Services and Objects

Andrea, Vincenzo D'; Aiello, Marco

UNIVERSITY
OF TRENTO

DEPARTMENT OF INFORMATION AND COMMUNICATION TECHNOLOGY

38050 Povo – Trento (Italy), Via Sommarive 14
http://www.dit.unitn.it

SERVICES AND OBJECTS: OPEN ISSUES

Vincenzo D'Andrea and Marco Aiello

December 2003

Technical Report # DIT-03-085

Services and Objects: Open issues

Vincenzo D'Andrea and Marco Aiello

Department of Information and Telecommunication Technologies
University of Trento
Via Sommarive, 14 38050 Trento
Italy
{dandrea,aiellom}@dit.unitn.it

Abstract. One of the common metaphors used in textbooks on Object-Oriented programming (OOP) is to view objects in terms of the services they provide, describing them in "service oriented" terms. This opens a number of interesting questions, moving away from the simple view of OOP as an implementation tool for Web Services. First of all: if an Object is a Service, can we also say that a Service is an Object?
While the short answers seems to be negative, there are several connections between the two concepts and it is possible to exploit the large repository of methodological tools available in OOP. What are the counterparts, in terms of services, of concepts like class or instance? Is it possible to apply techniques as containment or inheritance to services? What are interfaces, properties and methods for services? In this paper we try to start building some connections, underlining the open issues and the gray areas.

1 Introduction

One of the common metaphors used in textbooks on Object-Oriented programming (OOP) is to view objects in terms of the services they provide, describing them in "service oriented" terms (see for instance [3]). Building on abstraction and encapsulation, the key idea is to hide programming details that provide object functionalities. An interface describes these functionalities in terms of methods and properties, providing a logical boundary between operations invocations and their implementations. Then an object is just a "server" of its own methods. If on the one hand, this view is useful for educational purposes, on the other hand, it represents only a minor feature when compared to inheritance, polymorphism, code sharing, and so on.

If the object oriented paradigm is already 'service oriented' why is it then that we talk about a *new* computing paradigm with the advent of web services? Objects in OOP are already described as services, so is it because of the gaining momentum of web services that one describes this new trend as a shift in computing paradigm? To answer this let us consider more precisely what a web service is and what we mean by service orientation. In [6], Curberra et al. describe a web service in the following way:

> A Web service is a networked application that is able to interact using standard application-to-application Web protocols over well defined interfaces, and which is described using a standard functional description language.

The interfaces no longer hide units of code, but entire applications in a way closer to components [12]. In addition, the network plays a major role, with the consequences that web services have to deal with issues typical of distributed systems [5], such as: heterogeneity, openness, security, scalability, failure handling, concurrency, transparency. Web services are shifting perspective on programming and are now calling for a new term for programming. There seems to be consensus on the term *service oriented computing (SOC)*. A definition of SOC is in the "Service Oriented Computing Manifesto" [7].

> Services are autonomous platform-independent computational elements that can be described, published, discovered, orchestrated and programmed using XML artifacts for the purpose of developing massively distributed interoperable applications.

The SOC definition above generalizes the one of web services. One does not distinguish anymore among applications or components, but simply deals with computational elements. The *find-bind-use* model can summarize the idea of describing, publishing, discovering, orchestrating and programming the distributed computational entities. Standardization is explicitly mentioned and referred to XML-based languages.

In this position paper, we indicate some areas where web services may be contaminated by concepts and ideas from the object-oriented paradigm. We will base our analysis on abstract object-oriented concepts trying to avoid language peculiarities and tricks.

2 Similarities and differences

To justify a call for a paradigm shift, there must be some significant differences between object-oriented programming and service oriented computing. What we consider to be the key differences, among the many ones over which much hype has grown recently, are the following three:

OOP		SOC
invoke	vs.	*find-bind-use*
shared context	vs.	*multiple contexts*
synchronous method invocation	vs.	*asynchronous message passing*

Find-bind-use is the heart of service orientation. A software entity that needs a service from another entity first searches for available services, then decides

on the basis of some parameter among the available ones and only then binds it in order to use it. On the other hand, in object-oriented programming there is no search for service, but direct method invocation. The method must be provided by an object running at invocation time. The find-bind-use model allows for greater flexibility, especially in distributed environments, opening the road for the choice of services based on non-functional requirements, such as those ensuring quality of services.

In OOP the execution context is typically shared among all objects. Usually, objects are written in the same language, run on the same memory space and live for the execution span of the same program. Recent extensions allow for the objects to be distributed (e.g., Java RMI) and to be written in different languages (e.g., CORBA). These extensions go in the direction of service orientation, where everything is distributed and services live in heterogeneous multiple contexts. The operating systems in which web services live, the languages in which they are written, the middleware used for interoperation is completely transparent in the SOC model therefore we speak of *multiple contexts* of execution for interactive web service.

Finally, the interaction between objects through method invocation, which can be seen as a message passing mechanism, is synchronous. In open distributed environments a more flexible communication mechanism is often necessary, that is, the *asynchronous* communication among the software entities contributing to a computation. An example of asynchronicity is when one interacts with a web service by including an appropriate XML request inside an email.

If the above are the key mechanisms that differentiate between OOP and SOC, one may wonder at what is the different forms of abstractions that one considers when looking at SOC. In [3], Budd indicates how OOP realizes various forms of abstractions. Let us compare these with the SOC case.

Composition is a central issue in service oriented computing. A large amount of effort in research and industry is devoted to service composition. Some define ways to design the composition of service (e.g., [4, 13]) while others define how semantically annotated services can be automatically composed (e.g., [10]).

In Object Oriented systems, composition is a design activity and it is mainly a problem of statically designing the proper architecture of the system. The situation in Service Oriented computing is radically different: a service can build its functionalities upon others, for instance an e-commerce purchase service could include the actual purchase service, the shipping service and the insurance service. The composed services are not statically designed, the services and the supporting infrastructure are designed in terms of dynamically discovering the other services they need to include. In other words, the service paradigm provides the capabilities for dynamic, run-time composition rather than requesting a statically planned architecture.

The dynamic nature of composition has several consequences. Negotiation and contractual agreements cannot be accomplished off-line, they have to be dealt with at run-time. The role of catalogs and the discovery mechanism have no counterpart in the world of objects and components.

Services demand a transition from static binding between objects or components that are to be integrated to the dynamic binding of services. From the point of view of the design there is the need of a transition from designing an architecture to designing the *enabling medium*, that is, the infrastructure for runtime composition.

In object oriented systems, the term **inheritance** is used to describe the mechanism allowing the derivation of a class from another one. One may even distinguish between several forms of inheritance. The most common form is *specialization*; a class is defined in terms of specialization of a second one – this is expressed by the *is a* relationships (a TextWindow is a Window, i.e., the TextWindow has all the properties and behaviors of the Window). Specialization implies a semantic coherence between the two classes, one class is called a subtype of the other. Otherwise it is just a subclass, where the meanings attached to the interface can change. It is obvious that while a subclass must have at least some code differences with respect to the original class, a class that inherits in the sense of subtype can leave untouched the implementation details of the inherited class. In other words, the subclass requires a syntactical match, while the subtype implies also a semantical match between the involved classes.

The concept of subtyping is also related to a common distinction made between what is sometime referred to as "true" inheritance versus interface inheritance. The former is used when a class presents the same external interface *and* has access to the code of the inherited class, that is, the subclass is a subtype unless it overrides the behavior of the inherited one. The term interface is used when a class has the same external interface of the inherited one, but it has no direct access to its code. In this case, it became a subtype only when the behavior of the inherited class is reproduced with the same semantics.

In terms of implementation, a simplifying model is to view inheritance as a special form of composition. Composition generally implies wrapping the interface of the included classes, and filtering the communication between these classes and the external world. Inheritance can be described as if the inheriting class incorporates (composes with) the inherited one, but without filtering the communication; the inherited class can be accessed directly. An object of the inheriting class responds to the same invocations as an object of the inherited class. If the subclass is also a subtype, the results will also be the same.

To think at inheritance (subtyping) as a form of composition which maintains the interface (behavior) of the composed object, makes it easier to reason about similar concepts in the service world.

In OOP, **polymorphism** indicates an operation that can take operands of different type, i.e., objects of different classes. There are various kinds of polymorphism: parametric, inclusion, overloading and coercion.

Subtyping induces inclusion polymorphism. For instance, consider a class shape which has a method draw. The circle class, which subtypes the shape class, then also has a draw method. This allows to use a circle or a square object with the shape operations. One can then design a system relying only on the

methods of the inherited class; run-time binding mechanism will then call into use the proper object.

A similar concept is that of overloading. A symbol is overloaded when it is used for operations that have different semantics depending on the class of the operands (e.g., the '+' operator in Java which adds integers and concatenates strings).

In the service oriented architecture is hard to find equivalent notions, because a formal concept of typing and inheritance is missing.

Design **patterns** [8] are often connected with OO methodologies, especially for describing the interactions between the objects in a system. A Design Pattern is a well understood and proved solution to a design problem, such as creating a wrapper around an object or defining the interface between a client and a server. A pattern differs from an algorithm because it includes both procedures and architecture, described in a way resembling more a case study than a precise prescription.

This approach is quite effective and can be relevant for designing and developing individual services, but its application is more related to software engineering methodology while Service Oriented Computing appears to be more an information system engineering issue.

Other typical object oriented abstraction items to be found in [3] are (1) division into parts, encapsulation, interface and implementation, which directly map to SOC abstraction principles; (2) the service view which is exactly where the SOC emphasis lies; and (3) layers of specialization, history of abstraction, frameworks, which are not relevant in this first comparison between OOP and SOC.

3 Is a service an object?

We have so far seen that connections between objects and services are not at all new. Some references draw explicit links between the two concepts, e.g., "As a very rough approximation, one web service can be compared to one method in more traditional software context" [11]. Other connections are less evident; for instance in [1] the authors describe a methodology for defining what they call a "Compatible Service", that is an abstract description of a class of services, derived from concrete services description. While this generalization mechanism seems the opposite of creating an instance from a prototypical description, the concepts involved are quite similar.

In general, it is not immediate to identify in the world of services an analogy for the concepts of class and object. In OOP, a class is a category that represents a set of objects having the same characteristics, and an object is a concrete realization of a class – an *instance* of a class. While classes are stateless an instance of a class has a state which depends on the sequence of operations undergone by it. The object behavior in response to an external request is determined by the class. All the object derived from a class will respond in the same way in

response to the same invocation, provided they are in the same state, or the response does not depend on the state.

Are we now in the position to answer the question of whether a service is an object? We propose a negative answer to this question, but the analysis provided so far brings evidence to the fact that many similarities connect OOP and SOC. More object related concepts can move into the service oriented world in order to enhance the technology and, perhaps, clarify the role and scope of web services. Here are the most immediate example of concept migration:

Inheritance. Of the two concepts of inheritance for OOP, the interface inheritance seems to be the most immediate to apply to web services. Consider a payment service which could be subtyped in a service with acknowledgment of receipt. In a workflow, the former could be substituted by the latter as it is guaranteed that the same port types are implemented in the subtyped service.

Inheritance enables service substitution, service composition and it induces a notion of inheritance on entire compositions of services. Consider a workflow A built on a generic service and another one B with the same data and control links, but built on services which subtype the services of A. Could we say that B inherits from A or that B is a specialization of A?

Polymorphism. Both inclusion polymorphism and overloading can be extended to the service paradigm. A composition operation in a workflow may have different meanings depending on the type of the composed services. For example, composing a payment and a delivery service may have a semantics for which the two services run in parallel; on the other hand, the composition of two subtyped services in which the payment must be acknowledged by the payers bank and the delivery must include the payment transaction identifier have the semantics of a sequencing the execution of the services.

Composition. A formal and accepted notion of composition is currently missing in the SOC domain and, as just proposed, inheritance and polymorphism could induce such precise notions of composition over services. Could this help dissolve the fog around the meaning of composition for web services? Could this bring together "syntacticians" which claim that nothing can be composed if not by design with "semanticians" which claim that anything can be composed automatically? Perhaps not, but it could fill some of the gaps left by standards which do not have a clear semantics, most notably, BPEL [2] which is proposing itself as the standard for expressing aggregations of web services.

Statefullness. Finally, the difference between stateless class definitions and statefull objects in OOP can impact web services technology where services are stateless entities. Web services resemble more to class definitions, but the notion of an existing instance of a service with its state is paramount. Software entities need to access each others state in order to fully interoperate. Here the parallel is with the history of HTML pages. When first introduced HTTP/HTML interactions were stateless, but this limited by far the client-server communication. It did not take long before the introduction of statefull interactions via the invention of 'cookies' [9].

The object oriented paradigm has a solid formal background and is a well-established reality of today's computer science. Service oriented computing is, on the other hand, a new emerging field, which tryies to realize global interoperability between independent services. To meet this goal, service oriented technology will need to solve a number of challenging issues, such as how to manage precise service semantics. One way to attack this problems is by 'borrowing' concepts from the object oriented world. In this paper we presented a parallel between objects and services that might be somewhat arguable, but one cannot dispute that services exhibit a number of object-like behaviors. Our focus has been on inheritance and polymorphism for composition semantics and we have also stressed the need of state information for services, but we do believe that there is space for even further contamination between object oriented methodologies and service oriented computing.

References

1. V. De Antonellis, M. Melchiori, B. Pernici, and P. Plebani. A methodology for e-Service substitutability in a virtual district environment. In J. Eder and M. Missikoff, editors, *CAiSE 2003*, pages 552–567, 2003.
2. BEA, IBM, Microsoft, SAP AG, and Siebel. Business Process Execution Language for Web Services, 2003. http://www-106.ibm.com/developerworks/library/ws-bpel/.
3. T. Budd. *An Introduction to Object-Oriented Programming*. Addison Wesley, 2002. (3rd edition).
4. F. Casati, S. Ilnicki, L. Jin, V. Krishnamoorthy, and M.-C. Shan. eFlow: a platform for developing and managing composite e-services. Technical report, Hewlett Packard, 2000.
5. G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design*. Addison Wesley, 2001. (3rd edition).
6. F. Curbera, W. Nagy, and S. Weerawarana. Web services: Why and how. In *Workshop on Obejcet Orientation and Web Services OOWS2001*, 2001.
7. M. Papazoglou et al. SOC: Service Oriented Computing manifesto, 2003. Working draft available at http://www.eusoc.net.
8. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable ObjectOriented Software*. Addison-Wesley, 1995.
9. D. Kristol. HTTP cookies: Standards, privacy, and politics. *ACM Transactions on Internet Technology (TOIT)*, 1(2):151–198, 2001.
10. S. McIlraith and T. Son. Adapting Golog for composition of semantic web services. In *Proc. of the Int. Conf. on Knowledge Representation and Reasoning (KR2002)*, pages 482–493, 2002.
11. G. Piccinelli, A. Finkelstein, and C. Nentwich. Web service need consistency. In *Workshop on Obejcet Orientation and Web Services OOWS2002*, 2002.
12. C. Szyperski. *Component software: beyond object-oriented programming*. Addison-Wesley, ACM, 1998.
13. J. Yang and M. Papazoglou. Web component: A substrate for web service reuse and composition. In *CAiSE*, pages 21–36, 2002.