLICATION PROBLEM**

I am sure that several of you, especially those of you with database background, who read the suggestion to actively go and fetch and cache data from remote services, jumped out of your seats and questioned my sanity for promoting data duplication. However, the way I see it this is not really the same data. The data that is cached in a service is that service view of the data. It can be aggregated, processed or otherwise altered to fit the service needs. Naturally, one caveat you must have in mind is that the service that caches the data is not the master of the data.

Also note that caching is only good for specific relatively stable data. If you need aggregation in larger scale take a look at the Entity Aggregate and Aggregated Reporting patterns (in chapter 7)

A thread with a timer can take care of most of the other temporal events (you can have a timer per event if you have few events or wake up every known interval, see what events needs to be handled and process them if you have a lot of possible events).

A thread in the Edge Component(s) is a good way to deal with contract related temporal issues (e.g. make sure we publish state on time, timeouts etc.), while a thread on the service takes care on purely business related concerns like sending monthly bills or handle incoming messages queue (see decoupled invocation pattern)

Let's look at how the situation shown in Figure 2.3 can be redesigned using the Active Service pattern. To recap, Figure 2.3 shows a flow for a proposals service which needs to get external data from an internal customer service and an external publisher service to produce a pro-forma for a customer. Consider Figure 2.5 below. Now the Proposals service actively goes to fetch discounts on a regular basis and caches the results, thus when a request to produce a pro-forma arrives the proposal service can immediately calculate the discount and return a reply more quickly, plus it doesn't depend on the external services to be there when the request for producing the pro-forma arrives. Using Active Service, the coupling between the proposals service and the other services is decoupled in time.

By the way, an alternate approach to solve this problem can be achieved using the Inversion of Communication pattern (see chapter 6).

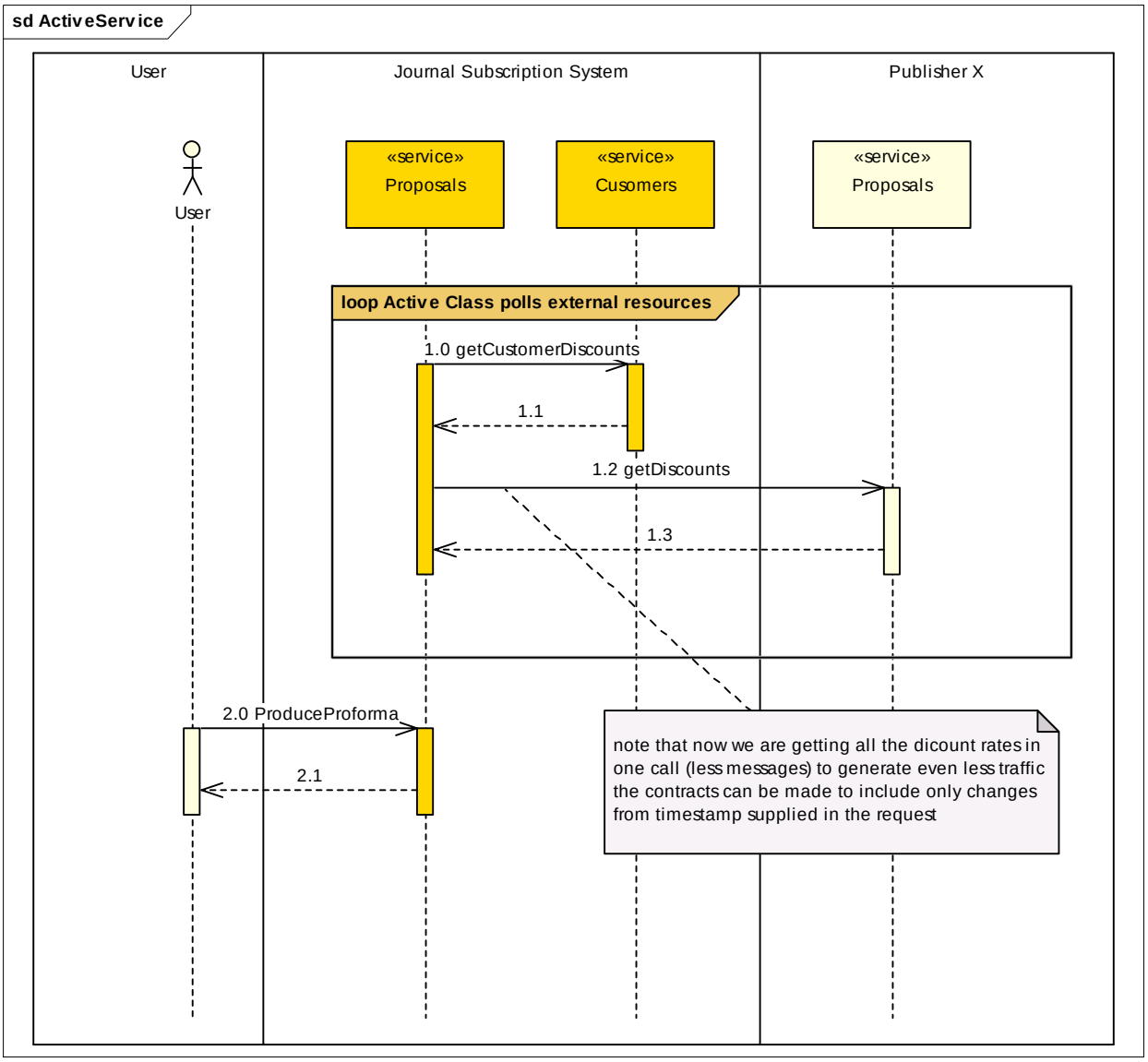



**Figure 2.5 Service interaction refactored to using the ActiveService pattern. The proposal's service actively polls the other services for the information it needs, thus when a request arrives it is able to return a reply more quickly and without depending on the other service's availability.**

The Active Service pattern is mostly a mindset pattern and doesn't have a lot of technological implications.

### *Technology Mapping*

The technology mapping section usually talks about how the pattern is implemented using SOA related technology, however since there are aren't real technological challenges in implementing the Active Service pattern this section will be brief.

The technological idea behind the Active Service patterns is simply to have an active thread on the service and/or the Edge Component (see later in this chapter) that will provide some specific functionality which you will have to code. Thus basically, the Active Service pattern relies on threading technologies which is available in any language you may want to use. The real trick is to decide what you want to do with this thread – and the previous section showed some ideas for that like caching data from other services and handling recurring reports.

The next question is when do you want to use the Active Service Pattern – let's take a look at few scenarios that can make us consider this pattern.

### *Quality Attribute Scenarios*

The quality attribute scenarios section talks about the architectural benefits of utilizing patterns from the requirements perspective. As was mentioned in chapter 1, most of the architectural requirement are described from the perspective of quality attributes (scalability, flexibility, performance et.) – through the use of scenarios where these attributes are manifested. The scenarios can also be used as reference for situations where the pattern is applicable.

Active Service is the prerequisite for few other patterns such as decoupled invocation and blogjecting watchdog mentioned above- and those patterns help handle many quality attributes such as reliability, availability etc. Nevertheless, even applied alone the Active Service Pattern helps satisfy several quality attributes.

Active Service helps satisfy latency aspects by allowing data to already be available for the service to consume. It helps with deadlines by making sure that the service will perform its tasks when needed in order to meet deadlines (such as producing monthly bills on time). Another quality attribute that benefits from the Active Service pattern is availability, since polling other services and caching their data means the service's availability is less dependent on other services availability.

Table 2.4 below list a few sample scenarios where the Active Server pattern can help.

| Quality Attribute (level1) | Quality Attribute (level2) | Sample Scenario |
|---|---|---|
| Performance | Latency | Under normal conditions evaluating the profitability of an offer will take no more than 2 seconds |
| Performance | Deadline | Under load and normal conditions the system can update stock prices at least twice a second |
| Availability | Uptime | While disconnected from the WAN, the users can still produce quotations |

**Table 2.3 Active Service pattern quality attributes scenarios The architectural scenarios that can make us consider using the Active Service pattern.**

Another important attribute of service construction is – how do we handle messages once we get them either on the edge component or in the service. The Transactional Service pattern allows for solving this problem while also dealing with reliability problems.

## *Transactional Service*

The nominal scenario of SOA is for a service to get a request to do something from a service consumer, the service then handles the request, maybe asking other services to do some stuff as well, and then produces one or more reactions for the consumer which initiated the request. Figure 2.6 below shows such a nominal scenario in an E-Commerce system. A front-end talks to an ordering service (see Client/Server/Service pattern in chapter 5 for more details on this type of configurations). The ordering service then registers the order, sends the order out to suppliers and notifies a billing service. When everything is done it sends a confirmation to the E-Commerce front-end application.

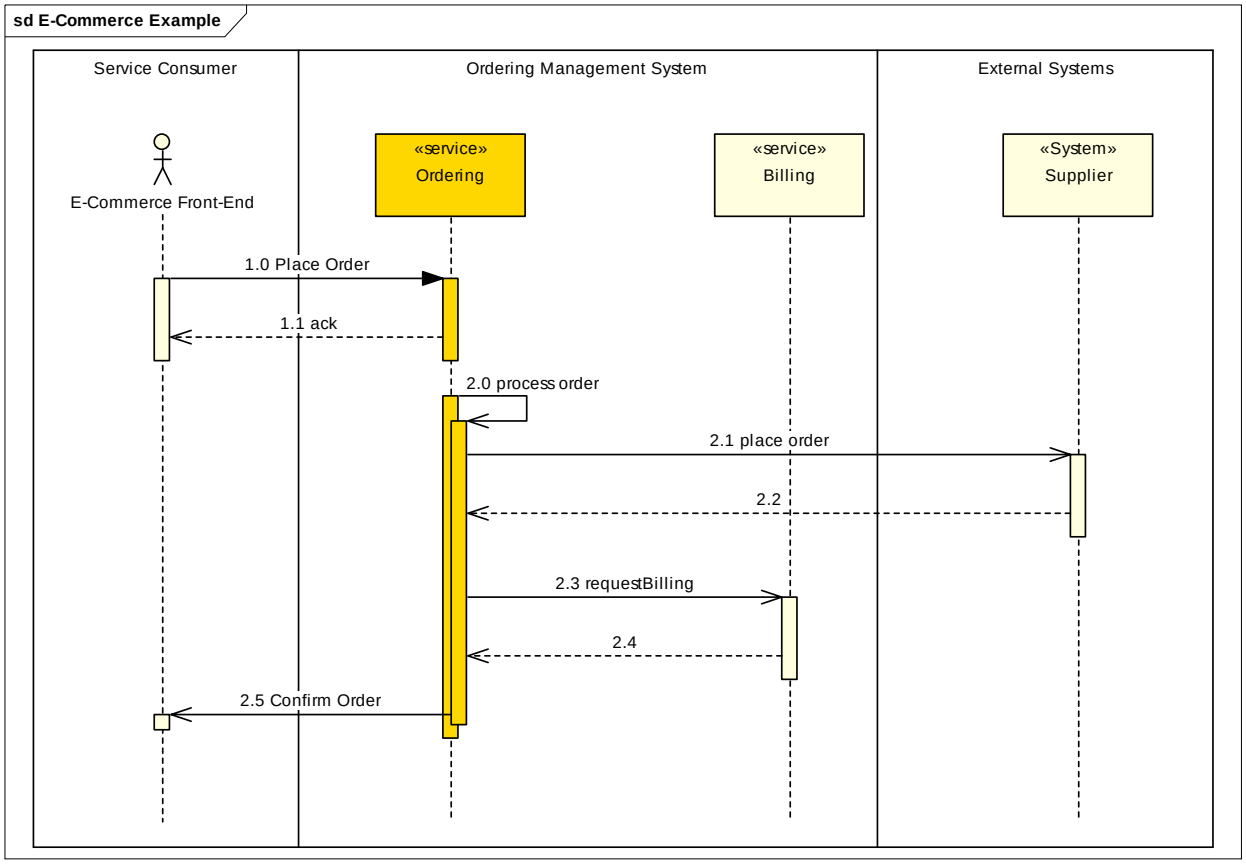



**Figure 2.6 Sample message flow in an E-Commerce scenario (talking to an Ordering Service). The front-end sends an order to an ordering service which then orders the part from a supplier and asks a billing service to produce bill the customer.**

The nominal scenario looks very assuring, but what if something goes wrong?

### *The problem*

Let's consider what might happen if the Ordering Service crashes between acknowledging receipt of the order and processing it – for instance between steps 1.1 and 2.0 in the Figure 2.6. I can imagine our customer sitting comfortably in her favorite sofa, waiting for the postman to deliver whatever it is she ordered. Well, for all I know, she is sitting there still – as the order was lost forever.

What will happen if the service fail just before requesting the billing to process the order -before step 2.3. In this case, the order is lost again - unless the ordering system doesn't wait for the billing, and processes the order, which is unlikely. What's more worrying is that we already placed an order with our suppliers and they have a bill coming as well as merchandize we have to store somewhere.

The handling of messages in services is filled with situations like the ones mentioned above. We can probably comfort ourselves that most of the time things will work just fine, however as Murphy have it – our service will crash eventually and naturally that would happen on that million dollar order. The question is then:

**How can a service handle requests reliably?**

One approach to solve the reliability problem is to push the responsibility to the service consumer. In the scenario above that would mean that if the service consumer doesn't get the order confirmation in step 2.5. The consumer can assume that the order failed. For one this approach is not very robust and decrease

the service's autonomy– the service doesn't have any control over its consumers and they may or may not handle problems. Also it only solves some of the problems – those that have to do with the service consumer. What happens to the interactions the service makes? For instance in the ordering scenario mentioned above – you can still be in trouble if you fail after step 2.1 of sending an order to the supplier.

Another option is to handle messages synchronously. Synchronous operation can prove to be very problematic in terms of performance especially when the service needs to interact with external services, systems or resources – as then the whole process has to wait for the other party to react before the service can return its reply. Even more important is the fact that this doesn't really solve the problem. If the service fails anywhere in the process we don't really know where we are at. The only thing we can know is what message failed and we need the consumer's help for that as.

It seems we can solve the problem, if the service will save its state into a persistent storage such as a database. I think that is a step in the right direction – however we are not safe yet we can still be in trouble if the service crashes just before persisting its state and the incoming message would still be lost without the service knowing. Another aspect we need to note is that using a persistent storage we can track where in the process a failure occurred – but we can't be sure if messages to other services were sent or not

To solve these last issues as well as the whole reliability problem we need the Transactional Service

### *The Solution*

**Apply the transactional Service pattern and handle all the flow from reading the message to sending out responses and reactions in a single transaction**

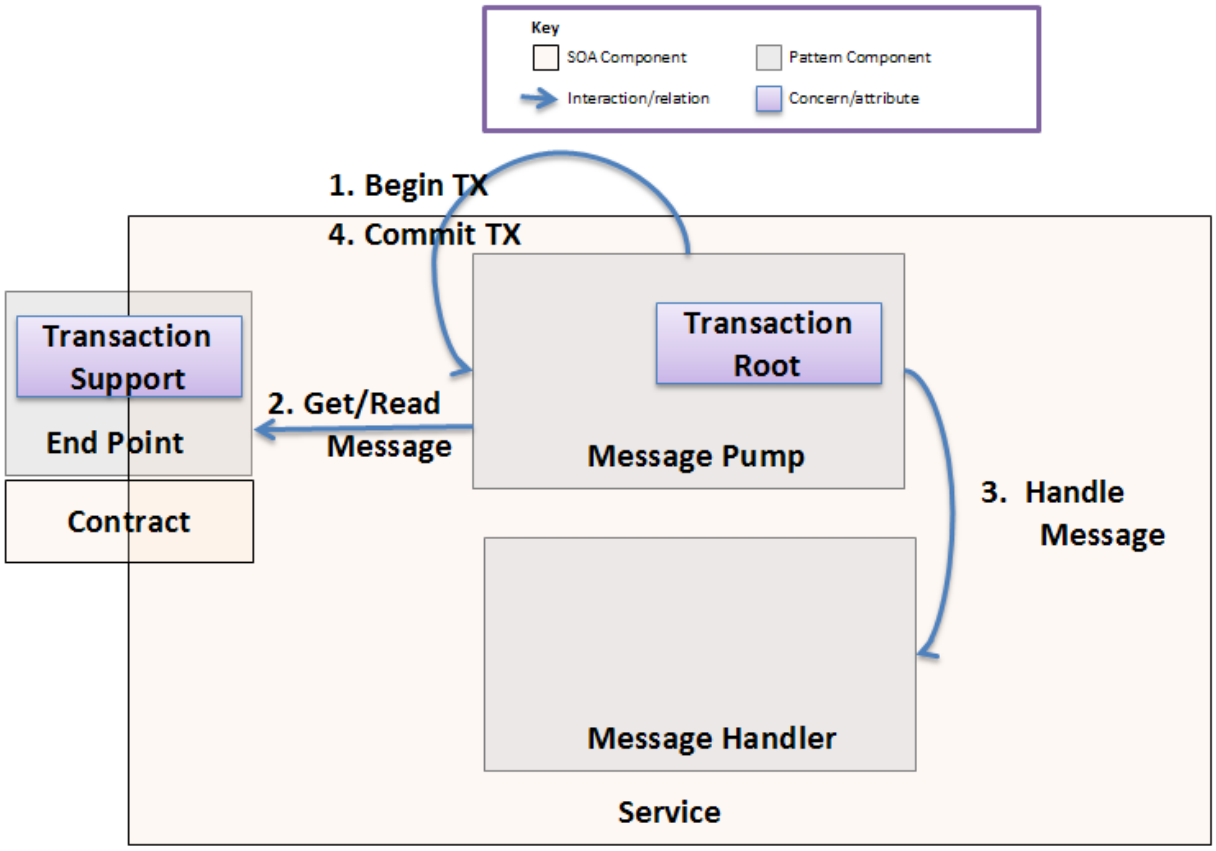



**Figure 2.7 Transactional Service Pattern. The transactional service is requires an endpoint/edge that supports**

**transactions. The flow is to open a transaction, read from the endpoint, handle the message, and close the transaction when finished.**

The main component of the Transactional Service pattern is the message pump. The message pump listens on the endpoint or Edge for incoming messages, when a message arrives the message pump can then begin a transaction, read the message, pass it to some other component/classes to handle the message and when is the processing is finished, wrap the transaction (commit/abort). If it is possible to send replies or requests in a transactional manner they can also participate in the transaction otherwise you will need compensation logic if the transaction aborts.

The advantage of a using a transactional programming model is that is it's all or nothing semantics which means you don't need to deal with edge cases. Due to the ACID properties of transactions all the operations and messages are guaranteed to be either completed or not, so you have a high assurance that if a message left the service, the incoming message that triggered that reaction has been fully handled.

**ACID TRANSACTIONS**

A transaction is a complete unit of work. A unit of work is qualified as a transaction if it demonstrates four tenets that go by the acronym ACID:

A – Atomic – everything that happens in a transaction happens as one atomic unit. Either all the actions happen or they all don't happen.

C – Consistent – The transaction aware resource is left in a consistent state whether the transaction committed or failed and throughout any interim step.

I – Isolated – Any external observers (that don't participate in the transaction) do not see the interim states. The either see the state before the transaction began or the state after it completed.

D – Durable – Changes made in the transaction are saved in persistent storage so that they are available after a system restart.

The tradeoff you are making when you go with the Transactional Service pattern is of course performance. Transactions are always slower than working without them because of the preparation, the IO needed for durability, lock managements etc. What I usually do is define target scenarios and test early to make sure the solution is good enough.

One option to implementing the Transactional Service pattern, is to use a transactional message transport for all the messages that flow between the services. Having a transactional message transport makes implementing the pattern very easy -you just follow the steps mentioned above: begin transaction, read, handle, send, commit. Another option , which I guess is the more common scenario is to put incoming messages into a transactional resource such as a queue or database table upon receiving it and then send an acknowledge as a reply to service consumers. Since the initial message handling in this case is not transactional you need to be able to cope with a message arriving multiple times if the consumer for example didn't get the acknowledge message and sends a request to withdrew 1 million dollars – again (see for example the discussion on idempotent messages in chapter 6).

Figure 2.8 below shows a redesign of the example in Figure 2.6 mentioned above in light of the Transactional Service pattern. To recap, the scenario talks about an e-commerce front-end, which talks to an ordering service. The ordering service then registers the order, sends the order out to suppliers and notifies a billing service. When everything is done it sends a confirmation to the E-Commerce front-end application.

Using the Transactional Service pattern, steps 2.0 to 2.5 in diagram, which are the actions taken by the ordering service itself, are in the same transaction. This means that if you don't handle the place order message because of a crash or other mishap - no message leaves the service. This is a boon since now we don't have to write complicated compensation logic to handle such failures. A subtle issue here is what might happen if the Ordering service crashes somewhere between steps 1.0 and 1.2. The scenario is not 100% fail-safe there's a slight chance that we enqueue the incoming message for processing but then crash before we acknowledge the message. As mentioned above, this may result in accepting the same request more than once. One way to handle reduce these duplicate messages on the service's side is to look at the incoming queue on service startup and send acknowledges for all the messages in the queue in this case the consumer might get more than one acknowledge for messages it sent.

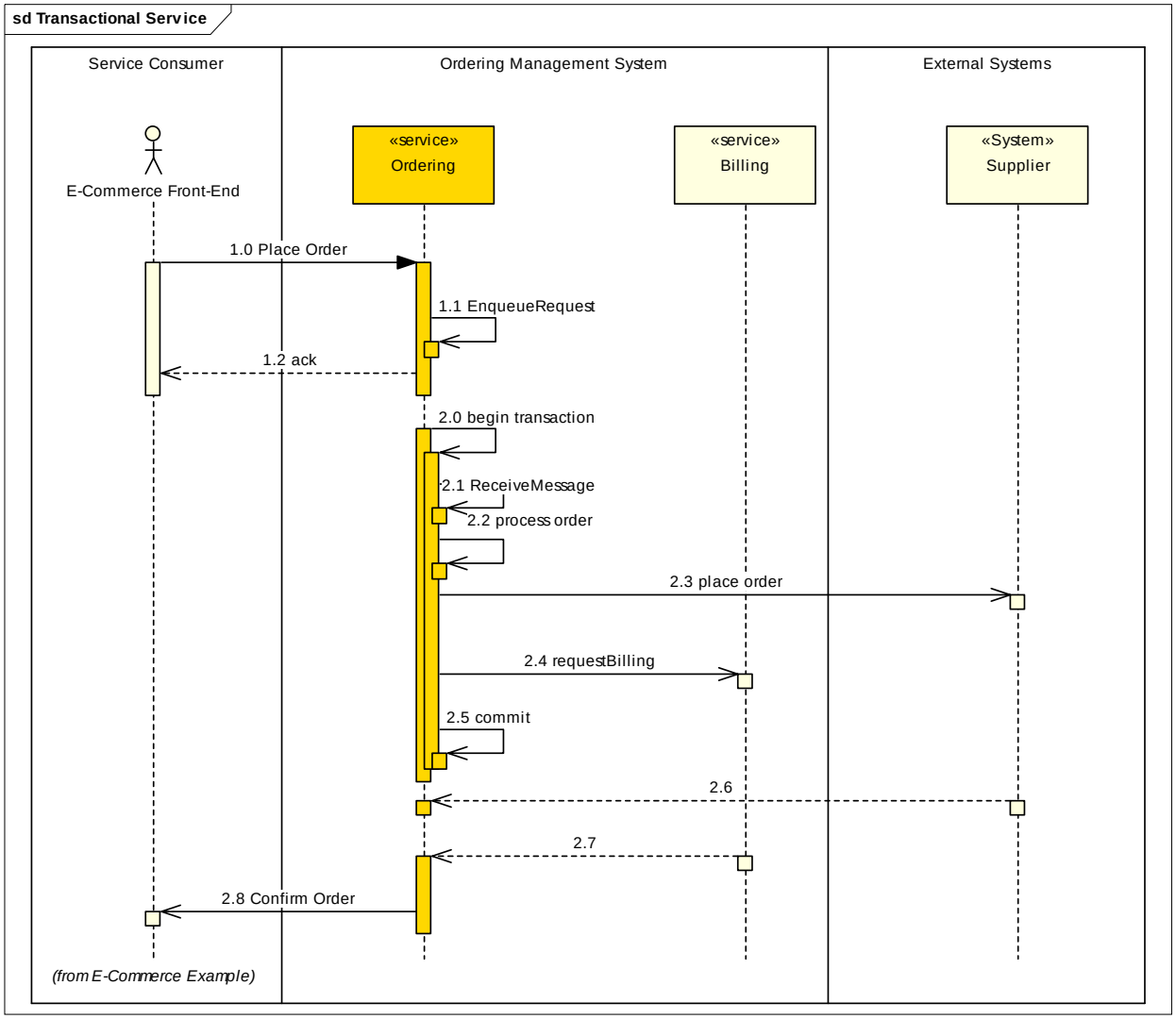



**Figure 2.8 The E-Commerce flow from Figure 2.6 redesigned to use the Transactional Service pattern. All the handling of the Place Order message is done within a single transaction.**

Note that for the example, using a single transaction would work only as long as the billing process just produces an invoice – it won't work if the billing service activity is to process a credit card and the ordering needs the confirmation to continue. W hen a single transaction won't work, the process needs to be broken to smaller transactions and the whole process becomes what's known as long running

operation (see the Saga pattern in chapter 6). Another reason besides long running processes to break the flow into a few smaller transactions is if the service itself is distributed.

It is important to notice the difference between setting the transaction scope from the endpoint/edge onward or setting it from the sender which is external to the service. The difference may not seem important, but it is – since in the former case you increase the reliability of the service and the latter case increases coupling in the system and can cause you a lot of headaches. When you extend a transaction beyond the service boundary you are making a leap of faith since the other service runs on its own machine, it has its own SLA etc. Holding internal resources for something beyond the service trust boundary is risky. You can read a more thorough discussion on the problem in the "Cross-Service Transactions" anti-pattern

Let's look at what's needed to implement a transactional service.

### *Technology Mapping*

As mentioned in the patterns structure in Chapter 1, the technology mapping section takes a brief look at what does it mean to implement the pattern using existing technologies as well as mention places technologies implement the pattern.

Implementing a Transactional Service can be easy if the message transport is transaction aware. Examples for transaction aware infrastructure can be found in most ESBs such as Sonic ESB, Iona Artix etc and also in messaging middleware like MSMQ, any JMS implementation or SQL Server Service Broker. If you are using a transactional message transport you can implement the Transactional Service pattern by just create a transaction on that resource. You may need to make the transaction distributed if, for example, you also perform database updates within the same unit of work. Then you just read a message from the ESB or messaging middleware, process it, send reactions or other messages generated by the process to outgoing or destination queues and commit the transaction if everything was successful.

Note that since you would usually use more than one resource in the transaction for instance a queue for the message and a database to save any state changes after handling that message you will most likely need a distributed transaction. In .NET 2.0 and up you can open a TransactionScope object (defined in System.Transactions) to transparently move to a distributed transaction if needed.

If the message transport doesn't support transactions, only Acknowledge after you've saved the message into some transactional repository such as a queue or a table. You still run the risk of handling a message without acknowledging the service consumer – so you need to be ready for the possibility of getting the request again in case the ack was lost or never sent. In case of a failure, you will also need compensation logic if you sent messages to other services within the transaction that handled the incoming message.

#### POISON MESSAGES

When we read a message in a transactional manner, we need to pay attention to identify and handle poison messages. A poison message is a message that is faulty in some way so that that makes that service crash or always abort the transaction when it is handled. The problem is that if you read the poison message inside a transaction, the crash causes the message to return to the queue, where it nicely waits until your service recuperate enough to read it again and repeat the cycle. Some technology options like messaging products may have a mechanism to identify and discard poison messages. You need to make sure the mechanism is there and that it takes care of all your crash scenarios or at least be aware of this and deal with it yourself.

A technology that seems related is WS-ReliableMessaging. However, despite its name, the protocol is only concerned with delivering the message safely from point to point – in a sense WS-ReliableMessaging is sort of like TCP for HTTP. There is no durability promise or any transactional trait

imbued in the protocol. Many ESBs, which are transactional, support this protocol so you can have the best of both worlds of using a standard protocol and transactional handling of messages

Other related protocols are WS-Coordination and its "siblings" WS-AtomicTransaction and WS-BusinessActivity. I'll look into WS-BusinessActivity in more detail when I'll discuss the Saga pattern in chapter 6. For now let's focus on WS-AtomicTransaction. WS-AtomicTransaction basically defines a protocol to orchestrate a distributed transaction between services – as a general rule I would not recommend using this as it introduces a lot of coupling between services for instance in the scenario in Figure 2.8 above – do we really want to lock resources while we wait for the supplier which is external to our company and may, for example, treat our orders with low priority? You can see the Cross-Service Transactions anti-pattern in chapter 10 for more details.

When you use a transaction-aware middleware the situation is a little different – instead of one transaction that span between services you have three little transactions one that the sending services and the middleware the other the middleware performs internally to guarantee delivery and the last one between the middleware and the reader – this is a coupling to the infrastructure which you can isolate in an Edge Component (see later in this chapter).

As for the other protocol - WS-BusinessActivity, we will look into that in the Technology mapping section of the Saga pattern (chapter 6).

### *Quality Attribute Scenarios*

The quality attribute scenarios section talks about the architectural benefits of utilizing patterns from the requirements perspective. As was mentioned in chapter 1, most of the architectural requirement are described from the perspective of quality attributes (scalability, flexibility, performance et.) – through the use of scenarios where these attributes are manifested. The scenarios can also be used as reference for situations where the pattern is applicable.

The transactional semantics which Transactional Service induces can simplify both coding and testing. Additionally it can greatly enhance the reliability and robustness of the service. The code becomes simple since The promise of "all or nothing" helps keep developers focused on delivering the business value rather than thinking about edge cases and other related distractions.

Here are a couple of examples for scenarios that can point you to looking at the Active Service pattern.

**Table 2.4 Transactional Service pattern quality attributes scenarios The architectural scenarios that can make us consider using the Active Service pattern.**

| Quality Attribute (level1) | Quality Attribute (level2) | Sample Scenario |
|---|---|---|
| Reliability | Data lows | under all conditions, a message acknowledged by the system will not be lost |
| Testability | Coverage rate | get 95% or better test coverage for all scenarios |

The Transactional Service pattern saves us coding because transactions don't have as many edge cases as you have when you write non-transactional code. Another pattern that can save writing code is the Workflodize pattern.

## *Workflodize*

On one of the projects I worked we had to build a sales support system for a mobile operator. It would probably not come as a surprise to you if I told you that the competition between mobile operators is very fierce. The result of this competition is that the operator's marking departments burn the midnight oil trying to come up with new usage plans and bundles to increase their sales. You know, plans such as friends and family, PTT for closed groups, reduced rates for international calls, bundles for 3.5G usage

and whatnot. For this particular operator, new usage plans were created several times a week. The billing system was based on Amdocs and SAP systems took their time to be adjusted to new plans, however, since marketing campaigns were usually initiated by the marketing department regardless of the IT readiness there was a burning need to be able to support new sales procedures ASAP.

Changing business needs is something that is common to many if not all business – it might not be in the same intensity as described above, but it is there. We need to find a way to enable our services to cope with these changing processes

### *The Problem*

**How can we increase the service's adaptability to changing business processes?**

The most obvious option is just to wait for the change requests, and develop the code and update the services every time requirement change. This poses several problems for one you need a full development cycle to make the change happen .Secondly, code changes means big portions of the system need to be retested – think of questions like "does the new change affect the plan we added yesterday?"; "What about the one we've added last week which is similar?" etc.. More development and testers immediately translates to longer time to market. For our example, it means that making the new plan operational will take a few weeks. Which in turn means management would not very happy. Which means your job just went down a notch or two. This is clearly not the way to go.

A better approach would be to try to isolate the more stable parts of the application from the ones that constantly change. For our sample scenario the demographic data like getting the customer's name, address etc. is probably the same regardless of the plan we want to sell. Nevertheless writing the choreography for the stable logic is still a daunting and error-prone task. Maybe we can do even better than that…

### *The Solution*

**Introduce a workflow engine within the service to handle the volatile and changing processes and orchestrate the stable logic**

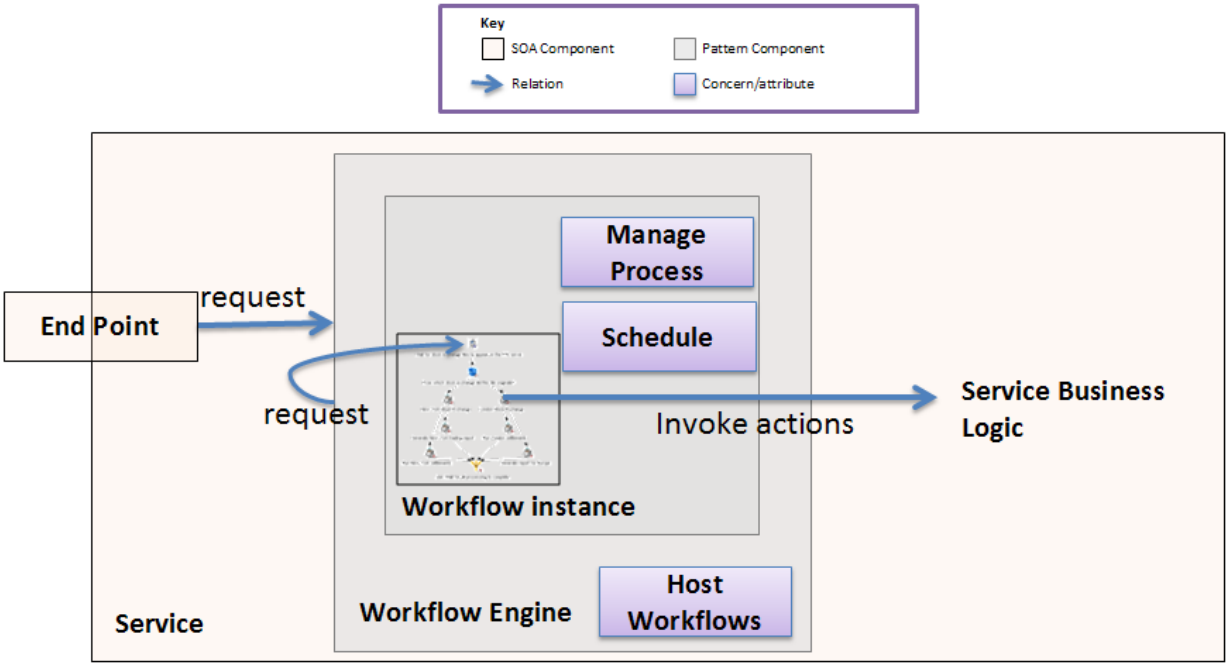



**Figure 2.9 Workflodize Pattern. Within the service a request gets routed to a workflow engine. The engine locates the appropriate workflow instance (or instantiate a new one) and execute the process. The workflow drives the business logic. The business process is made of the individual small building blocks that are relatively easy to rearrange.**

The Workflodize pattern as depicted in figure 2.9 above, is based on adding a Workflow Engine to the service to drive business processes. The Workflow engine hosts instances of workflow. The nominal case is a workflow per request type, however, workflows can also be more complex to handle long running processes and have several entry points, where requests or data arrives from external service( see the Saga pattern in chapter 6 for more details on long running processes).

The advantage of workflows is that it gives you an tool that both makes you think in building blocks called activities and lets you rearrange them into processes in a flexible and easy way. Modeling the processes as a flow of activities means it would be easier to identify the stable ones and reuse them as the change requests arrive. Since the activities can be tested by themselves, reusing an activity means you don't have to retest it as heavily as you would otherwise. The flexibility for rearranging the activities means you can quickly respond changing business needs.

One question that the inviting option to easily change the service behavior (vie workflow) raises is should the contract version be updated every time the behavior changes. The answer is, of course, it depends. My guiding rule is that if the Liskov's Substitution Principle is kept in regard to the way the contract behaves then there's no reason to add a new version.

#### WHEN TO CHANGE CONTRACT VERSIONS - LISKOV'S SUBSTITUTION PRINCIPLE FOR SERVICES

Liskov's Substitution principle, which is also known as design by contract, is an object-oriented principle. Barbara Liskov originally published it as follows: "If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T.". In plain English this means that a subclass is a class that can be **usable** instead of its parent class without breaking the behavior of any user of the base class. Applied to SOA this means that when changing the internal behavior of a service, you don't need to create a new version of the contract if, for each operation defined in the contract the preconditions are the same or weaker and the post-conditions (i.e. the outcome of the request) are the same or stronger. In other words in order to keep the same contract version,

> the new version of the service should meet the expectations that consumers of the service have come to expect from the old service version's observable behavior.

Lets Workflodize the example scenarios mentioned above and see how adding a workflow can help us. To recap, the scenario talks about introducing new usage plans quickly for a mobile operator. When a new plan is introduces the backend systems are usually not ready - it takes a few days or weeks to change, test and deploy them. One thing we can use the workflow for is to route requests for new plans that don't have backend implementations for human intervention. For example we can let some customer service register the change in the CRM and then notify technicians to configure the network etc. Later when the backend systems will be ready we can reroute the flow to them. Furthermore, there are many steps in the flow which are stable, as I already mentioned, getting the customer's demographics (name, address etc.), offering add-on and accessories for phones etc. These steps can be reusable activities or steps in the flow which most, if not all, selling processes reuse. Adding a workflow in this scenario greatly enhanced the business ability to react and remain agile. When one of the competitors launches a new plan which is all the rage, this mobile operator is able to react and launch a competing plan within a day or less. This is real and tangible business value.

The ability to handle long running processes mentioned above is another advantage of workflow engine. Visualizing the complete processes that involve multi-message interaction makes it easier to understand the big picture and how the process unfolds and thus debug the process from the business perspective.

Workflodize can, of course, be combined with other patterns, for example, it is easy to use job scheduling (which most if not all workflow engines support) to implement the Active Service pattern.

A pattern closely related to Workflodize is Orchestrated Choreography; both patterns use the same underlying technology of using a workflow engine. Nevertheless, Even though the technology used is the same there are different architectural considerations that warrant a different pattern, for example, one easily observable difference, is that Workflodize is constrained to stay within the boundaries of a single service and the Orchestrated Choreography has to do with adding workflow coordination between services. Further discussion of the differences between the patterns can be found in chapter 7 "Composition Patterns".

### *Technology Mapping*

As mentioned in the patterns structure in Chapter 1, the technology mapping section takes a brief look at what does it mean to implement the pattern using existing technologies as well as mention places technologies implement the pattern.

The natural technology mapping for the Workflodize pattern is of course workflow engines. There are many workflow engines in the market. Microsoft released Windows Workflow Foundation as part of the .NET 3.0, which I guess will make it a popular choice in the .NET world - though there are several other companies that provide .NET workflow solutions like Skelta or K2. Java, as usual, has more options from the usual suspects, IBM , JBoss etc. as well as companies that specialize in workflows such as flux. Oracle even has a workflow package for its database (WF_Engine) and a supporting Java API

Most Workflow engines have a built-in visual designers for modeling the workflows themselves. Figure 2.10 below shows a modeling of the active Service pattern for report generation using the visual designer of Flux.

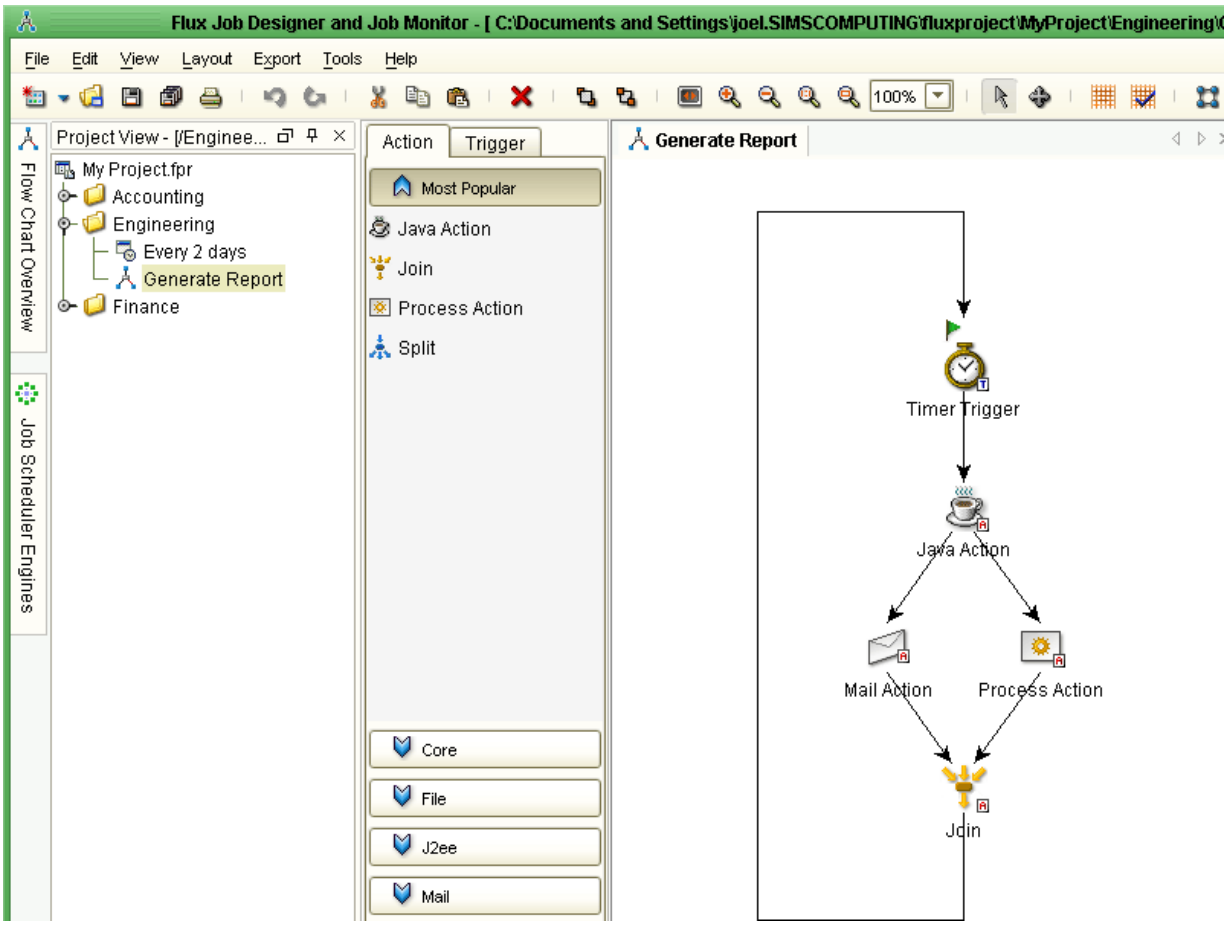



**Figure 2.10 A sample for a Workflow visual designer. Most workflow engines also come with a visual designer. The example above shows an implementation of an Active Service pattern using a workflow to produce scheduled reports**

While using a visual editor such as the one in Figure 2.10 above is usually the preferred option for modeling flows, you can usually also use XML to define the workflows. Several tools, such as the open source (BSD license) OpenWFE don't have a visual editor at all and rely solely on XML for configuring their workflows. Code Excerpt 2.1 below shows an example of a flow modeled in OpenWFE.

**Code excerpt 2.1 : partial XML of a credit approval workflow implemented for OpenWFE**

```
<process-definition name="Credit approval">
        <sequence>
            .
            .
            .
            <participant field-ref="order_value" />
            <if>
              <greater-than  field-value="order_value" other-value="10000" />
              <!-- then -->
                  <sequence>
                     <participant ref=”supervisor”/>
                     <subprocess ref=”ReviewAndApproveOrder”/>
                  </sequence>
              <!-- else -->
                    <subprocess ref="TaskPaypal" />
```

```
            </if>
            .
            .
            .
        </sequence>
</process-definition>
```

**CHOOSING A WORKFLOW ENGINE – THE FLEXIBILITY PERSPECTIVE**

There are few simple building blocks for modeling workflows like activities, exclusive choice (one of possible execution paths), parallel split, etc. You should be aware that there are a few sometimes there are more complex scenarios. For example how to merge many execution paths without synchronizing them – but still execute the subsequent activity only once. Another example is how to handle multiple instances of an activities (what if they need synchronization, what is there is the number of activities are not known in advance, and so on) and quite a few other such problems. Solutions to these problems are defined as workflow patterns (many of them are described in "The Workflow patterns page" which can be found at http://is.tm.tue.nl/research/patterns/patterns.htm).

I suggest, that you also take a look at how many of the workflow patterns the engine supports to ensure you will have the flexibility and not hit a brick wall later in the game. Flexibility, of course, is not the only selection criteria (you still need to consider performance, availability, security, etc.) but I think it is an important one for a tool whose main premise is to introduce flexibility.

Some of the workflow engines such as Microsoft Biztalk or Websphere MQ Workflow are better suited for orchestrating inter-service interaction (see Orchestrated Choreography pattern in chapter 7) and not as an internal workflows due to their costs.

### *Quality Attribute Scenarios*

The quality attribute scenarios section talks about the architectural benefits of utilizing patterns from the requirements perspective. As was mentioned in chapter 1, most of the architectural requirement are described from the perspective of quality attributes (scalability, flexibility, performance et.) – through the use of scenarios where these attributes are manifested. The scenarios can also be used as reference for situations where the pattern is applicable.

The main benefit of using Workflodize is the added flexibility. Programming a workflow is a visual process (at least with most workflow implementations) - which is very easy to master, The added flexibility can also result in quicker time to market for change requests. Workflow is, in my opinion, one of the most important tools to incorporate in a service on the long and winding road to business agility.

Here are a couple of examples for scenarios that can point you to looking at the Workflodize Service pattern.

**Table 2.5 Workflodize pattern quality attributes scenarios The architectural scenarios that can make us consider using the Active Service pattern.**

| Quality Attribute (level1) | Quality Attribute (level2) | Sample Scenario |
|---|---|---|
| Flexibility | Add process | For radical all pre-paid plans, adding support for the new plan will take less than 2 days. |
| Reusability | Core module | Reuse 90% or more of the of the common sales process for each new plan |

Workflodize pattern adds a lot of flexibility to a service as you can dynamically change behavior another aspect of flexibility can be found in the Edge Component pattern.

## Edge Component

The last pattern of the basic patterns is the Edge Component pattern. Unlike the other pattern which are basic just because they are very common, the Edge component is also basic because it is a platform for implementing other patterns. Because the Edge Component pattern is a step in implementing other patterns it is a little hard to come up with a concrete example that shows exactly why it is needed since the concrete examples fit the patterns the build on Edge Component. Instead I'll try to introduce few short examples and the communalities between them will lead us to the Edge Component.

### The Problem

Scenario 1

On one company I worked for we developed a Military Naval C4I platform. This platform had services that are reusable per solution. For example one of the core services provides a unified and centralized view of targets. The first implementation that was built on that platform used a messaging infrastructure using Tibco Rendezvous®. The next implementation has to use a different technology altogether ( WSE 3.01 ). Both implementations had to use the same business logic but needed different technologies to access that logic.

Scenario 2

On another project (that is also mentioned in the Workflodize pattern) - a cellular carrier constantly wanted to introduce new usage plans and offerings such as friends and family, night rate and the like to a service that handled ordering . The service interface remained pretty stable as the changes in the details were part of the XML but the business logic kept changing and adapting to the new plans.

What we see here is the opposite of what we had in scenario 1, here the interface and technology are stable and the business logic changes

Scenario 3

The last scenario is is a common situation in many projects. You normally have more than one service in a system. Each of these services handles a different business aspect yet all of them have to perform common tasks like making sure a request is authorized before performing a request, saving an audit trail etc.

In this scenario we have a functionality that is not related directly to a single service and is mostly repetitive across services – as it takes pretty much the same code to log the request even it on service handles orders and the other handles customers

The commonality between these scenarios is that we have different concerns (business logic, technology, logging etc.) all bundles within each service. As we've seen in the different scenarios, each of these concerns can change independently the others depending on the circumstances – we need a way to enable that flexibility so our problem is

> **How do we allow the business aspects of the service, technological concerns and other cross-cutting concerns like security, logging etc. to evolve in their own pace and independently of each other?**

The simplest, not to say simplistic, option is not to do anything in particular. An example for this approach is taking a piece of logic and directly exposing it as a web-service[2] which by the way, is very common in on-line samples technology vendors such as Microsoft (WCF) or Sun (JAX-WS) provide on their tutorials. However, when the handling of the contract is directly intertwined with the business logic

[1] Microsoft interim solution for the WS-* stack before Windows Communication Foundation

[2] Web service – is a method that is exposed over http. This is an unfortunate term since it overloads the word service for something that can easily be abused to create

implementation, the maintainability of the code greatly suffers – for example it would have been very hard to support scenario 1 and replace technology using this approach.

We could try to solve the problem of replacing a technology for an existing service by making a new copy of the service and making the changes there . An approach also known as "own and clone". The problem is, however this creates a maintainability problem as you now have multiple copies of the same business logic lying around and you'd have to make changes to all copies not to mention that it doesn't solve problems such as the ones presented in scenario 3 of adding logging capabilities to several services.

If not doing anything and cloning don't work maybe we can go for separation of concerns

### *The solution*

Separation of concerns is very known concept in the object oriented thinking. The root principle behind it is known as the "The Single Responsibility Principle" or SRP for short. SRP states that a class should have only one reason to change and that a responsibility is such a reason. We can apply the same principle within SOA and consider the business logic as one responsibility and the other concerns as another responsibility we get the following pattern:

> **Add Edge Component(s) to the service implementation to add flexibility and separate between the business logic and the other concerns like contacts, protocols, end point technology and additional cross-cutting concerns**

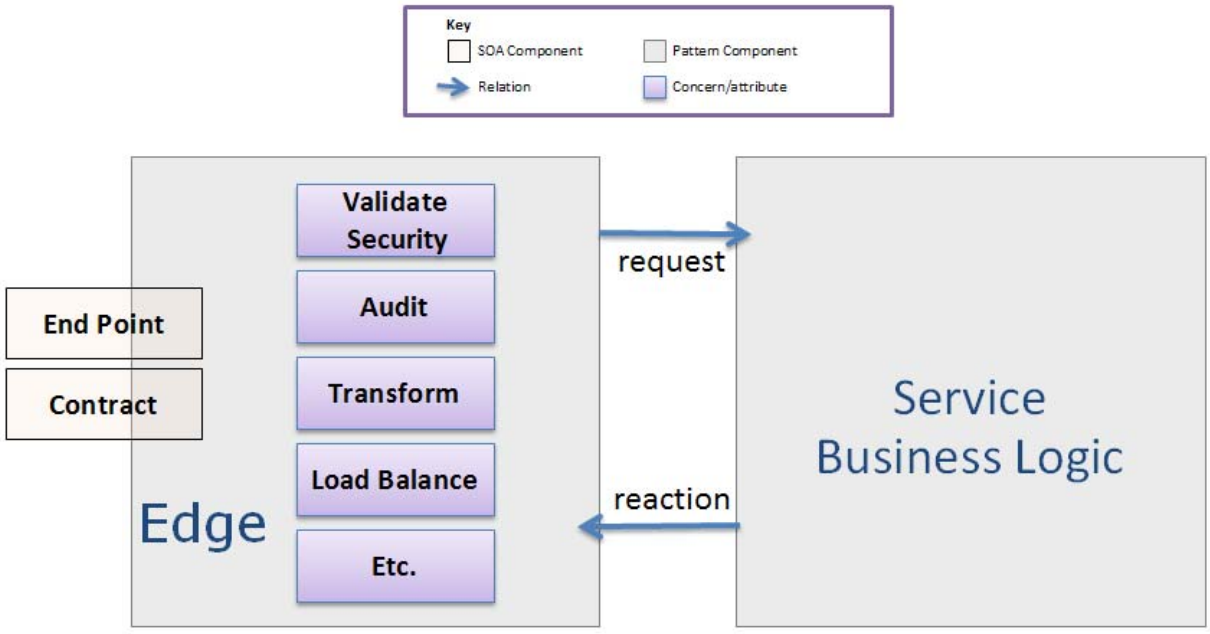



**Figure 2.11 The Edge Component pattern. The pattern means adding a component in from the business itself and having that new component, called an Edge, handle cross-cutting concerns like load balancing, auditing etc. Adding an edge allows the service itself to focus on the business logic and not clutter it with unrelated stuff.The edge component delegates request and replies between the service and its consumers.**

As Figure 2.11 above demonstrates, the main idea behind adding an Edge Component is separation of concerns. The Edge Component takes care of all the cross-cutting concerns and other concerns that are not in the core business of the service. These concerns can include areas such as load balancing, format

transformations, auditing. The business logic of the service is then handled in another component that focuses solely on the business logic and remains free of the other concerns. This separation allows supporting all the scenarios mentioned above since the separation allows each component to evolve in its own pace. For example to support a new technology (scenario 1) you just add an additional edge component but the business logic doesn't have to change. When you change the behavior of the business logic and add a new usage plan (scenario 2) the Edge component doesn't change.

In a sense the Edge Component pattern can be used to provide a façade, proxy and AOP patterns for SOA.

We still have to show how the problem in scenario 3 of implementing cross cutting concerns across services can be takes care of. The best approach to do that is to take single responsibility principle further and remember that the Edge Component is, indeed, a component and it doesn't have to map to a single "monolith" class. For example you can apply a pipes and filters architectural style and chain several classes/component, each dealing with a specific concern, to create incoming or outgoing pipe-lines For example, the figure 2.12 below shows a sample Edge Component that applies a validation filter to make sure the message is correctly formatted. Then there's a transformation filter to translate an external contract format into an internal one. Lastly there's a routing filter to send the message to the correct component within the service. These sub components can be reused from service to service as needed as well as change and evolve independently.

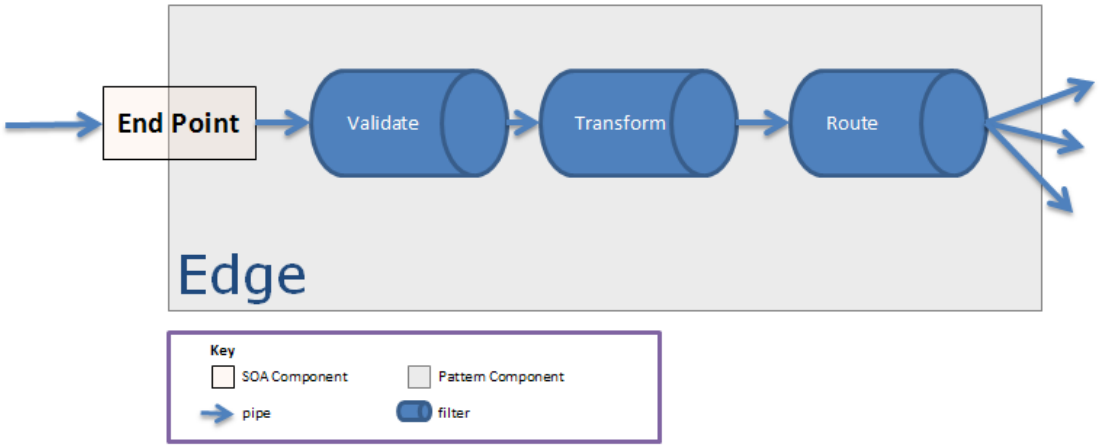



**Figure 2.12 Sample Edge Component pipe-line. As an incoming message is received on the endpoint it goes through validation, transformation to an internal format and routing to the correct component within the service.**

While it is tempting to define an inner contract between the Edge Component and the Service at the onset, there is no real reason to do that unless, maybe, you have to support multiple external contract (be weary of implementing a contract per consumer though – see PTP Integration anti-pattern). When the service evolves and newer versions of the contract are created like in scenario 1 of adding a new technology, you may have to add inner contract when you still want to support the older versions of the external one.

The edge component is very useful and I've introduced it in most of the SOA projects I architected. Many of the structural patterns mentioned in this book expand and build on the Edge component pattern.

Let's take a look at the technological aspects of the Edge Component pattern.

### *Technology Mapping*

As mentioned in the patterns structure in Chapter 1, the technology mapping section takes a brief look at what does it mean to implement the pattern using existing technologies as well as mention places technologies implement the pattern.

There is no technology that will take care of all the concerns the Edge component can fulfill automatically. The up side is that no technology, I know of, prevents you from implementing the Edge Component pattern and some technologies will even handle some of the concerns for you.

For example, JAX-WS or Windows Communication Foundation (WCF) basically implement the Edge Component pattern for you, but they only handle lower level concerns, which they sometimes call bindings. The concerns handled by JAX-WS and WCF are those mentioned in the various WS* standards, for example, WCF can handle MTOM encoding or Security for you. However as I already mentioned,you still need to code high-level concerns like routing, contract translations etc. by yourself.

An interesting technology option is a Java engine called Restlet®. Restlet® has a few built in classes that sets it as a good example for implementing the Edge Component Pattern

#### EDGE COMPONENT EXAMPLE – THE RESTLET® ENGINE

The Restlet® engine by Noelios Consulting, which is a Java library for implementing RESTful services, has some built is classes like filter and router that allow for easily building an Edge Component. Consider the example in the diagram below:

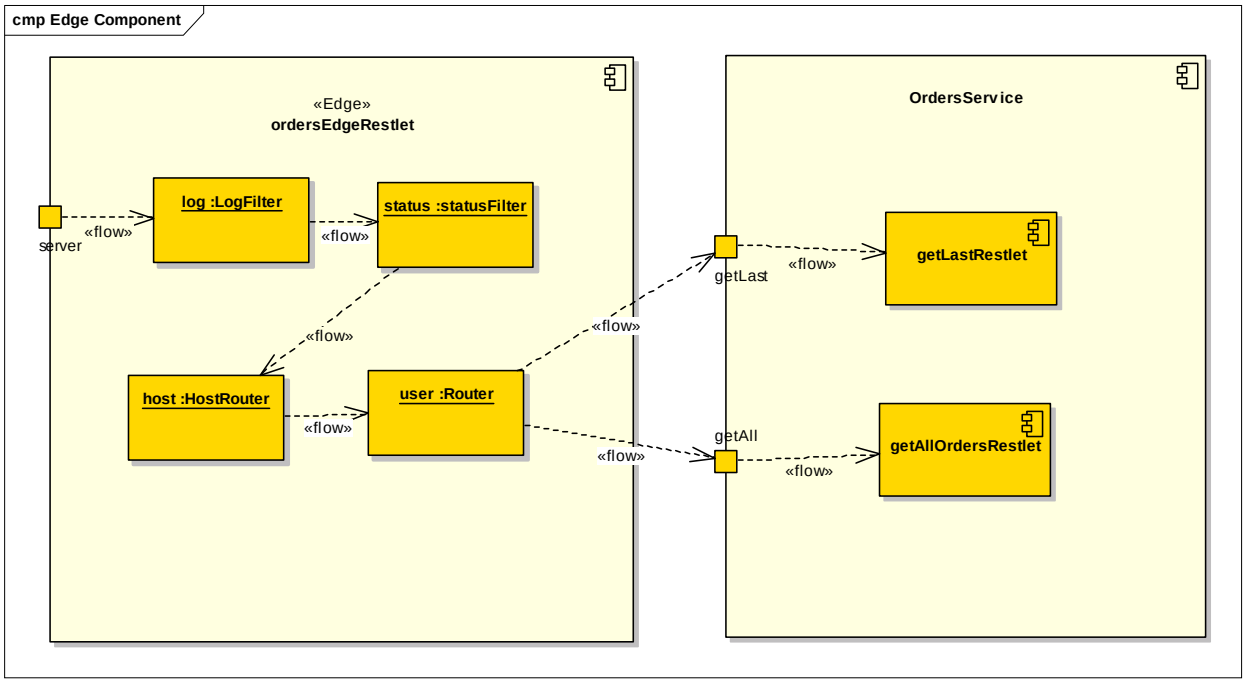



**Figure 2.13 Implementing Edge Component in Restlet®. We see an Implementation of an edge component by using some of the Restlet® components such as HostRouter, Router, statusFilter and LogFilter. As a request is received it gets routed through and handled by the different components before it gets to the actual business service.**

Figure 2.13 shows a possible Edge configuration on an Orders service whose contract has two operations: getLast which returns the last order and getAll which returns all the orders for a specific customer. However, before the call actually makes it to the business logic we have to log it, handle statuses or problems, make sure the correct host was used and finally route the call to the appropriate business functionality. Adding an Edge component lets us achieve all that without affecting the business logic which just processes the business requests.

Here is an excerpt of the above mentioned configuration in code :

```
Builders.buildContainer()

            .addServer(Protocol.HTTP, portNumeber)

            .attachLog("Log Entry")

               .attachStatus(true, "webmaster@mysite.org", "http://www.mysite.org")

                        .attachHost(portNumber)
                                              .attachRouter("/orders/[+")
                                                .attach("/getAll$",
getAllRestlet).owner().start();
                                                   .attach("/getLast$",
getLastOrderRestlet).owner().start();
```

Code excerpt 2.2 : except of an instantiaion code for defining an edge component using Restlet®

As we've seen The Edge Component pattern is supported by all current technologies and even implemented internally by some of them. You can take a look at the Further reading section at the end of the chapter for links to resources that expand on the technologies mentioned in this section.

### *Quality Attribute Scenarios*

The quality attribute scenarios section talks about the architectural benefits of utilizing patterns from the requirements perspective. As was mentioned in chapter 1, most of the architectural requirement are described from the perspective of quality attributes (scalability, flexibility, performance et.) – through the use of scenarios where these attributes are manifested. The scenarios can also be used as reference for situations where the pattern is applicable.

The Edge Component pattern can be associated with a lot of quality attributes. Most of these attributes however are the result of applying the Edge Component Pattern along with another pattern. There are however 2 quality attributes that are directly related to the Edge Component Pattern. The first is Flexibility – making it easy to change and enhance the external properties of the service without affecting the business logic. The second is maintainability – separation of concerns makes it easy to understand what each component is doing. Recall the three sample scenarios – adding a new technology to an existing service, changing business behavior without changing the contract and solving cross cutting concerns only once - with the Edge Component in place, we were able to solve the problem without affecting the rest of the solution or at least by minimizing it. Table 2.2 below shows a couple of sample scenarios that can direct us for using the Edge Component.

**Table 2.6 Edge Component quality attributes scenarios. The architectural scenarios that can make us think about using the Edge Component pattern.**

| Quality Attribute (level1) | Quality Attribute (level2) | Sample Scenario |
|---|---|---|
| Maintainability | Backward Compatibility | As contracts evolve, the services should be able to support consumers using older versions of the contract (at least the last 2 revisions) |
| Flexibility | Extension points | Within the next year the customer is expecting to need SOX compliance and add auditing across the board |

The Edge Pattern is the last of the basic structural patterns for SOA. To wrap this chapter up, let's take a look at the patterns that were covered - before we take a look at additional structural pattern for availability and scalability patterns.

## *Summary*

Chapter 2, being the first patterns chapter, deals with few of the basic structuring patterns for building services. The patterns covered in this chapter include:

- Edge Component – separate interface (contract) from implementation to enable flexibility and maintainability
- ServiceHost – make a common wrapper to host service instances and reuse across services
- ActiveService – Have at least one independent thread in the service so it can initiate
- Transactional Service – handle messages inside a transaction to gracefully recover from crashes
- Gridable Service – make (parts of ) your service stateless to enable running it in a grid cluster and solve computational heavy tasks
- Workflodize – add a workflow *inside* the service for added flexibility

The next couple of chapters talk about additional issues of building services as handling scalability, performance and availability requirements (chapter 3) as well as security and management (chapter 4)

## Bibliography

**Barbara Liskov** Data Abstraction and Hierarch [Conference] // Proc. of OOPSLA conference . - 1988.
**Anderson Robert W. and Daniel Ciruli** Scaling SOA with Distributed Computing [Journal] // Dr. Dobb's Journal. - 2006.