A pattern language for service input data provisioning

Geert Monsieur, Monique Snoeck, Wilfried Lemahieu
Katholieke Universiteit Leuven
Faculty of Business and Economics
The Leuven Institute for Research on Information Systems (LIRIS)
firstname.lastname@econ.kuleuven.be

ABSTRACT

A common practice in service-orientation is the creation of a composite service by combining a set of other services. As discussed in this article, the orchestration of services to construct a new service requires several service interactions. This is why the construction of a composite service can be a complex and time-consuming task. Some services in a service composition can have the role of providing other services with (additional) input data. The pattern language in this article can help to design the service interactions that are needed for provisioning input data.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures—*Patterns*

General Terms

Design, Management, Languages

Keywords

Service composition, service input data, patterns, coordination, data flow, data dependencies

1. INTRODUCTION

1.1 Service-orientation

In modern software engineering service-orientation is about grouping a company's capabilities into well-defined and scoped services. A service should consist of a collection of capabilities that are grouped together because they relate to a functional context established by the service [2]. A finance service provides finance-related capabilities, e.g. invoice creation or invoice payment status monitoring. In that way, services become independent entities which hold solutions that are valuable for the business. This should promote reuse of solutions and increase flexibility. In the ideal service-oriented world, changes in business processes are carried out by changing the way the company's business services are combined to support the business process. By clustering solution logic into services, the *building blocks* for business processes become less dependent on and more agnostic to any business process. The end idea is that all business services can be combined to fulfill the customer's needs and requests. It is assumed that when a company's business services are constructed in a proper way, an activity in a business process can be executed by consuming a service and accessing a capability provided by that service. In summary, the service-oriented architecture is an organizing paradigm that enables one to get more value from the use of both capabilities that are locally 'owned' and those under the control of others that are exposed as business services. The concept "service" refers to the enabling mechanism that provides access to a set of capabilities. Service implementation details are considered as less important when specifying and communicating service descriptions to potential consumers, which are only interested in the valuable use of the service, instead of the way the service is provided. Nevertheless, providing and composing new services by combining capabilities and services of other services is one of the key principles behind service-orientation. Therefore, the way a service is provided or implemented is sometimes specified using a set of supporting services, possibly complemented with some kind of process logic which describes how the supporting service should be combined to provide the new service. The resulting service is a *composite* service.
In this article the main focus is on the provisioning of a composite service. As we will discuss in the next subsection the orchestration of services to construct a new service requires several service interactions. This is why service composition can be a complex and time-consuming task. The pattern language presented in this article can support the process of designing the required service interactions.

16th Conference on Pattern Languages of Programs (PLoP). PLoP'09 August 28-30, Chicago, IL, USA

1.2 Service interactions

As described in the previous subsection, a service implementation can be specified in terms of a set of supporting services which are combined in a certain way to provide the new composite service. In figure 1 a composite service is represented. Services A to F form the set of supporting services. Combining this set of services requires precise coordination. The exact service interactions that are needed strongly depend on the role each service plays in the overall orchestration. Therefore we propose to make a distinction between two roles a supporting service can have:

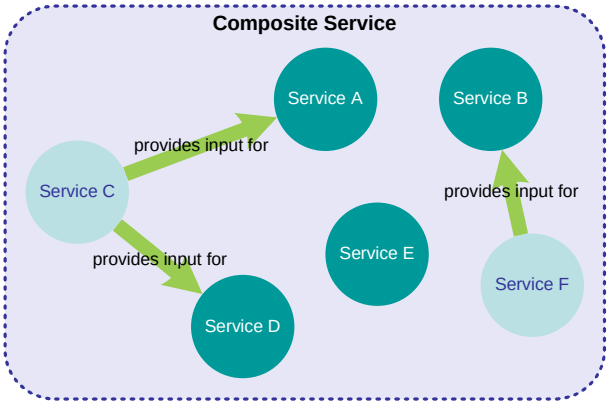


Figure 1: Six supporting services that together form a composite service

- *Service providers* contribute in a *direct* way to the realistion of a capability. The idea is that each service provider involved should be consumed in order to provide the capabilities of the composite service. A service provider provides a service that is crucial to the provisioning of the composite service. It is possible to consider two kinds of service providers. A first kind refers to service providers that should do some processing and the result of this processing is relevant for the successful provisioning of the composite service. As a consequence the service provider should return some kind of an output. This can be either a simple confirmation that the processing went well or it can be some specific data. A second type refers to service providers that simply need to be notified. This interaction can be considered as an application of the fire and forget pattern [5]. In delivering the composite service, it is important that this service provider is notified. However, *how* this service provider *processes* this notification is not relevant for the successful provisioning of the composite service.

- *Data holders* can be another type of supporting services. Services of this type only contribute in an *indirect* way to the provisioning of the composite service. Consumption of these services is only needed because they can provide input data that service providers need to contribute to the composite service. Per definition, the data provided by a data holder is only needed by a service provider. Services that provide data that is not used as input data for service providers, are considered to be service providers. Such a service provider only provides data to the composite service.

In figure 1 services A,B,D and E have the role of service providers. Services C and F are data holders, which means that these services provide input data for some service providers. Note that in reality, a service can have both roles, but in order not to overload the analysis a strict distinction is made in the rest of the paper.
If more than one service provider is involved in a service composition many service interactions can be required. Not only should each service provider be triggered, which requires service interactions, but possibly also every service provider returns some kind of answer. Furthermore, it can become even more complex when additional interactions are required for transactional purposes. Coordinating service providers to construct transactional (business) services is out of scope of this paper, but for a detailed discussion the reader is referred to [3].
The use of data holders in a service composition is another challenge. As defined above, data holders provide data that is required as input data for service providers. This implies that interactions are required to transmit the data between the data holder and the service provider. As we will discuss in the next section, the pattern language presented in this article is about designing the service interactions that are needed for providing the service provider with (additional) input data that is available at the data holder.
In section 5 a concrete example is presented, describing a service composition case in a hospital.

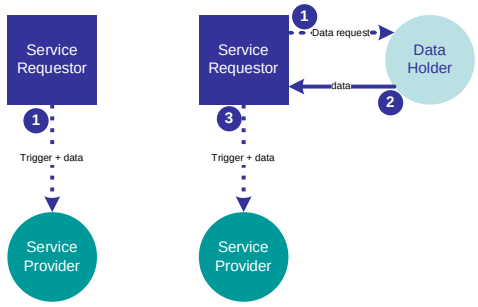
2. THE PATTERN LANGUAGE

2.1 Context

As described in the introduction, a service provider should do some sort of processing in order to contribute to the provisioning of a new composite service. In that way the service provider provides some kind of a service to the composite service or the entity that is responsible for composing the new composite service. In the rest of this article the entity that consumes that service (by sending a request for processing to the service provider) is referred as the *service requestor*. In order to process the request and provide the required service, the service provider possibly needs a certain amount of input data. We refer to such services as *needy* service providers. In some cases all input data is available at the service requestor and can be sent together with the request to the service provider (see figure 2(a)). In other cases, it is possible that the data is not available at the service requestor, but can be retrieved in advance from an external party called a *data holder* (see figure 2(b)). It is assumed that data holders only provide data after requests are sent to them. In both cases all input data is available just *before* the service requestor triggers the service provider, which makes it quite straightforward to manage and coordinate the necessary interactions. However, in other situations it is desirable to trigger the service provider as soon as possible. As a result it can occur that (part of) the input data is only available *after* the service provider is triggered and should still be provided by a data holder. As discussed in the next subsection these situations can lead to complex service interactions and any guidance when designing the interactions can be very valuable. The pattern language in this article is applicable and supportive in such situations where (part of) the input data is only available some time after the moment the service requestor sent a request for processing to the service provider.

2.2 Problem

A service provider needs certain input data in order to process a request of the service requestor. Some part of the input data matches data that is available just before triggering and is sent together with the request to the service provider. Other parts of the input data can only be provided by one or more data holders later on. **This raises the question what would be the best solution to deliver missing input data, that is available at the data**



(a) All input data available at the service requestor

(b) Input data available at data holder is retrieved in advance

Figure 2: All input data available just before triggering the service provider

holder, to the service provider. Many scenarios are possible. For example, the responsibility of collecting missing input data can be assigned to either the service requestor or the service provider. In the first case data provided by the input data can easily be reused for triggering other service providers, while the second case is perhaps more preferable when the input data provided by the data holder is confidential and can not be shared with a central service requestor. Another scenario could consist of the service requestor instructing the data holder to send the required data to the service provider.
Given the fact that more than one data holder can be involved in providing the required input data for a service provider, and the fact that a data holder can possibly provide data to more than one service provider, it is important to know how to *evaluate* each scenario in order to figure out which scenario is preferable in which business case and for which data.

2.3 Solution and overview of the pattern language

To solve the problem of providing the missing input data, we propose a pattern language that consists of three basic patterns: ACTIVE-PASSIVE SERVICE PROVIDER (see subsection 3.1), DIRECT-INDIRECT REQUEST (see subsection 3.2) and DIRECT-INDIRECT REPLY (see subsection 3.3). The context as described above (see subsection 2.1) is closely related to the context of the ACTIVE-PASSIVE SERVICE PROVIDER pattern. An application of this pattern can function as a first necessary step in solving the problem of providing missing input data. The next steps towards a solution can be derived from figure 3, which gives an overview of the complete pattern language. Each pattern is visualized in a box containing both the pattern name and sub-boxes referring to possible solutions in that pattern. The arrows show the relationships between the patterns (based on the context and resulting context decriptions of the patterns). In particular, an arrow pointing from a pattern sub-box A to a pattern box B indicates that the pattern represented by B should be applied next when a pattern is applied in the way represented by sub-box A. For example, there is an arrow that indicates that the DIRECT-INDIRECT REQUEST pattern should be applied when an active service provider is chosen by applying the ACTIVE-PASSIVE SERVICE PROVIDER pattern.

3. PATTERNS FOR INPUT DATA PROVISIONING

3.1 Active-Passive Service Provider

Context

A service provider is triggered by a service requestor. At the moment of triggering the service requestor did not send sufficient data to the service provider for completing its internal processes. Therefore additional input data should be collected. In the rest of this pattern this task is referred to as *the data collection process*.

Problem

How can the data collection process be initated?

Forces

- *Interface modification*: In a service-oriented environment it is critical that the propagation of modifications due to the modification of the interface of a service provider is minimized. Consumers prefer to rely on a service provider that only rarely changes its interface. A change in the data requirements should minimally change the way the service provider is consumed.
- *Handling staged data provisioning*: Providing input data in several stages can have substantial impact on the service provider's internal working. The service provider should have a mechanism to correlate each incoming part of the input data. In other words, all data received by the service provider should be linked to the right request.
- *Loose coupling*: Service-orientation is often related to a loosely coupled world. This means that preferably a service provider's implementation does not have to rely on several other services. As such, in order to loosen the coupling between a service provider and a data holder it can be desirable that the service provider is not responsible for collecting all data that is required.

Solution

In general a service provider can be considered either active or passive, which results in two possible solutions for iniating the data collection process (see figures 4(a) and 4(b)).
In cases a service provider does not wait for additional input data, but starts requesting additional data it can be considered an *active* service provider. An active service provider iniates the data collection process by sending out a data request (see step two in figure 4(a)). It is not specified to which entity the service provider sends out its data requests. This problem is discussed in an other pattern (see DIRECT-INDIRECT REQUEST in subsection 3.2).
In cases the service provider simply waits for the missing input data it can be considered a passive service provider. After the service provider is triggered, the service requestor iniates the data collection process by sending a data request (to the data holder) (see step two in figure 4(b)).
In summary the main difference is in step two. In the active service provider scenario the service provider iniates the

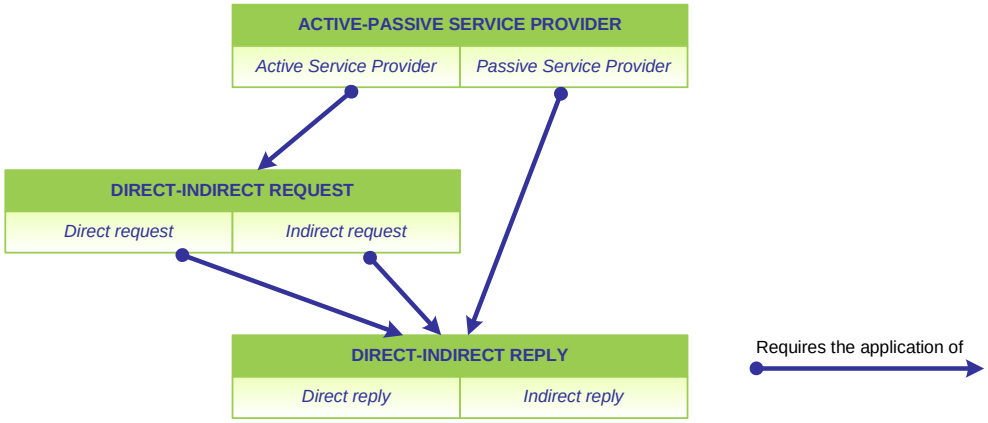
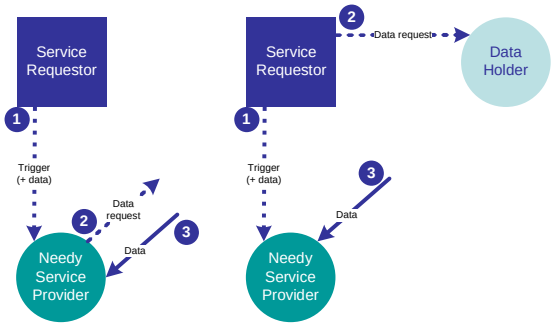


Figure 3: Overview of the pattern language



(a) Active SP (b) Passive SP

Figure 4: Active versus passive Service Provider (SP)

data collection process by sending out a data request, while in the passive service provider scenario the service requestor iniates the data collection process by sending a data request to the data holder.
Selecting an appropriate scenario is about balancing the forces. Each scenario deals in its own way with the forces that are relevant in this pattern:

- *Interface modification*: In the passive scenario every change in data requirements results in a change in the implementation of the service requestor. In contrast, in the active scenario theses changes are only reflected in modified data requests sent by the service provider itself. Consumption of active service providers is considered to be rather stable.
- *Handling staged data provisioning*: Clearly, a passive service provider must be able to handle data provisioning in several stages. As a consequence it should have a mechanism to correlate each incoming part of the input data. For an active service provider things are easier, because additional input data is only received as an answer to requests that the service provider itself sent out.
- *Loose coupling*: It is clear that an active service provider is more coupled with the external world, because it needs to send out data requests to known external parties. In contrast, a passive service provider simply expects that the data is provided at some point in time. Passive service providers do not have to iniate interactions with external parties (for input data purposes).

Resulting context

An active service provider sends out data requests in order to receive the missing input data (see step two in figure 4(a)). As such, the resulting context when using an active service provider can be linked to the context of the DIRECT-INDIRECT REQUEST pattern.
The passive service provider scenario implies that the service requestor sends data requests to the data holder (see step two in figure 4(b)). As a consequence the data holder receives data requests, which corresponds to the context of the DIRECT-INDIRECT REPLY pattern.

3.2 Direct-Indirect request

Context

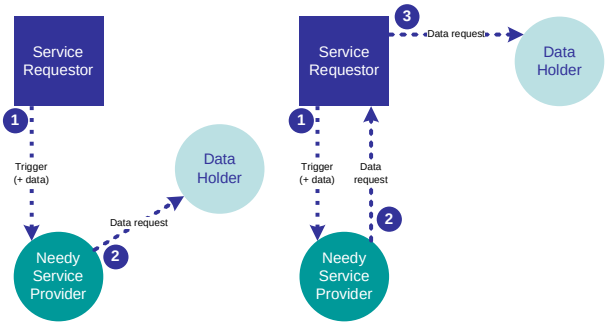
A service provider is triggered by a service requestor. The service provider is active, which means that it sends out data requests in order to receive the missing input data.

Problem

Where can the service provider send its data requests to?

Forces

- *Is the data holder known by the service provider?* Sometimes it is possible that the service provider does not know *which* service holds the required data. Then the service provider can not send its data requests directly to the data holder.
- *Does the service provider have access to the data holder?* Sometimes it is possible that the service provider does not have *access* to the specific data holder. Then the service provider can not send its data requests directly to the data holder.
- *Coupling between data holder and its requesting party*: how long has a specific service the role of data holder? Each time a service takes over the role of data holder



(a) Direct request (b) Indirect request

Figure 5: Direct versus indirect requests

the party that is sending data requests to the data holder needs to be notified and modified properly. How often is the interface of the data holder modified? Each change in the interface of the data holder, requires a change in the implementation of the party that is interacting with the data holder.

- *Is data holder known by the service requestor?* If only the service provider knows the data holder, while the service requestor does not, the service requestor can not send out data requests to the data holder.
- *Does the service requestor have access to the data holder?* If only the service provider has access to the data holder, while the service requestor does not, the service requestor can not send out data requests to the data holder.

Solution

An active service provider can send its data requests to two entities, as shown in figures 5(a) and 5(b). Firstly, an active service can send a *direct request*, which means that the data request is sent directly to the data holder. Secondly, an active service provider can send its data request to the service requestor (see step two in figure 5(b)), which is supposed to forward the data request to the appropriate data holder (see step three in figure 5(b)). This alternative is referred as an *indirect request*.
By taking the different forces into account, an appropriate solution can be chosen:

- *Is the data holder known by the service provider?*: The direct request scenario requires that the data holder is known by the service provider, while in the indirect request scenario only the service requestor needs to know which service plays the role of data holder.
- *Does the service provider have access to the data holder?*: The direct request scenario requires that the data holder can be accessed by the service provider, while in the indirect request scenario only the service requestor needs to have access to the data holder.
- *Coupling between data holder and its requesting party*: In the direct request scenario there is a strong coupling between the service provider and the data holder. Sending data requests to the service requestor, as in the indirect request scenario, removes this coupling. However, note that in the indirect request scenario there is a coupling between the service requestor and the data holder. Perhaps this can be considered more acceptable because service requestors are also strongly coupled with service providers that need to be triggered.
- *Is data holder known by the service requestor?*: In the direct request scenario only the service provider needs to know which service plays the role of data holder, while the indirect request scenario requires that the data holder is known by the service requestor.
- *Does the service requestor have access to the data holder?*: In the direct request scenario only the service provider needs to have access to the data holder, while the indirect request scenario requires that the data holder can be accessed by the service requestor.

Resulting context

In both scenarios the data holder receives a data request (see step two in figure 5(a) and step three in figure 5(b)). As such, the resulting context of this pattern can be linked to the context of the DIRECT-INDIRECT REPLY pattern.

3.3 Direct-Indirect reply

Context

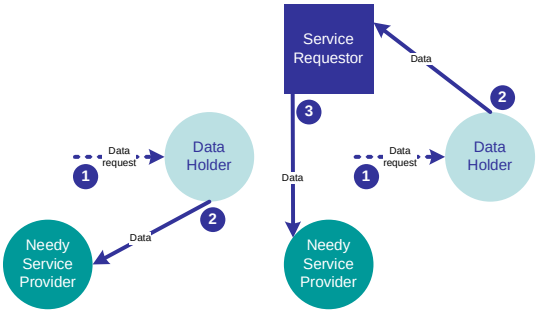
The data holder received a data request. The service provider needs the requested data.

Problem

How can the data be transmitted from the data holder to the service provider?

Forces

- *Confidentiality of data:* When requesting a data holder to send the required data to an entity, it is important to realize that the provided data can be confidential and therefore the data holder can limit the entities with which it is willing to share the data. For example, a data holder can demand that the provided data is only sent to the service provider that needs the data and that it is can not be shared with other service providers or stored by a service requestor.
- *Reuse of retrieved data:* In some business cases data provided by a data holder is used by more than one service provider. In such situations you can either request the same data several times, for each service provider that needs the data as input, or you can request the data only once and store it for reuse with other service providers. The latter approach sets restrictions on the solution for transmitting the data to the service provider.
- *Data transformations*: When the data holder replies, the data that is provided is possibly not in a form that is expected by the service provider. For example, the data format needs to be adapted, or the data should be made anonymous. In summary, in some cases data transformations are needed before the data is received by the service provider.



(a) Direct reply (b) Indirect reply

Figure 6: Replying

Solution

After a data holder receives a data request (see step one in figures 6(a) and 6(b)), its reply can be transmitted in two ways. Firstly, the data holder can send a *direct reply*, which means that the data is sent directly to the service provider (see step two in figure 6(a)). Secondly, the data can be transmitted from the data holder to the service requestor (see step two in figure 6(b)) and subsequently to the data holder (see step three in figure 6(b)). This alternative is referred to as an *indirect reply* (see figure 6(b)).
Based on an evaluation of all forces, an appropriate scenario can be chosen:

- *Confidentiality of data:* When the provided data is confidential, a direct reply is the best scenario, since an indirect reply implies that the data is passed through the service requestor before it is received by the service provider.
- *Reuse of retrieved data:* An indirect reply could facilitate the reuse of the provided data. For example, the service requestor only receives the specific data once, before distributing the same data to several service providers.
- *Data transformations*: An indirect reply allows data transformations, since all data that needs to be transmitted to the data holder is passed through the service requestor. As such, the service requestor can be responsible for data transformations. However, in a direct reply scenario the service requestor is not involved when the data needs to be transmitted from the data holder to the service provider. As a consequence, data transformations are not possible.

Resulting context

The service provider received the additional input data.

4. COMBINING THE PATTERNS

As described in subsection 2.3 clear relationships between the patterns exist, which shows how the patterns can be combined to construct a proper solution for providing a service provider with additional input data. In total, one can come up with six possible combinations that form a solution. In figures 7(a) to 7(f) the solutions are represented. Underlined words in the figures' captions indicate how the patterns are applied.

5. EXAMPLE

This pattern language is intended as a tool for service composition. In particular it helps to design service interactions that are required for transmitting additional input data from the data holder to the service provider. Although the concept of a service is often related to a pure software service, every single idea in this pattern language can also be applied to pure business services. In this section we give a small example that shows how the pattern language can be applied in a hospital.
In a hospital nurses provide several (business) services to patients. One of these services could be lowering a patient's fever. In order to compose this service the nurse needs to consume another service. In particular the nurse should request a febrifuge from the pharmacist. Hence, the nurse can be considered as a service requestor, while the pharmacist plays the role of service provider. Both aspirin and paracetamol are fever reducers. However, aspirin has the unpleasant side effect that it can cause stomach bleeding in certain circumstances. Therefore, it is supposed that the pharmacist needs information concerning the risk for stomach bleeding, before he or she can deliver an appropriate febrifuge. The risk for stomach bleeding is known by the patient's doctor. As such, the doctor can be considered as the data holder.
Below you can find an overview how the patterns can be applied to this example:

- Active-passive service provider: Since nurses do probably not want to understand which input data is required by the pharmacist, it is probably more desirable to choose an active pharmacist. Nurses simply want to use some services provided by the pharmacist, and it is not preferred that changes in data requirements result in changes how the nurses work (or consume the pharmacist's services).
- Direct-indirect request: This pattern needs to be applied, because pharmacists are considered as active service providers. Since the pharmacists do not know which doctor is treating the patient, it is preferred that the pharmacist asks the nurse for more information concerning the risk for stomach bleeding (see step two in figure 8). Subsequently, the nurse can forward the request to the right doctor (see step three in figure 8).
- Direct-indirect reply: Suppose the risk for stomach bleeding is quite confidential data and can not be shared with the nurse. Then, the indirect reply scenario is the best solution. Hence, the doctor should send the information concerning the risk for stomach bleeding directly to the pharmacist (see step four in figure 8).

The complete solution for this example is shown in figure 8.

6. RELATED WORK

One of the first papers that discussed the use of patterns in service composition is [4]. In that paper it is investigated how patterns can be used in service composition to help in the development of business applications based on e-services. In particular, they presented payment mechanism patterns, which can be applied quite often in e-services. Payment mechanisms can be seen as interactions between three different processes: billing, payment and execution of service.

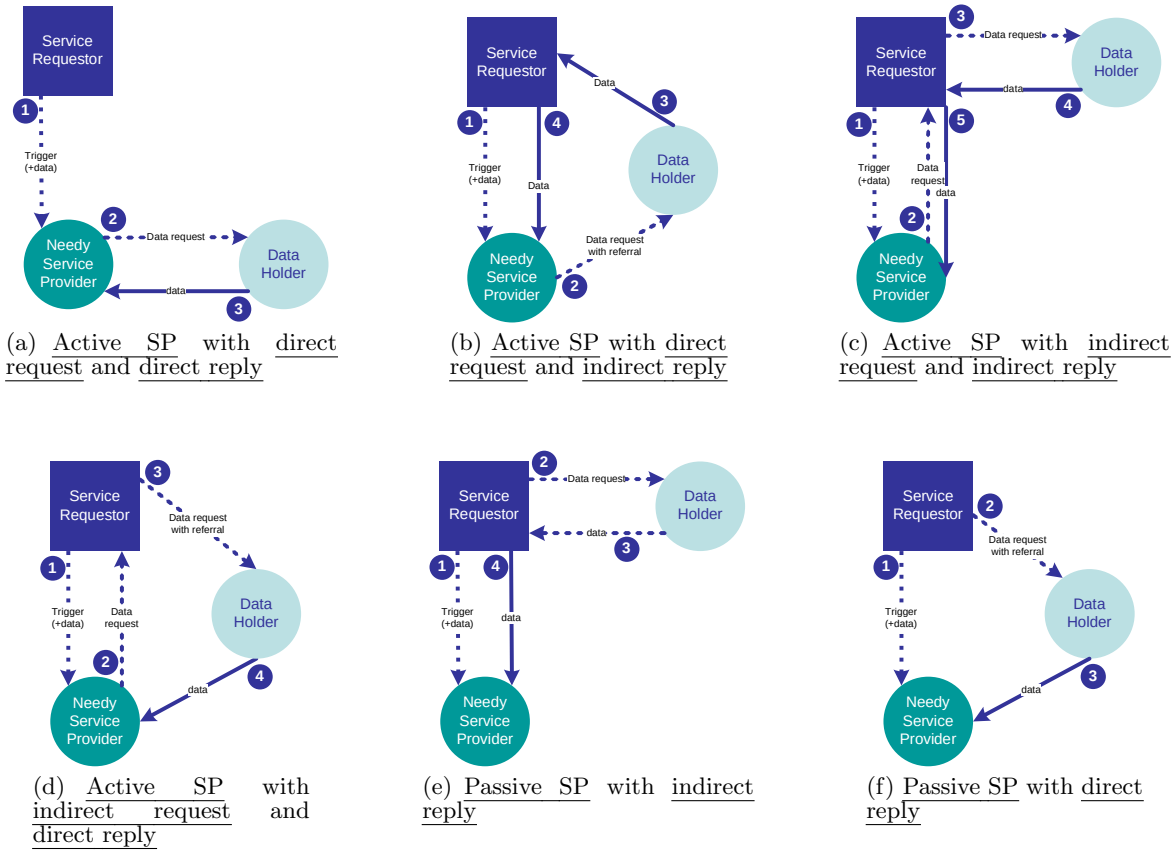


Figure 7: Six possible solutions (SP = Service Provider)

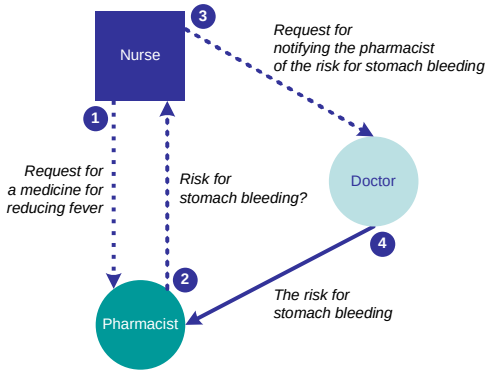


Figure 8: Active pharmacist with indirect request and direct reply

Subsequently, the authors present several patterns for the payment composition (e.g. payment in advance or afterwards). It should be clear that these kinds of patterns are only useful in a limited set of service composition problems. By discussing more *generic* service and data holders, the pattern language presented in this article is intended for a much broader use.

As described in the introduction (see subsection 1.2) our pattern language can support the process of designing the *service interactions* that are needed to combine services into a new composite service. In particular, we presented three kinds of *service interaction patterns* for requesting the data holder and subsequently transmitting the data from the data holder to the service provider. In that context the research in this article can be related to the often-cited work of Barros, Dimas and Ter Hofstede [1]. In their work they present about thirteen service interaction patterns. A large part of the set of patterns correspond to elemantary interactions where a party sends a message (a *request*) to another party, and as a result a *reply* is expected. The receiver of this reply can be the same as the sender of the request or not. The first group of interactions is referred to as round-trip interactions, while the latter type is named routed interactions. As mentioned in the context description of this pattern language (see subsection 2.1), a request should be sent to the data holder, before any data can be provided. As such, the round-trip and routed interactions can be part of a solution to the problem of input data provisioning. When considering a direct request and direct reply, part of the solution matches the *send/receive* pattern as described in [1] (see figure 7(a)). The same service interaction pattern can also be found in the scenario represented in figure 7(e). Clearly, in the solutions represented in figures 7(d) and 7(e) the *request with referral* pattern is applied, because the service requestor requests the data holder to send the response to the service provider. Scenarios that contain an indirect request (see figures 7(c) and 7(d)) can be related to the *relayed request* pattern, since the data request sent to the service requestor is relayed to the data holder. While the pattern described in [1] can be very valuable for constructing choreographies and evaluating service composition languages, there are no specific guidelines mentioned for when to use a specific service interaction pattern. In this paper, several service interaction patterns as described in [1] are applied in the context of input data provisioning. Furthermore, each pattern in the pattern language discusses some forces that are taken into account for constructing the most appropriate solution.

The results of this article can be considered as an extension of the work presented in [6]. The authors of that article propose to make a distinction between the *logical dependencies* that are modelled by the interaction logic from the *operational coordination* that refers to the procedure or method that is utilised to enforce the logical dependencies. Therefore the authors suggest that a technical solution for service composition should consist of a combination of design- and implementation patterns. A design pattern corresponds to the interaction logic that only specifies the generic process characteristics, while an implementation pattern refers to the the refinement of the interaction logic that is needed for the concrete coordination of services. In the context of this article, the design pattern consists of *triggering the service provider*, *requesting the data holder* and *sending the data to the service provider*. These different steps are needed to provide the composite service. In [6] it is said that the criteria for the choice of the most appropriate coordination pattern must be specified by so called coordination policies. A coordination policy describes the effect of a coordination variant in terms of specific (non-functional) service properties and thereby controls the choice of alternatives. In that way, the different patterns presented in this article, including the forces and the several combinations that can be made by the patterns, can be considered as coordination policies. They support the transformation of the interaction logic needed for input data provisioning into concrete coordination processes. In summary, one can state that our pattern language extends the work presented in [6] by proposing some concrete coordination policies (for input data provisinoning).

Acknowledgements

The authors would like to thank Pam Rostal for her many constructive and valuable remarks during the shepherding process. Furthermore the quality of this article has increased substantially after the conference's writers' workshop. The authors thank the workshop participants for their extensive, constructive and helpful discussion of a previous version of this article. In completing this paper, the authors benefited from the comments of several colleagues: Lotte De Rore, Raf Haesen and Pieter Hens. We also thank Marijke Braeken for her inspiring suggestions. This article has been written as part of a project funded by the Research Fund K.U.Leuven (OT 05/07 and IOF HB/07/022), whose support is gratefully acknowledged.

7. REFERENCES

[1] A. Barros, M. Dumas, and A.H.M. Hofstede. Service interaction patterns. *Lecture notes in computer science*, 3649:302, 2005.

[2] T. Erl. *Service Oriented Architecture: Principles of Service Design*. Prentice Hall, Boston, 2008.

[3] G. Monsieur, L. De Rore, M. Snoeck, and W. Lemahieu. Handling Transactional Business. *Proceedings of the 15th Conference on Pattern Languages of Programs (PLoP)*, 2008.

[4] M.T. Tut and D. Edmond. The use of patterns in service composition. *Lecture notes in computer science*, pages 28–40, 2002.

[5] U. Zdun, M. Voelter, and M. Kircher. Design and implementation of an asynchronous invocation framework for web services. *Lecture notes in computer science*, pages 64–78, 2003.

[6] C. Zirpins, W. Lamersdorf, and T. Baier. Flexible coordination of service interaction patterns. In *Proceedings of the 2nd international conference on Service oriented computing*, pages 49–56. ACM New York, NY, USA, 2004.