SOA Patterns

Arnon Rotem-Gal-Oz

10.1 The Knot

Everything starts oh so well. Embarking on a new SOA initiative the whole team feels as if it is pure green field development. We venture on - The first service is designed. Hey look it got all these bells and whistles; we are even using XML so it must be good. Then we design the second service, it turns out the first service has to talk to the second – and vice versa. Then comes a third, it has to talk to the other two. The forth service only talks to a couple of the previous ones. The twelfth talks to nine of the others and the fourteenth has to contact them all – yep our services are tangling up together into an inflexible, rigid knot

The above scenario might sound to you like a wacky and improbable scenario - why would anyone in the right mind do something like that? Let's take another look, with a concrete example this time and see how the road to hell is paved with good intentions. In Figure 10.1 below we see a vanilla ordering scenario. An ordering service sends the order details to a stock service, where the items are identified in the stock, marked for delivery and then sent to a delivery service which talks to external shipping companies such as DHL, FedEx etc.

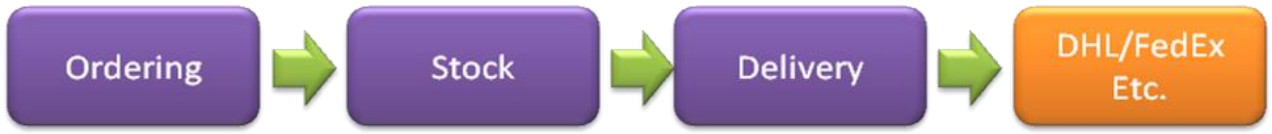

Figure 10.1 a vanilla ordering scenario. An ordering service sends the order to a stock service, which provisions the goods to a delivery service which is responsible to send the products to the customer

If we think about it more we'll see that when an item is missing from the stock we probably have to talk to external suppliers, order the missing items and wait for their arrival- so the whole process is not immediate. Furthermore since the process takes time, it seems viable to cancel the process if an order is cancelled. It seems we have two options (see Figure 10.2) either the ordering service will ask the two other services to cancel processing related to the order or the two services call the ordering service before they decide what to do next. Naturally the system wouldn't stop here, we would want to introduce more services and more connections e.g. an Accounts Payable service that interacts with the external suppliers, the stock service and the delivery service(since we also need to pay shipping companies) etc.

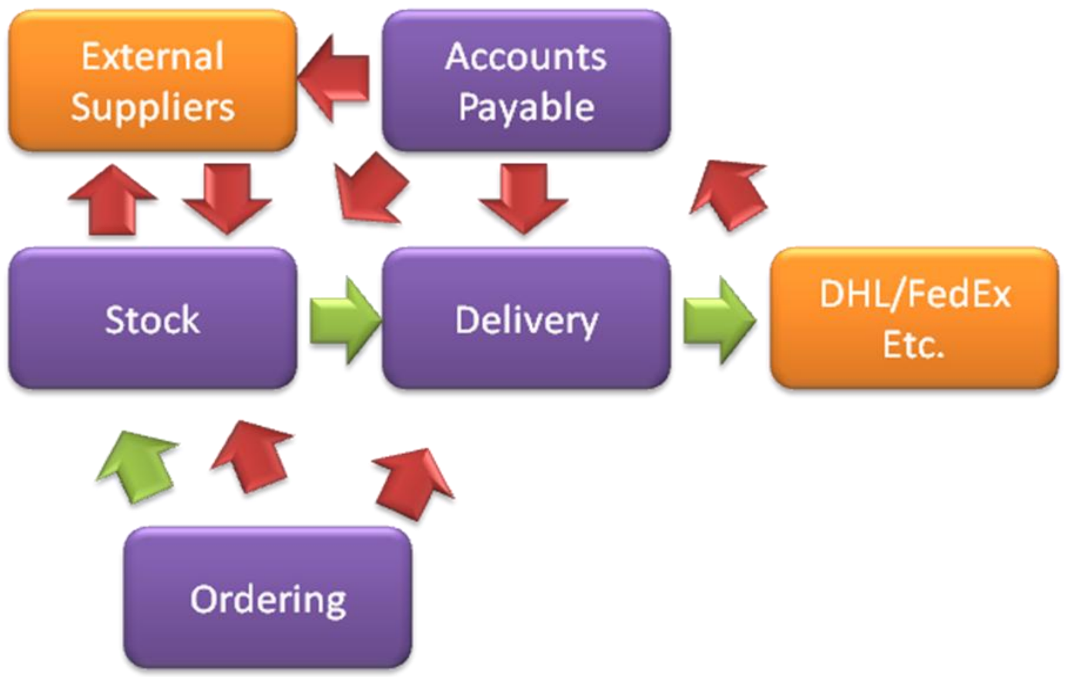

Figure 10.2 a little more realistic version of the Ordering scenario from figure 10.1. Now we also need to handle missing items in the stock, cancelled orders and paying external suppliers. In this scenario the services get to be more coupled. For instance the Ordering service is now aware of the delivery service and not just the stock service.

With each new service we draw more lines going from service to service, and with each new service we update the services' business logic with the new business rules as well as knowledge of the other services' contracts.

10.1.1 Consequences

Well, so we get more lines going from service to service that normal isn't it? After all if the services won't talk to each other they won't be very useful? Isn't that the whole point of SOA?

Well, yes – and no. Yes it is normal for services to connect to each other. After all, creating a system in an SOA is connecting services together. As for the "no" part, the problem lies with the way we develop these integrations if you are not careful it is easy to get all the integration lines in a big, ugly mess – a knot

> **A knot is an Anti-pattern where the services are tightly coupled by hardcoded point-to-point integration and context specific interfaces**

For instance, what happens when we want to reuse the ordering service mentioned above? No problem, we just call it from the new context. Alas, the knot prevents us from reusing it without hauling in the rest of the baggage - all the other services we defined above (the stock, delivery etc.) if the new context is not identical in it ordering processes and matches what we already have we can't use it. Or we can't use it without adding one-off interfaces where we add specific messages for the new context and all sort of "if" statements to distinguish between the old and the new behavior. Another option is to make this distinction in the original messages, which either not possible or forces us to make sure the other services are still functioning. In any event it is a big mess.

Let’s recap. We moved to SOA to get flexibility, increase reuse/use within our systems, prevent spaghetti point to point integration – what we see here is not flexible, hard to maintain and basically it seems like we are back in square one and we invested gazillions of dollars to get there.

10.1.2 Causes

How did that happen? How can a wonderful, open standards, distributed, flexible SOA deteriorate to an unmanageable knot?

It is tempting to dismiss the knot as the result of lack of adequate planning. If we only planned everything in advance we wouldn’t be in this mess now. Well, besides the point that trying to plan everything ahead of time is an anti-pattern in itself (an organizational anti-pattern – which isn’t in the scope of this book). There’s still a good chance you’d get to a Knot anyway since the problems are inherent in the way business work.

If we take a look back at the Integration Spaghetti scenario discussed in chapter 1 (depicted as figure 10.3 below), we can see that the phenomena was there as well, when we our business processes evolve we find we need to interact with information from other parts of the system. The flow of a business process expands to supply that needed information or service and thus the Knot grows.

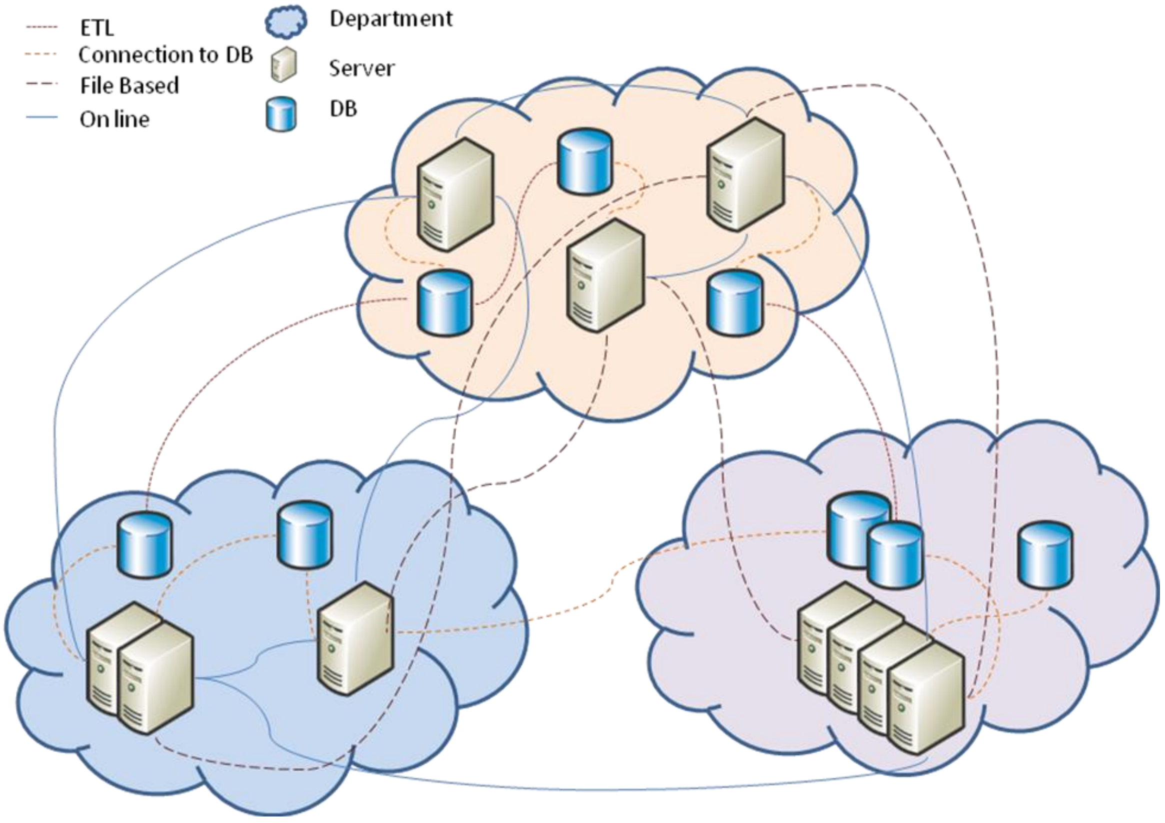

Figure 10.3 the Knot anti-pattern is similar in both effect and origin to the spaghetti integration in non-SOA environments

From the technical perspective, we have two forces working here. One is the granularity of the services. On the one hand, Services are sized so that a business process requires several of them to work together. On the other hand they aren't small enough so that they would be an end-node in the process (i.e. only other services would call the service and it will just return a result). Note that this isn't a bad thing in itself, after all if each process was implemented by a single service we'd have silos not unlike the ones we try to escape by using SOA and if we set the services too small we'd fall into another trap (see the Nanoservices anti-pattern later in this chapter). The bottom line is that while the granularity is a force that drives us toward the Knot, there's not a lot we can do about it without getting ourselves into worse problems.

The second, stronger, force that pushes a system into a Knot is the business process itself. Since, as we mentioned above, the process flows through the services, the services needs to be aware of the flow and then call other services to complete the flow. In order for a service to call another service it has to know about its contract and know about its endpoint. When another business flow goes through that service we not only add the new contracts and endpoints but also the contextual knowledge of which other services to call depending on the process. And that's my friends, is exactly the thing that gets us into trouble – the services start to tie themselves to each other more and more, as we implement more business process and more flows.

Hey, you say, but SOA should have solved all that, surely there is something we can do about it – or is there?

10.1.3 Refactoring

The previous section explains that most of the problem is caused by having the services' code determine where to go next and what to do with the results of the services' processing. If there was only a way to somehow pry these decisions away from the services' greedy hands… As you'd probably guessed there is such away, in fact there are several such ways and this book lists three of them: The Workflodize pattern (Chapter 2), Orchestrated Choreography (Chapter 7) and Inversion of Communications (Chapter 5). Let's take a brief look at each of these patterns and see how they help.

The workflodize pattern suggests adding a workflow engine inside the service to handle both Sagas (i.e. long running operations, see chapter 5) and added flexibility. The "added flexibility" is the card we want to play here. When we express the connections as steps in the workflow they are not part of our services' business logic. They are also easier to change in a configuration-like manner both of these points are big plusses.

Still, a better way to solve the service to service integration problem is to use an external orchestration engine. The idea of using the Orchestrated Choreography pattern is to enable Business Process Management- or a way for the organization to control and verify it processes are carried out as intended (you need an orchestration engine for that but it helps…). In the context of solving or avoiding the Knot anti-pattern, Orchestrated Choreography is better than Workflodize since it centralizes and externalizes all the interactions between services and thus effectively removing all the problematic code from the services themselves. Note that there’s a fine line between externalizing flow and externalizing the logic itself (see discussion in Orchestrated Choreography pattern, in chapter 7).

The third pattern we can use to refactor the Knot is Inversion of Communications. Inversion of Communications means modeling the interactions between services as events rather than calls. Inversion of communications is, in my opinion, the strongest countermeasure to the knot. The two patterns mentioned above bring a lot of flexibility in routing the messages between the services. The inversion of communications pattern also helps the message designers remove specific contexts from the messages since when the service’s status is raised as an event it isn’t addressed to any other service in particular. Note that using inversion of communications doesn’t negate using either of the two other patterns mentioned above since that once the event is raised we still need to route it to other services and using a workflow engine is a good option for that. Another implementation option is to use an infrastructure that supports publish/subscribe (see the pattern’s description in chapter 5 for more details.)

Going back to the ordering scenario we mentioned above. As I mentioned, the services grow with needless knowledge of specific business process. So for instance, the ordering service had to know both about the stock service and the delivery one. Refactored with the Inversion of Communications pattern, the same Ordering service doesn’t have to know about any of the other services. In Figure 10.4 we can now see that the Ordering service sends two business events (new order, cancelled order) and the routing of these messages is no longer the responsibility of the service

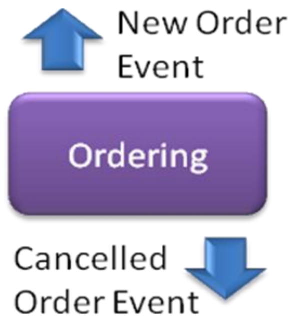

Figure 10.4 the Ordering service using the Inversion of Communications pattern. Now the service doesn’t know/depend on other services directly. It is only aware of the business events of new order and cancelled order which are relevant to the business function that the service handled

Refactorings aside, one question we still need to think about is whether there are any circumstances where having a Knot is acceptable.

10.1.4 Known Exceptions

In a sense the Knot is a distributed version of an anti-pattern described by Brian Foote and Joseph Yoder as “Big Ball of Mud” – spaghetti code where different types of the system tied to each other in unmanageable ways. The reason for mentioning the connection is that the reason that “Big Ball of Mud” might be considered a pattern rather than an anti-pattern also apply here:

“[when] you need to deliver quality software on time on budget… focus first of feature and functionality, then focus on architecture and performance”

Starting out on a large project, such as moving an enterprise to SOA, is difficult enough as it is. You can’t figure everything in advance; you need to deliver something – so as Nike says “just do it”. Get something done. You do need to be prepared to let go and redesign further down the road. In the current system I’m working on – a visual recognition/search engine for mobile, we went with a “knot” approach for the first release. The simplicity of the implementation, i.e. less investment in infrastructure, ad hoc integration etc. enabled us to deliver a first working version in less than 6 months. These 6 months also helped us understand the domain we are operating in much better and more importantly get to market with the feature the business needed in the schedule the business wanted. We spent the next 6 month rewriting the system in a proper way, including applying the Inversion of Communications pattern mentioned above.

To sum this up, coding the integration code into services is likely to end as a Knot. It is acceptable to go down this path for a prototype or first version i.e. to show quick results. However you do need to plan/make the time to refactor the solution so you will not get stuck down the road.