Design Patterns for Agent-Based Service Composition in the Web

Ingo Müller, Peter Braun, Ryszard Kowalczyk
Swinburne University of Technology
Faculty of Information and Communication Technology
Hawthorn, 3122 Victoria, Australia
{imueller, pbraun, rkowalczyk}@ict.swin.edu.au

Abstract

Service-oriented architectures have been recognized as advantageous architectural styles for future enterprise and scientific applications. However, on top of already available middleware layers, many problems regarding services engineering and management have been identified as open issues. The integration of agent-based approaches can provide promising solutions in order to overcome these prevailing obstacles. This paper identifies two agent-based design patterns based on a classification scheme outlining several ways for agent-service integration in the context of agent-based service composition.

1. Introduction

Service-oriented architectures, such as Web services and Grid services, have been recognized as advantageous architectural styles for future enterprise and scientific applications. However, on top of already available middleware layers, many problems regarding services engineering and management have been identified as very challenging and most of them still remain open. For example, the service community faces challenges such as missing semantic integration of functionality, inflexible interactions, inflexible workflows, as well as low robustness that all need to be overcome in order to achieve adaptive, flexible, and fault-tolerant compositions of multiple services in the context of uncertain environments.

As part of an on-going project we are working on novel techniques for adaptive service agreement and process management (ASAPM). In this project we employ software agents for service discovery and composition, as well as workflow enactment and monitoring. A clean and robust integration of service-oriented and agent-oriented paradigms is of major importance in our project.

The agent community has intensively debated how agent and service-oriented paradigms can be integrated in various application domains. Despite a few extreme views, in which the concept of software agents is expected to be absorbed by future web service models [9], several research groups argue in favor of integration approaches [13, 11, 17]. After our intensive literature review, we found that design decisions of the selected integration approach have been adequately justified and described only in a few papers. In contrast, a straightforward two-layer approach has been chosen in most papers, in which software agents form the upper and service components the lower layer.

This paper reports on the results of our intensive literature review. We have identified three classes of integration approaches, which we describe and discuss in the following section. After that, we focus in Sec. 3 on the most promising integration class and pinpoint two design patterns for the integration of agents and services. Design pattern are a common means in software engineering to describe best practices. Following other examples for the description of design pattern, such as [20], we apply a description template including problem description, application context, solution, and known uses. Finally, the paper is summarized and an outlook is given in the last section.

2. Integration Classes

This section outlines a classification scheme that identifies three different classes for the integration of software agents and services (as depicted in Fig. 1). Our scheme purely bases on the two fundamental components that underlie the paradigms namely software agents and service instances, and their particular characteristics. Agent-service integration can also be discussed on a system level, which is out of scope here.

2.1. Adjacent Integration Class

Adjacent Integration is the simplest integration form. Software agents and service instances coexist in their specific environments in parallel and interchange messages via...
defined interfaces. Thus, the integration is only achieved through communication. The strict separation of both component types might be a consequence of different concerns addressed by them. Software agents provide a sophisticated way to program intelligent software whereas services address standardized access to loosely coupled pieces of functionality. Adjacent Integration can be achieved in several ways as shown in Sec. 3.

2.2. Composite Integration Class

This integration bases on the formation of a new component type comprising of an agent and a service sub component (Fig. 1b). The agent and service sub-components are responsible for different tasks due to separation of concerns and different capabilities they provide. Typically, the service sub-component can provide mechanisms for describing and binding to the actual business functionality. The agent part is responsible for the management of interactions with other components. A composite component offers two logically separated interfaces as consequence of the strong separation between the two sub-components.

2.3. Affiliated Integration Class

The creation of a single new component type that combines concepts of the agent and service capabilities inside one component and offers one uniform interface is the strongest form of integration (Fig. 1c). Due to the characteristics of both paradigms it is most likely to change current services standards (e.g. substitute central UDDI registry with a decentralized agent-based approach, extend WSDL for expressing knowledge and state, or send ACL messages via SOAP) to level the way for using augmented reasoning, planning, and negotiation capabilities with services.

2.4. Discussion

To the best of our knowledge no work does exist that addresses fundamental integration aspects of agents and services in a structured way. However, we believe that a discussion of different approaches is necessary in order to obtain good design and to derive best practices. We have identified three different integration classes so far that vary in the degree how the two fundamental paradigms are integrated. Due to the different character of the three classes they have different impact.

Composite Integration for example is not mentioned in literature so far. It seems to be rather difficult to maintain two different paradigms in one single component because inherent differences between agent and service paradigm complicate the implementation.

Affiliated Integration is discussed in basic forms in [22, 1]. Moreau et al. [19, 4] had a look at techniques to closely interleave agents and services when they evaluate the mapping of ACL communication onto several transport layers such as SOAP or RMI. Additionally, it is depicted how agent behavior can be described and discovered using Web services standards. However, the consequent realization of this approach will most likely lead to the creation of new standards and notions, which restrict the reuse of existing tools and implementations.

The Adjacent Integration class is widely accepted in the research community and implemented in current state-of-the-art prototypes [2, 6, 5, 7, 18, 10, 21]. The reason for the strong use of Adjacent Integration may be found in (i) the simplicity of the strategy, which implies low implementation costs, (ii) the attempt to keep with current standards and notions, and (iii) the strong relations to the Semantic Web. The nature of Web services standards is to provide a uniform interface to functionality that is therefore a good candidate to form a bottom layer in a software design in order to integrate legacy and backend software in new solutions. On the other hand service notions lack mechanisms for composing and enacting services as well as monitoring the enactment. Thus it makes sense to delegate those tasks to software agents. Although, adjacent or layered integration seems to be a very simple approach, several design patterns can be recognized that e.g. differ in varying relations between service and agent instances or agent role modeling.

3. Agent-Service Design Patterns

The agent research community has realized for a long time the value of design patterns as means to utilize the de-
The focus of most current approaches of agent patterns or agent pattern languages is usually centered on facilitating the internal design of agents (e.g. [15, 14]), the design of multi-agent systems (e.g. [3, 8]), or the design process itself [23, 16] based on the assumption that a designer already has decided to use software agents.

In contrast, Weiss [25] proposes a pattern language that aims at supporting designers to make a decision towards modeling and implementing a given problem using agent technology.

However, most patterns or pattern languages cannot deny the strong influence of object oriented design patterns as for example proposed by Gamma et. al. [12]. On the other hand we have not found any literature proposing agent patterns for the design of agent-based service compositions on the Web. Of course, similarities of our contribution to existing work can be found (agent as delegates or agents as mediators). However, those similarities are vague. Current propositions are too general [16], too trivial [3], too context-specific [24], or strongly related to particular design methodologies such as for example the classification scheme of Lind to MASSIVE [16]. For that reasons they are hard to apply to the specific problem of Web service discovery, composition, and enactment.

In addition, almost all approaches do not consider agent specific characteristics such as autonomy or pro-activity except Lind who denotes specific fields such as autonomy or reasoning capabilities i.e. in the architecture view of his pattern description scheme [16].

Finally we would like to mention Oluyomi et al. [20]. The authors provide an agent pattern classification scheme that we want to align our pattern descriptions to. Although particular agent-specific characteristics have been left out, we will make use of it because it seems to be most promising and mature. We enrich the given scheme with additional information about agent autonomy, reactivity, pro-activity, and social behavior in order to address at least four fundamental agent characteristics.

3.1. Service Client Pattern

Name – Service Client, Service Mediator
Aspect of Design – Structural
Level of Design – System Design
Example – End-user applications that offer single or composite services to customers such as e-commerce applications, e.g. travel booking or online shopping. The application should not be limited to a fixed number of known services but implement an agent, which is able to compose multiple services through matchmaking based on a pre-defined workflow plan as well as finds and binds to concrete Web service instances dynamically.

Context – Service Client pattern can be applied for creating applications that flexibly discover, enact, or also compose Web services. The software needs to have planning and/or reasoning capabilities for matching service descriptions and/or refilling generic workflow plans with concrete service due to a loose coupling between services and client agent. In contrast to classical SOA applications the software is not limited to a specific set of Web services at design time but able to locate services that conform with a particular ontology dynamically.

Problem – A software agent acts as a client for one or multiple Web services.

Solution – The Service Client pattern is a representative of the Adjacent Integration class. An agent interacts with one or multiple services. Agents and services, both are separated and reside in their particular environments. The agent acts as a client for the services. The agent possesses knowledge to find and bind concrete Web services. The relation between agent and service is not static but dynamic and will be established during runtime.

Agent Autonomy: The agent is partially autonomous since it needs to be triggered by a request from another agent or user.

Agent Reactivity: The agent reacts on initial triggering by another agent or user and possible execution errors during service enactment.

Agent Pro-activity: After being assigned the agent is responsible to find and bind appropriate service in order to satisfy the request. We imply an agent to have understanding of how to interact with the Web services it is programmed to access.

Agent Social-Behavior: Limited. Depends on the complexity of the implementation. If the agent simply receives a WSDL description of services to be executed the approach is relatively straightforward and does not need social interactions. If the agent is provided only with key words and input data, it might request additional information from the requester, e.g. for effectively matching services from the given set of keywords.

Dynamics – The two participating instances are organized in a strictly layered way with the communication flow from...
the agent in the upper layer down to the service instance in the lower layer. The agent first queries a service description repository (e.g. extended UDDI registry) to gain knowledge about services that potentially satisfy the given requirements. In the next step the agent binds to a Web service using SOA standards such as XML-RPC or SOAP and requests the service. After receiving a result the agent can convert it if necessary and send it back to the agent or user that actually requested the task. In case an error occurs during service execution the agent can request another service in order to be able to provide a result.

Figure 2. Service Client Pattern.

Variants – The Service Client pattern can be implemented using different cardinalities. An agent can be associated with one or multiple services depending on the complexity of the agent implementation. On the other hand a service can be bound and accessed by multiple agents as well. In addition the pattern is applicable in both, single-agent and multi-agent systems in different levels of complexity. Thus one scenario could be the implementation of a single user agent that provides a meta e-shopping service that requests and compares prices of different e-shops and chooses the best bid in order to place an order.

Known Uses – [2, 6, 21]

Consequences – The agent is not statically associated with a service. The agent acts basically as a client and is therefore restricted to the capabilities provided by a service. The agent does not contribute additional capabilities to the service provision such as QoS negotiation, service monitoring, etc. The focus is strongly on the SOA paradigm. The agent is more or less an implementing unit for service-oriented applications.

See Also – The Service Client pattern can be combined with Broker Agents, Mediator Agents in multi-agent systems. It can be used to implement a specific user agent in a single-agent approach. In addition, a strong relationship to Semantic Web approaches cannot be denied.

3.2. Service Representative

Name – Service Representative
Aspect of Design – Structural
Level of Design – System Design

Example – End-user applications that offer single or composite services to customers such as different types of e-commerce applications, e.g. travel booking or online shopping. The application should not be limited to a fixed number of known services but implement agents, which are able to compose multiple services according to functional requirements and QoS parameters dynamically.

Context – Service Representative pattern can be applied for creating applications that flexibly locate, enact, and compose Web service instances. The software needs to have planning and reasoning capabilities for service composition and in addition the ability to negotiate and contract particular services prior to enactment. Representative agents engage in negotiations according to the status of their associated services. Service composition is achieved through agent coalition formation. The software bases on a close coupling between agent and service and is not restricted to a particular set of services at design time.

Problem – A software agent acts on behalf of one or multiple Web service and offers its associated services together with additional capabilities (e.g. QoS negotiation) to other agents or users.

Solution – This pattern is a representative of the Adjacent Integration class. Agents and services are both separated and reside in their particular environments. An agent acts on behalf of one or multiple services and adds additional capabilities such as service monitoring, service negotiation, and active service advertising. The agent is supposed to keep track of service use and monitors service QoS parameters in order to engage in negotiations to optimize the service outcome or the service provider profit. The relationship between agent and service is typically fixed during design time. Service discovery is not compulsory the association is implicitly given but can be added for the sake of flexibility.

Agent Autonomy: The representative agent acts fully autonomous on behalf of a service/service provider. A service provider is able to adjust negotiation settings etc. However, this does not affect the agent’s autonomy.

Agent Reactivity: The agent reacts on initial triggers by
other agents or users and possible execution errors during service enactment.

**Agent Pro-activity:** After having established (active or passive) contact to other agents the representative agent engages in negotiation and coalition formation process according to the service’s negotiation settings, goals, and characteristics. We imply an agent to have monitoring capabilities for its associated Web services.

**Agent Social-Behavior:** Very important. Because the representative agent engages in coalition formation process and negotiations its social capabilities are crucial for the success of a service. The social behavior gets even more important the more complex the negotiation and coalition formation interactions are. In case of misunderstanding, ambiguity, or unexpected responses it can happen that agents request additional information from communication partners in order to solve a given problem.

**Dynamics –** The two participating instances are organized in a strictly layered structure with the communication flow from the agent in the upper layer down to the service instance in the lower layer. After other agents got in contact with a representative agent the latter checks functional and QoS requirements of the requested service, engages in negotiation or coalition formation. For all activities the agent can call functions of its associated Web services both, functional and management-related. After the representative agent agreed on a contract it requests the service immediately. In case the representative agent participates in a service composition it waits for enactment initiation.

**Known Uses –** [10]

**Consequences –** The agent acts on behalf of a service and is therefore statically associated with it. Thus, the agent can add additional capabilities to pure service functionality such as QoS negotiations and active advertising. This pattern provides a powerful means for complex service compositions. The focus of this pattern is clearly on the agent paradigm with services solely providing basic functionality.

**See Also –** The Service Representative pattern can be combined with Broker Agents, Mediator Agents, and UserAgents in multi-agent systems.

4. Conclusion

We identified a classification scheme comprising three different classes for agent-service integration in the context of agent-based service composition. We discussed the three approaches and argue that one approach is not used so far, one approach is hard to achieve and therefore we focus on exploring the third, the Affiliated Integration class. This general scheme can be refined by two different patterns (Service Client and Service Representative), which we specify in Sec. 3. The focus of this work is rather on stimulating discussion than providing a complete formally specified catalogue of patterns. This paper marks an early stage of our project that aims at implementing software for agent-based service composition. We plan to evaluate the presented findings in that project.

5. Acknowledgements

This work has been partially supported by DEST in the context of the project Adaptive Service Agreement and Process Management (AU-DEST-CG060081).

References


