ForeverSOA: Towards the Maintenance of Service Oriented Software

Dionysis Athanasopoulos   Apostolos Zarras

ForeverSOA Associated Team - DMOD
Department of Computer Science
University of Ioannina - Greece
{dathanas, zarras}@cs.uoi.gr

Valerie Issarny

ForeverSOA Associated Team -ARLES
INRIA - UR Rocquencourt
France
Valerie.Issarny@inria.fr

Abstract

In this position paper, we argue about the need to adapt/refine fundamental object-oriented design principles with respect to the specificities of service-oriented software, to address realistic maintenance scenarios. Moreover, we sketch an approach that relies on a reverse engineering process, which recovers service abstractions out of available services, to enable the adoption of the refined principles in the development of service-oriented software towards improving its maintainability quality attribute.

1. Introduction

Service-Oriented Architecture (SOA) is an architectural style that emerged as the answer to the latest requirements for loosely-coupled distributed computing [1], [2]. Inline with the conventional distributed computing paradigm, functionality is decomposed into distinct architectural elements, distributed over the network. Nevertheless, in SOA the basic architectural elements (a.k.a services) are by themselves autonomous systems that have been developed independently from each other.

Until now, state of the art research in SOA systems has focused mostly on issues concerning the construction phase of service-oriented software. The outcome of these research efforts was mechanisms for discovering, composing and accessing available services (e.g. [3, 4, 5]). However, several other phases of the development process are currently underdeveloped. In this paper, *we focus on the maintenance phase of service-oriented software*, i.e., software built by composing services. Specifically, we concentrate on the *maintainability quality attribute*. The importance of this issue is evident towards the success of the SOA paradigm [6], which promotes the development of software consisting of independently evolving basic engineering elements that may further vary in quality (e.g., performance, availability, reliability).

In conventional Object-Oriented (OO) software, maintainability can be improved by employing well known fundamental design principles such as OCP (Open Closed Principle) [6], DIP (Dependency Inversion Principle) [7] and LSP (Liskov Substitution Principle) [8]. In this paper, we revisit these principles in the context of the SOA paradigm and argue about the need to adapt/refine them to the specificities that characterize the paradigm. Specifically, the contribution of this paper is threefold:

1. We examine the maintenance scenarios that can be handled by the conventional use of the fundamental design principles in the SOA paradigm and discuss why these scenarios are not realistic.

2. We adapt/refine the fundamental design principles such that their use in service-oriented software becomes effective towards handling realistic maintenance scenarios.

3. We sketch the ForeverSOA infrastructure, which aims at facilitating the adoption of the refined principles in the development of SOA software. The prominent concept in ForeverSOA *is a reverse engineering process that recovers service abstractions out of available services*. An abstraction characterizes a group of services, providing similar functionalities via different interfaces and serves for developing software that may access any of the grouped services without depending on their interfaces.

The remainder of this paper is structured as follows, Section 2 discusses the conventional use of OCP, DIP and LSP. Section 3 proposes the refinement of these principles and provides an overview of ForeverSOA. Finally, Section 4 gives a summary of this paper.

2. Maintenance Scenarios in SOA

In the OO paradigm, OCP [7], is the key principle that concerns the maintenance of OO software.

According to OCP *software should be open for extensions and closed to modifications*. In other words, it should be possible to change elements of a given software without modifying the code of the remaining software elements. Achieving OCP is typically based on further related design principles and in particular LSP [9] and DIP [8]. LSP formalizes the basic correctness criteria that guarantee that software that uses elements of a particular type can further use elements of another type. DIP states that *higher level software elements should not depend on lower level software elements; they should both depend on abstractions.*

Employing the aforementioned principles in the development of OO software typically involves performing the following development steps:

STEP 1. Define an abstraction element (e.g., an abstract class or an interface) for each software responsibility that may be subject to changes.

STEP 2. For every abstraction element, develop a derived hierarchy of concrete implementations, with respect to LSP.

STEP 3. As suggested by DIP, develop the rest of the software such that it uses references to abstraction elements instead of references to concrete implementation elements.

In the SOA paradigm the situation is more complicated but still the fundamental design principles can be applied in the conventional way. In particular, *STEPs 1 & 2* are performed by service providers, independently from *STEP 3* which takes place on the side of service clients. Typically, we cannot assume a close collaboration between the development teams of the two parties. During the whole lifecycle (design, construction, testing, and maintenance) of services, service providers are usually unaware of the client applications that use them, making the boundaries of service-oriented software weakly defined.

Specifically, during *STEP 1*, the development team of a service provider defines an abstract type in SOA terms, i.e. a service interface. In the standard W3C language for the specification of service interfaces (WSDL) an interface is defined in terms of a *port type* which consists of a set of *operations*. An operation accepts at most one *input message*, and produces at most one *output message*. Moreover, the operation may possibly generate a number of fault messages that signify erroneous situations.

During *STEP 2*, the development team of a service provider realizes the service interface specified in *STEP 1*. In particular, the development team provides an implementation in a conventional OO language (e.g. Java, C++). Following, the service implementation is

deployed in an application server, which comprises the necessary functionality that facilitates interaction between the potential client applications and the service. Finally, the service interface description is complemented with certain additional binding information and registered in a service registry/catalog so that it can be discovered by the potential clients.

During *STEP 3*, the development team of a client application looks in a service registry/catalog for a service that can be used in the application. The application must be developed with respect to an interface reference to the service. This reference is bound to the service implementation based on the binding information found in the service registry. Finally, the interface reference is used throughout the application code to invoke operations on the service implementation. Typically, the service invocations are realized via a standard RPC middleware mechanism such as JAXRPC. Invocations may be either static or dynamic.

Concerning the OCP principle, the client application is open for extensions and closed to modifications in the following sense: If the requirements of the client application are no longer satisfied by the service implementation that realizes the service interface assumed by the client application, the development team of the service provider can *extend* the hierarchy of service implementations derived from the particular service interface with a new implementation that meets the new application requirements. Following, the service client development team can substitute the service implementation that is currently used for the new one, by rebinding the service interface reference to the new service implementation. The rest of the software remains *closed* to the particular modification (i.e., unchanged), since access to the new service implementation is based on the operations of the service interface.

Taking an example, suppose that we decide to develop an application that allows manipulating information about scientific publications, according to the SOA paradigm. In that case we can take advantage of a Web search engine for publications that is exposed through a programmable Web service interface (Figure 1). The interface provides an operation, named `cis`, which accepts as input a string parameter `query` that contains search terms and returns as output a string parameter `result` that corresponds to an HTML document, comprising information about publications that contain the given search terms. Note that the interface of the Web service is inspired by the front-

end Web interface of the Citeseer[1] publications search engine. If the requirements of the application are no longer satisfied by the Citeseer service (e.g., the service response time is not satisfactory), the Citeseer development team must *extend* the hierarchy of Figure 1 with a new Citeseer service implementation that is going to be used in the client application by simply rebinding the application to the new service implementation (Figure 1, lines 3, 4).

Concerning the aforementioned maintenance scenario, *the main observation is that it is not realistic in the practical sense*.
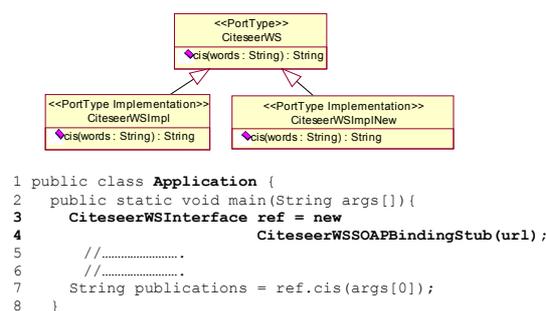
```
1 public class Application {
2   public static void main(String args[]){
3     CiteseerWSInterface ref = new
4                     CiteseerWSSOAPBindingStub(url);
5       //.........................
6       //.........................
7       String publications = ref.cis(args[0]);
8 }
```

Figure 1. Conventional maintenance scenario in SOA.

Requiring a service provider development team to extend the implementation hierarchy derived from a service interface with the addition of a new implementation that addresses the evolving requirements of every client application seems virtually impossible, especially considering the fact that typically service providers may not even know about which client applications are using their provided services.

In practice, dealing with a service that no longer satisfies the possibly evolving requirements of a client application amounts to searching in the service registry/catalog for another service that fulfils the application requirements. Discovering such a service with the additional requirement that its interface is the same with the interface of the service that is currently in use in the client application also seems far from being realistic.

Taking our example, the realistic approach for dealing the fact that the Citeseer service no longer meets the requirements of the application is to use another publications search engine such as

GoogleScholar[2]. In this case, the corresponding service offers an operation, named scholar, which accepts as input a string parameter q with search terms and returns as output a string parameter r (Figure 2). Apparently, the interfaces of the two services are different and this maintenance scenario can not be handled while keeping the application closed to modifications; several parts of the application code should be changed (Figure 2, lines 3, 4, 7). Hence, in this scenario there is not much benefit from the conventional adoption of the fundamental design principles.
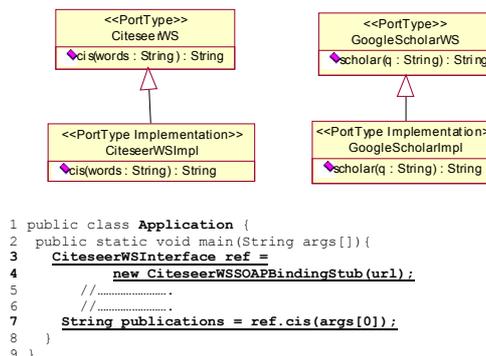
```
1 public class Application {
2   public static void main(String args[]){
3     CiteseerWSInterface ref =
4         new CiteseerWSSOAPBindingStub(url);
5       //.........................
6       //.........................
7       String publications = ref.cis(args[0]);
8 }
9 }
```

Figure 2. Realistic maintenance scenario in SOA.

3. Overview of ForeverSOA

Based on the discussion so far, the problem we are facing in the context of the SOA paradigm is to adapt/refine the fundamental design principles in a way that would render their adoption in the SOA paradigm beneficial towards handling realistic maintenance scenarios. Specifically, the ultimate goal is to be able to develop SOA software that complies with the *OCP principle* (i.e., it is open for extensions and closed to modifications) in the following sense:

- A client application that uses a particular service can be *extended* towards using another service that offers the required functionality possibly via a different interface.

- The extension of the client application involves *minimum modifications* in the client application code.

Achieving the refined *OCP* principle, involves refining the DIP principle as follows: *The client application code should not depend on a particular*

---

[1] http://citeseer.ist.psu.edu/

[2] http://scholar.google.gr/

service interface; they should both depend on abstractions. Defining a higher level of abstraction, beyond service interfaces and developing the client application code based on this level of abstraction is a quite straightforward concept [10, 11]. However, the *real challenge behind this concept is the provisioning of a systematic reverse engineering process that extracts service abstractions out of existing services that offer similar functionality via different interfaces*. The aforementioned *abstraction recovery process* should take place with respect to the *LSP* principle, which must hold for recovered service abstractions and the services that realize them.

Therefore, the main objective of ForeverSOA is to provide an infrastructure that would facilitate the development of SOA software based on the aforementioned abstraction recovery process.

Specifically, we envision the provisioning of a service *registry* that manages information about available services offered by service providers and facilitates the development of client applications that use these services (Figure 3). The registry organizes available services into *groups*. Each group is characterized by a *service abstraction* and a set of available *services* that realize this abstraction. Service abstractions are reverse engineered based on a *clustering mechanism* that accepts as input the overall set of available services and divides them into groups consisting of services that provide similar behavioral features (i.e., service operations). The representative of each group is the service abstraction that is realized by the group members. Currently, we focus on the customization of a classical agglomerative clustering algorithm [12] to the specificities of the SOA paradigm.
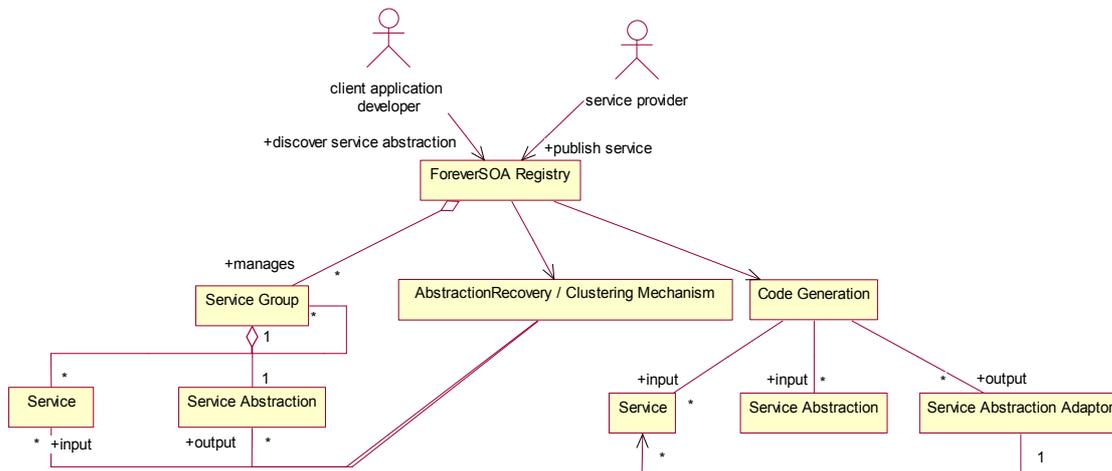

Figure 3. Overview of ForeverSOA

The output of the abstraction recovery process (i.e. the reverse engineered service abstractions) further serves as input to a *code generation mechanism* which produces (possibly with partial human intervention/inspection) typical *proxies/adaptors* that realize service abstractions and serve for accessing the services that realize these abstractions without depending on the services' interfaces. The well-known adaptor pattern has been used in several approaches for bridging the semantic gap between a service client and a service provider [10,11,13,14,15]. In the context of ForeverSOA, the challenging issue is to provide means for dynamically upgrading the adaptors used on the client application side according to the application

evolving requirements and the contents of the ForeverSOA service registry.

Taking our example, it is possible to reverse engineer the SearchEngine service abstraction given in Figure 4 by clustering similar behavioral features (i.e. operations) of the Citeseer and the GoogleScholar services of Figure 2. This particular abstraction offers an operation named search which accepts as input a string parameter containing search terms and produces as output a string that contains search results. Following, it is straightforward to generate a simple adaptor, which can be configured via the homonymous operation to translate invocations of the search operation into invocations to the cis and/or the

`scholar` operations, provided, respectively, by the Citeseer and the GoogleScholar services. Finally, the code of the client application is developed with respect to a reference to a search engine adaptor instance. Therefore, the application can use any of the two publications search services without code modifications.
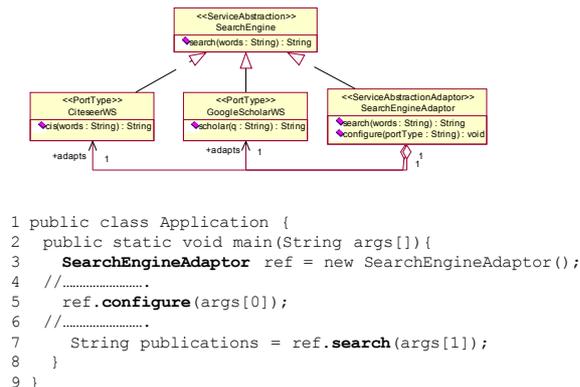


```
1 public class Application {
2   public static void main(String args[]){
3     SearchEngineAdaptor ref = new SearchEngineAdaptor();
4   //.....................
5     ref.configure(args[0]);
6   //.....................
7       String publications = ref.search(args[1]);
8   }
9 }
```

Figure 4. Abstraction recovery in ForeverSOA.

4. Conclusions

In this position paper, we investigated the maintenance scenarios that can be handled by the conventional use of OO design principles in the SOA software. We argued about the need to adapt/refine these principles, to address realistic maintenance scenarios. Finally we proposed an approach that relies on a reverse engineering process, which recovers service abstractions out of available services, to enable the adoption of the refined principles in the development of service-oriented software towards improving its maintainability quality attribute.

Currently, we are in the process of developing the main mechanisms of the ForeverSOA approach, while we also investigate the potential of employing further OO design principles in the context of SOA such as the Single Responsibility Principle (SPR [8]) and the Interface Segregation Principle (ISP [8]).

References

[1] E. Thomas, *Service-Oriented Architecture: Concepts, Technology, and Design*, Prentice Hall, 2005.

[2] J. Cardoso and A. Sheth, *Semantic Web Services, Processes and Applications*, Springer, 2006.

[3] S.-B. Mokhtar, A. Kaul, N. Georgantas, and V. Issarny, "Efficient Semantic Service Discovery in Pervasive Computing Environments", in *Proceedings of the 7th ACM/IFIP/USENIX International Middleware Conference (MIDDLEWARE)*, pp. 240-259, 2006.

[4] D. Berardi, D. Calvanese, G. DeGiacomo, M. Lenzerini, and M. Mecella, "Automatic Service Composition Based on Behavioral Descriptions", *International Journal of Cooperative Information Systems*, 14(4), pp. 333-376, 2005.

[5] J. Yang and M. Papazoglou, "Service Components for Managing the Lifecycle of Service Compositions", *Information Systems*, 29(2), pp. 97-125, 2004.

[6] G. Lewis, D. Smith and K. Kontogiannis, "SOAM 2008: 2nd Workshop on SOA-Based Systems Maintenance and Evolution", in *Proceedings of the 12th IEEE European Conference on Software Maintenance and Reengineering (CSMR)*, pp. 336-337, 2008.

[7] B. Meyer, *Object-Oriented Software Construction*, Prentice Hall, 1988.

[8] R. C. Martin, *Agile Software Development, Principles, Patterns, and Practices*, Prentice Hall, 2002.

[9] B. Liskov and J. Wing, "A Behavioral Notion of Subtyping", *ACM Transactions on Programming Languages and Systems (ACM TOPLAS)*, 16(6), pp. 1811-1842, 1994.

[10] L. Melloul and A. Fox, "Reusable Functional Composition Patterns for Web Services", in *Proceedings of the IEEE International Conference on Web Services (ICWS)*, pp. 498-506, 2004.

[11] Y. Taher, D. Benslimane, M.-C. Fauvet, and Z. Maamar, "Towards an Approach for Web Services Substitution", in *Proceedings of the 10th International Database Engineering and Applications Symposium*, pp. 166-173, 2006.

[12] O. Maqbool and H. Babri, "Hierarchical Clustering for Software Architecture Recovery", *IEEE Transaction on Software Engineering (IEEE TSE)*, 33(11), pp. 759-780, 2007.

[13] S. R. Ponnekanti and A. Fox, "Interoperability Among Independently Evolving Web Services", in *Proceedings of the 5th ACM/IFIP/USENIX International Middleware Conference (MIDDLEWARE)*, pp. 331-351, 2004.

[14] S. R. Ponnekanti, "Application-Service Interoperation Without Standardized Service Interfaces", in *Proceedings of the 1st IEEE International Conference on Pervasive Computing and Communications*, pp. 30-37, 2003.

[15] H. R. Motahari Nezhad, B. Benatallah, A. Martens, F. Curbera and F. Casati, "Semi-Automated Adaptation of Service Interactions", in *Proceedings of the International World Wide Web Conference (WWW)*, pp. 993-1002, 2007.

Acknowledgement