USING UML 2.0 IN SERVICE-ORIENTED ANALYSIS AND DESIGN

Ing. Petr WEISS, Doctoral Degree Programme (1)
Dept. of Information Systems, FIT, BUT
E-mail: weiss@fit.vutbr.cz

Supervised by: Dr. Jaroslav Zendulka

ABSTRACT

This paper deals with Service-Oriented Architecture Design (SOAD). I briefly discuss advantages of Object Oriented Analysis and Design (OOAD) in SOAD. The key activity in my work was modeling components, services and service choreography in UML 2.0 [3].

1 INTRODUCTION

Although service-oriented architecture (SOA) is not a new concept in the area of distributed software architectures, it comes to the foreground in recent years thanks to modern technical solutions, such a well-defined communication networks and modeling disciplines. SOA implementation rarely starts on the green field that means creating a SOA solution is almost based on integrating existing systems by decomposing them into services, business processes, and business rules. Consequence of this is the SOAD composition of well-established practices such as OOAD, Enterprise Architecture (EA), Business Process Modeling (BPM) and some other innovative elements. This paper describes how can be OOAD, especially UML, used by modeling components, services and service choreography.

2 SERVICE-ORIENTED ARCHITECTURE

Since there are not any standards, which will exactly define the meaning of SOA, we can say, that SOA presents an approach for building distributed systems that deliver application functionality as services to either end-user applications or other services.

It is evident that the key element of SOA is a *service*. Services are loosely coupled software entities with well-defined, published *interfaces*. The service interface separates provided functionality from its implementation (services are implementation-independent) and forms so-called service description. The service description is available for searching, dynamic binding and invocation by a service consumer. The communication between services is based on *message sending*.

On a higher level of abstraction, services can be composite into a *business process*. In SOA terms, a business process consists of a series of operations which are executed in an

ordered sequence according to a set of business rules. The sequencing, selection, and execution of operations are termed *service choreography*. Typically, choreographed services are invoked in order to respond to business events.

On the other hand, service is a composition of *components*. Rather, service encapsulates interfaces of collaborating components. Components are responsible for providing service‘s functionality and maintaining its quality of service.

The above mentioned abstract view of SOA is depicted on Figure 1 as a partially layered architecture.

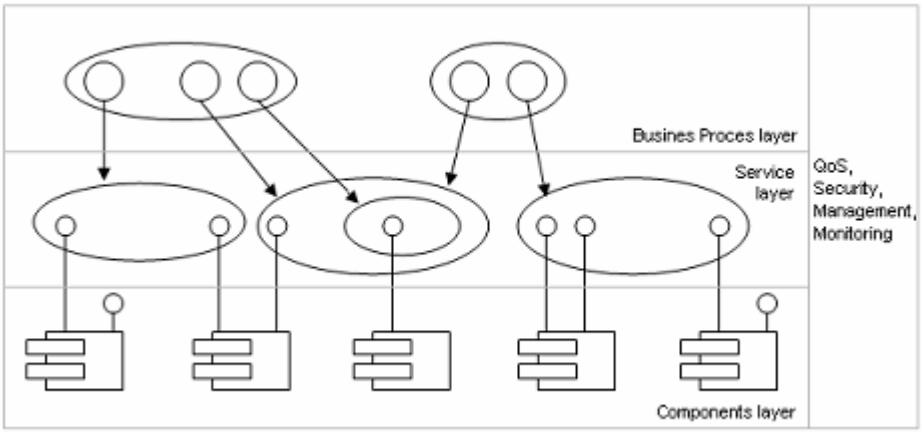

Fig. 1: *The layers of a SOA*

More SOA’s principles are described in [2].

3 OBJECT-ORIENTED ANALYSIS AND DESIGN

Object-Oriented Design is an approach in which a system is modeled as a collection of objects. The behavior of the system is achieved through collaboration between these objects, and the state of the system is the combined state of all the objects in it. Collaboration among objects allows them sending messages to each other.

OOAD is a very powerful and proven method, and that is why SOAD should use of OOAD techniques as much as possible. The fundamental principles of Object-Orientation (OO), which can be used in SOAD, are following:

- Information hiding: Well-structured objects have simple interfaces and do not reveal any of their internal mechanisms to the outside world. SOA’s components are black boxes with well-defined interfaces.
- Messaging makes objects communicationable with each other. As mentioned before, messaging is the fundamental communication model for services in SOA.
- Inheritance is used to derivate new (sub)classes from a general (super)class. A subclass includes all the data and methods of all of its superclasses, but it can add some other or change them.

- Polymorphism describes the situation where the result (of a behavior) depends on in which class the behavior is invoked. In other words, two or more classes accept the same message, but respond on it differently. Maybe it could seem to be a problem, because services are loosely coupled and they are not addressed directly. They are discovered by the means of „what to do“. However, polymorphism is closely associated with inheritance and I do not consider inheritance of services in my work, moreover, each service has well-defined description, messages cannot be misinterpreted.
- Classes and instances: Classes are templates for creating objects. Classes define the kind of properties and behavior of a given object. Instances are the individual objects that hold values for those properties. From the point of view of a SOA, such a concept can be applied only by modeling components, because services are not instanceable.

The last important property of OO is Encapsulation. Encapsulation means wrapping up of data and associated functions into a single unit (object). Thus, the unit is accessible only through its interface. This kind of property cannot be used by modeling services in SOAD, because services only provide functionality, data are passed through messages.

One of the underlying differences between OOAD and SOAD is that SOAD is generally no longer “use case-oriented”, but driven by business events and processes. Use case modeling comes in as a second step on a lower level.

4 USING UML IN SOAD

In this chapter I discuss the key activity of my work – using current existing tools (in this case UML) as much as possible for modeling components, services and service communication and choreography. As mentioned before, object-oriented technology and languages are great ways to implement components.

4.1 COMPONENTS AND SERVICES

Modeling components is pretty easy, because UML provides standard stereotype <<component>>. A <<component>> is a self contained unit that is able to interact with its environment through its *provided* and *required* <<interfaces>>. A component can be replaced at run-time by a component that offers equivalent functionality based on compatibility of its interfaces. For modeling components it is used external view (or “black-box” view) that hide the inner structure of <<component>>.

A service is modeled as a stereotype <<service>> which is derived from <<component>>. <<service>> compared to <<component>> represents the internal view (or “white-box” view). This view shows how the external behavior is realized internally. In this case, the inner structure depicts how instances of components are interconnected to provide service’s functionality. An example of this interpretation is shown on Figure 2. If it is necessary, the behavior can be specified more detailed using an interaction diagram. The mapping between internal and external view ensures a delegation connector. A delegation connector is represented by a port and a relationship-stereotype <<delegate>>. The port is shown as a small square symbol on the boundary of the rectangle symbol denoting a <<service>>.

A port delegates to a set of subordinate components (and vice versa). At execution time, signals will be delivered to the appropriate instance. In the cases where multiple target

instances support the handling of the same signal, the signal will be delivered to all these subordinate instances.

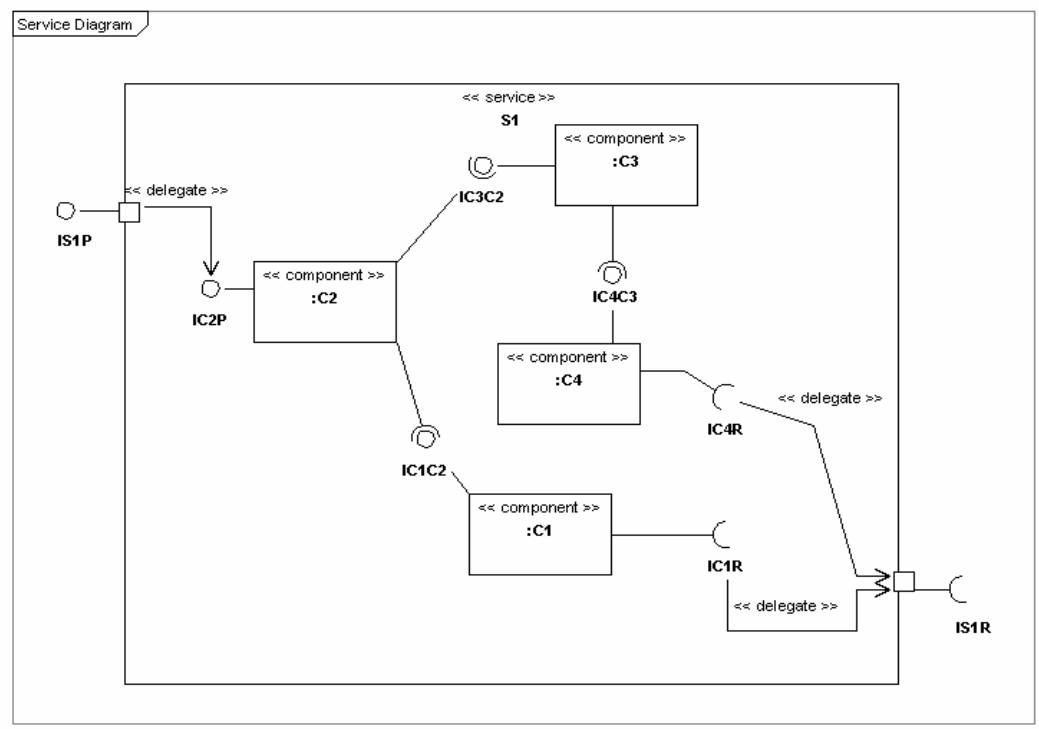

Fig. 2: *Service Diagram*

If we want to model a service as a composition of two or more services we cannot use the above-mentioned concept (<<service>> stereotype), because among services is another type of communication than among components. I solved this problem by using packages. Packages encapsulate both structure and communication. Such a package is a model of a composite service and, of course, has a provided and a required interface.

4.2 COMMUNICATION

As we already know, communication among services is based on message sending. For modeling this principle UML uses sequence diagrams. Since the modeled entities are services, the diagrams have to fulfil following restrictions:

- Services are not instanceable, therefore in the diagram cannot be create- and destroy-elements.
- Lifelines belong to interfaces of a given service.

A sequence diagram can also express the princeple of observer design pattern, which is described in the following paragraph. Such a diagram depicts Figure 3b.

A modification of the observer design pattern helps us to model such a situation, where a service accepts multiple requests and answers on them one by one. Such a model (Figure 3a) consists of two super classes: Provider and Consumer, which define necessary methods for interfaces of concrete services. Provider holds list of incoming requests. Each Consumer registers itself by invocating method Attach(service, id). Detach() serves Consumer to erase itself from the list of observers. When Provider completes the request processing, the method Update() is invoked. This method notifies the Consumer of available request-data. The Consumer can take this data through calling the SendResult(id) method.

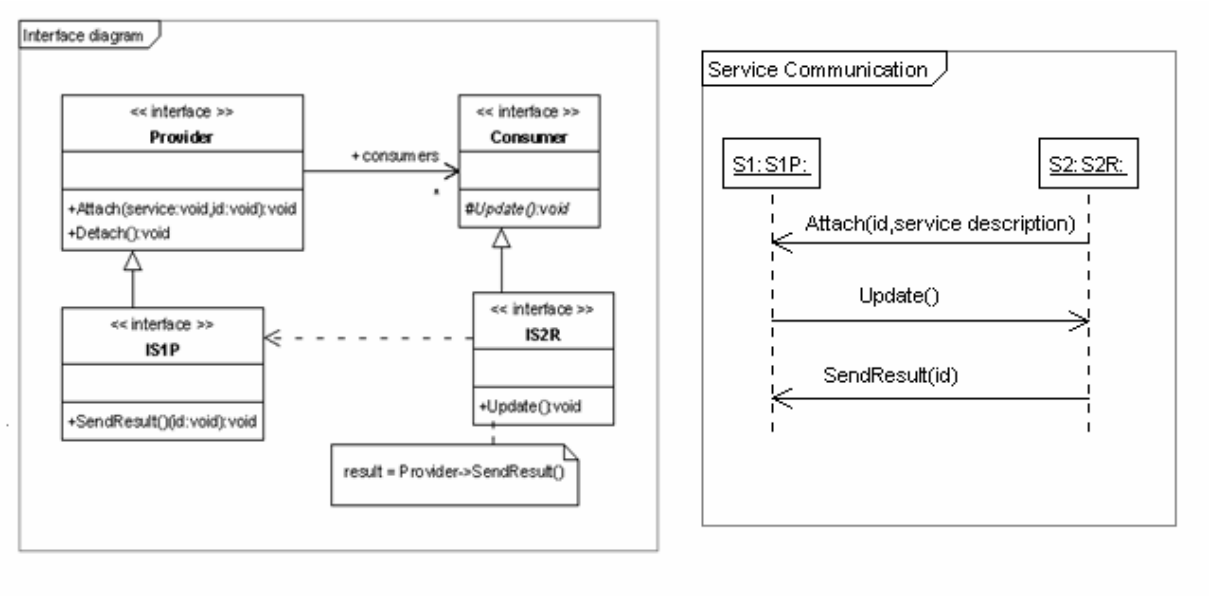

Fig. 3: *a) Observer design pattern in Interface Diagram; b) Sequence Diagram*

5 CONCLUSION AND FUTURE WORK

This paper presents an approach for using object-oriented methods in service-oriented design. Unified Modeling Language approved itself as a powerful tool for modeling components and services. It provides also well-defined diagrams for modeling communication among services, but modeling choreography is a little problem. My future work will focus on improvement and extension of the approach mentioned above. Furthermore, I will integrate OO methods with other well-established modeling methods, such a Business Process Modeling.

REFERENCES

[1] Booch, G., Rumbaugh, J., Jacobson, I.: Unified Modeling Language User Guide. Addison-Wesley 1999. ISBN 0-201-57168-1

[2] Endrei, M., et al.: Patterns: Service-Oriented Architecture and Web Services, IBM 2004, ISBN 073845317X. URL: http://www.redbooks.ibm.com/redbooks/pdfs/sg246303.pdf (February 2006)

[3] Unified Modeling Language - UML 2 Superstructure, Document is available on URL http://www.omg.org/technology/documents/formal/uml.htm