Building Reusable Components with Service-Oriented Architectures

Haibin Zhu, *Senior Member, IEEE*
Department of Computer Science and Mathematics, Nipissing University
100 College Drive, North Bay, Ontario, P1B 8L7, Canada
Email: haibinz@nipissingu.ca

Abstract *– To build, manage and apply reusable components easily and efficiently are an ideal of software developers. Service-oriented architectures provide more hopes for reusable components. However, there are still many challenges for services to become a new paradigm to support reusable components. This paper analyzes the difficulties of software reuse and reusable components, discusses the different categories of service architectures, points out the major issues of reusable services and proposes initial solutions.*

Keywords: *Reusable, Component, Services, SOA*

1 Introduction

To obtain a higher software production rate is the major aim of software engineering. People have worked hard for more than three decades in this direction. To improve the production rate of software development, reusability has been considered as an ideal and key factor. Component-based development is one of the possible solutions to this requirement. The initial idea of component-based development is learning from the hardware industry to reuse hardware components such as integrated circuits (IC), large-scale integrated circuit (LSI), very large scale integrated circuits (VLSI), chips, chip-sets, boards, etc [12]. However, more than 20 years have passed and component-based development has not accomplished as much as expected.

Service-oriented architectures (SOA) may contribute to software engineering in that it could provide more reusable components. Therefore, it is important and productive to conduct research on how to develop software with service –oriented computing technology. There is not enough research and practice on how to make the " register, find, bind and execute" paradigm really practical and cost-effective. We need to analyze this process deeply to provide practical architectures and methodologies for reusable services.

This paper is arranged as follows. Section 2 points out the key problems of software reuse; section 3 proposes a methodology to categorize different service architectures; section 4 discusses the fundamental challenges in building practical reusable services, such as service registry, service negotiation, service binding and service execution; and last, section 5 concludes that services are new types of reusable components and we still need more research to make them practical to reuse.

2 The Key Problems of Reusable Components

There have been many discussions on the difficulties of software reuse. The reasons why reusable components are not satisfactory hitherto have been discussed in [2, 6, 9]. In fact, reusing hardware components is well accepted, because in hardware engineering, to understand a component is much easier than to make it. It is impossible for application developers to develop hardware components such as chips, but it is possible for application developers to develop software components such as a program segment, a function, or even a class. In other words, it is very difficult to build a development platform for hardware components but it is relatively easy and cheap to build a platform to develop software.

In the software industry, a software component either is too simple or has too complicated specifications. If it is too simple, the programmers believe that they can make it by themselves. If the specifications are too complex, the situation is that, after understanding the component's specification, the programmers believe that they can make a better one by themselves. " If it takes the typical programmers more than 2 minutes and 27 seconds to find something, they will conclude it does not exist and therefore will reinvent it [3]".

Therefore, traditional software components cannot be designed and applied in the same way hardware components are. The key difficulty is to make specifications much simpler than the implementation logic.

To make services more reusable, we need two kinds of efforts. One is that we must provide the services that are difficult to develop and we should have an advanced development environment that is difficult or expensive for application developers to build or purchase. The other one is that we should make the services easy to apply and we should provide cheap service application tools that are easy and cheap to install. With these two aspects successfully implemented, we could really separate the concerns of application developers and the service developers. That is why the " register, find, bind and execute" paradigm is a possible way to make more reusable components. With a service registry that is a directory on a distributed or networked environment, we can manage service contracts dynamically. After finding a service, asking for the service is nothing but just providing the data that the service will process. This style completely blocks the intention of intervening with the details of the service implementation.

3 Fundamental Architectures for Services

Architectures are the first considerations for any newly developed technologies. To make services more reusable, we need to analyze the possible architectures first. To analyze architectures, we must be clear about the criteria in evaluating newly developed reuse technologies. Krueger proposed a framework in 1992 to evaluate software reusability with the following four aspects [9]: abstraction: a reusable component should have a specific level of abstraction that avoids the software developers to sift the details of it; selection: a reusable component should be easy to locate, compare, and select; specialization: a group of reusable components should be generalized and the generalized form should be easy to be specialized to an application; and integration: There must be an integration framework to help reusable components be installed and integrated into newly developed applications.

From the above four criteria, we found that abstraction and specialization for service computing are definite, i.e., all the components are in the form of services, and all the service providers are trying to provide as many special services as possible. SOA adds a layer of abstraction on top of existing computing environments, which is part of why it is so useful in heterogeneous situations. A service as a good abstraction should be affordable, i.e., purchasing a service should be cheaper than developing by the application developers themselves; attractive, i.e., a service should provide exciting performance; necessary, i.e., a service should be necessary for developing specific applications; professional, i.e., the providers should have strong experiences and background for the services; and trustworthy, i.e., it is safe to use the services provided.

To achieve these goals, we need to make it easy to select and integrate a service. A good architecture may contribute to these two aspects, i.e., to improve selection and integration. From the viewpoint of object-oriented methodology [13] and architectures of distributed systems [4], we can find the following four architectures.

3.1 Centralized service center

For selection, this architecture makes it easy to locate a service, because the number of service centers is limited and definite. For integration, the framework to apply a service into an application is restricted by the service center. There is not much flexibility for the application providers, i.e., the application developers should use the formats provided by service centers to request services.

The service is computing extensive or data storage extensive (Fig.1). The data to be processed is small. The service specification is clear and it is easy to sign a contract. This architecture is not considered appropriate sometimes, because large quantities of cheap workstations and desktop computers are available. But in reality, it is still one practical solution for many complicated services such as Global Positioning System (GPS) services, Coordinated Universal Time (UTC) services, weather forecast services and bioinformatics services. This kind of architecture can be found in space and medical research centers such as National Aeronautics and Space Administration (NASA) and the Europe Space Agency.

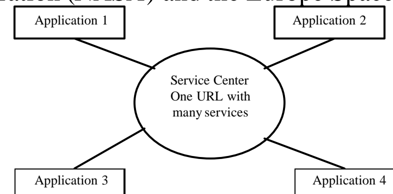

Fig. 1 Centralized Service Center

This center might need to install powerful supercomputers that are normally not afforded to application developers. The computing power should be greatly superior to any application developer's computer. It also needs to provide large databases that normally application developers are unable to be managed. The volume of database should be greatly superior to any database managed by an application developer. It may also install special expensive equipment to provide special services such as atomic clocks, highly exact GPS receivers, and bioinformatics databases.

3.2 Distributed Service Center

For both selection and integration, it is the same as centralized service centers. Its difference from the

centralized center is the construction of the center but not the service provisions (Fig. 2). It provides large quantities of services concurrently. This kind of center should be equipped with high speed networks. The bandwidth should support the rush service hours without significant delay for satisfactory services. This architecture is appropriate to large number of service requests with small number of data to be processed. This architecture can be found in VSM [8].

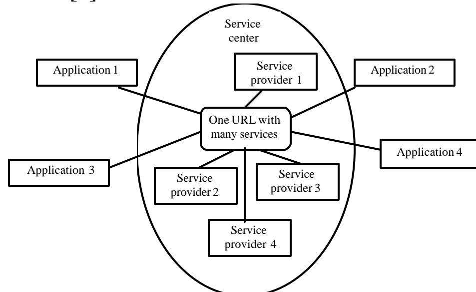

Fig. 2 Distributed service center

For application developers, it is still a center. There is no difference for application developers to ask for services from between a distributed service center and a centralized service center. But there are significant differences between building a distributed service center and building a centralized service center.

3.3 Distributed service providers with centralized service registry

The selection is easy because the application developers only need to contact the service registry. The integration of this architecture is different from the former two in that it applies the services provided by different service providers. The service registries store the service specifications but not the real services. Therefore, the integration with this architecture is more flexible (Fig. 3).

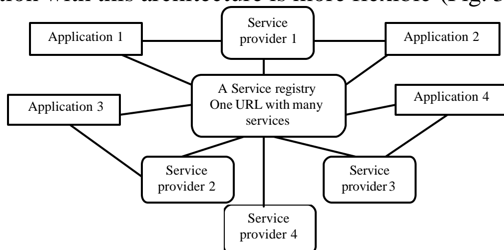

Fig. 3 Distributed providers with a centralized registry

The goal of this architecture is to build a service registry and many small service providers. This direction is the fundamental idea for grid computing, that is, using large quantities of the idle computing power of workstations and desktop computers. In this style, each service provider may only provide a small number of services. The difference between this architecture and the distributed service center is as follows:

- In the former architecture, there is one special computer that is responsible for accepting and replying to service requests.
- In the latter, the service providers will accept and reply to service requests by themselves.

This architecture has some variations:

- Completely duplicate service registries: each registry possesses all the services;
- Complete different service registries: each only possesses partial services without any overlapping; and
- Interleaved service registries: each only possesses a part of the services and there are overlapped services among some of the service registries.

The service specifications are standard with a predefined format and service negotiation is relatively easy. The service providers have the illustration of the services. The Jini service architecture also belongs to this category [10].

3.4 Distributed Service Providers and Service Registries

This architecture is the most complex one. Its selection and integration are much more flexible. However, it requires special consideration of service registry, service negotiation and service provisions. Much of the literature in service computing is trying to contribute within these types of architectures [7] (Fig. 4).

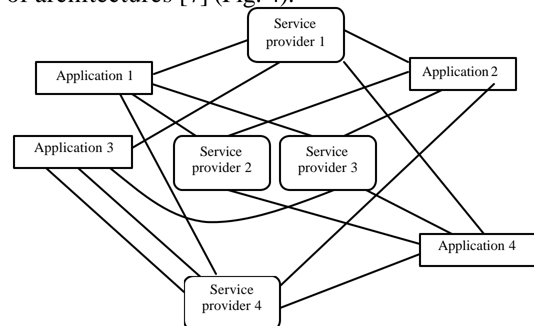

Fig. 4 Distributed providers and registries

With this architecture, service providers are located at different computers with different URLs. We call them independent service providers. The service registries are also distributed on different computers. All of the service providers form a service network. The application developers will broadcast or multicast their requests to all the networked computers, collect the replies, compare and select one service provider, sign a contract and send out the service request. Handorean and Roman [7] proposed a similar architecture. With this architecture, both the service providers and the application developers may specify the services. It needs a lot of negotiation for the two parties of the service to make an agreement.

4 Issues of Reusable Services

The fundamental reason to take services as reusable components is that services may provide shorter cognitive distances. Cognitive distance is defined by the amount of intellectual effort that must be expended by software developers in order to take a software system from one stage of development to another [9]. In SOA, a service must be described, specified, implemented, registered, published and finally applied.

4.1 Service Registry

To register a service, we need a structure to manage large quantities of service specifications. We developed a service specification model for service registry. A service is defined as a tuple $\mathcal{S} ::= <\mathcal{N}, \mathcal{P}, I, \mathcal{R}, \mathcal{M}>$, where,

- $\mathcal{N}$ is the name of the service provider (internally an identification or Universal Resource Locator (URL) of the service provider);
- $\mathcal{P}$ is the pattern of the service (the simplest form of the pattern is the name of the service and the most complex form is the complete function description of the service);
- I is the format of inputs of the service;
- $\mathcal{R}$ is the format of returned results of the service; and
- $\mathcal{M}$ is the implementation of the service.

We can obtain basic guidelines to evaluate if an entity can be developed into a good service. For a good service, it is either (1) or (2) as follows.

(1) $\mathcal{T}_{dev}(\mathcal{M}) >> \mathcal{T}_{und}(\mathcal{P})$ means that the time to implement a service should be greatly larger than the time to understand the service pattern, where " >>" means " greatly larger than" . In this case, the service logic is complex and needs professional experiences or strong backgrounds in the problem fields. The major concern to develop this kind of service is time but not space.

(2) $Q(I+\mathcal{R}) << Q(\mathcal{M})$ means that the space occupied by the input data and the result data should be greatly less than the space occupied by the service including data and processes, where " <<" means " greatly less than" . This case is mainly concerned with the space occupied by the service implementation.

A service registry stores and manages a collection of service specifications or a set of bindings between a service name and a service specification. To provide a good service registry, we need to have a definite format to specify services. Some authors use Jini, Java interfaces and XML as languages to specify services [10]. Dijkman et. al. [5] proposes to distinguish four levels of description technology: interface description, service description, internal behavior description and choreography description. In reality, the interface description and service descriptions are syntactical parts of a service and the internal behavior description and choreography descriptions are about the semantics of the service.

We can extract the service specification $\mathcal{S}_{spec} ::= <\mathcal{N}, \mathcal{P}, I, \mathcal{R}, \Upsilon>$ from the definition of services $\mathcal{S}::=<\mathcal{N}, \mathcal{P}, I, \mathcal{R}, \mathcal{M}>$, where Υ is the semantics description of the service. To manage service specifications with Java, a service registry can be expressed by a class Registry that implements the Map interface where the key of an entry for this map is a service patter $\mathcal{P}$ and the value is a service provider, a service input format, a returning result format, and a semantics description of the service, i.e., $<\mathcal{N}, I, \mathcal{R}, \Upsilon>$. Considering the situations of overloading, a value may consists a list of $<\mathcal{N}, I, \mathcal{R}, \Upsilon>$.

With the third architecture in section 3, a service registry should support different kinds of search methods for service requesters to search a service. If a service requester issues a query with a pattern $\mathcal{P}$, a service registry will send a service specification $<\mathcal{N}, \mathcal{P}, I, \mathcal{R}, \Upsilon>$ or a list of service specifications as a reply. If a service requester issues a request with an $\mathcal{N}$ such as a URL, the reply from a service registry should be a list of service specifications.

4.2 Service Discovery and Negotiation

Service requesters can be either arbitrary application developers or other service providers. A service provider needs to register its services with a service registry and provide services directly to interested parties. Each service may have multiple service interfaces to meet the needs of different requesters, and requesters can dynamically discover the interfaces they require. Making discovery-based service abstraction is challenging.

A service provider (or a service center, a service registry) should provide the search and retrieval of services. A service registry is attached to many service providers and will not actually execute the services by itself. It only shows all the service specifications, illustrates the meanings of services and negotiates specifications with application developers and providers.

To negotiate the semantics of a service, we need a tool, a regulation or a platform. A contract is one such tool. A contract is an agreement for providing specific services or performance of specified work at a certain cost, rate, or commission [1]. A contract actually demonstrates an agreement on the syntax and the semantics of the service between the two parties relevant to the service.

A contract consists of the time to complete the service, the cost for the service and the data provided to

the service. It should describe not only the functionality, the time cost, and the money cost but also the service and contract conditions. To help service negotiation, a contract should have the properties as follows:

- Concise and understandable: it should be easy to understand. This is the same as the requirement for service interfaces. It is also consistent with the criterion (1) of our service evaluation.
- Clear and rigid: every item should be clear and without ambiguity.
- Concrete and easily evaluated: for every service it should be easy to evaluate the quality of a service or to check if the service is completed.

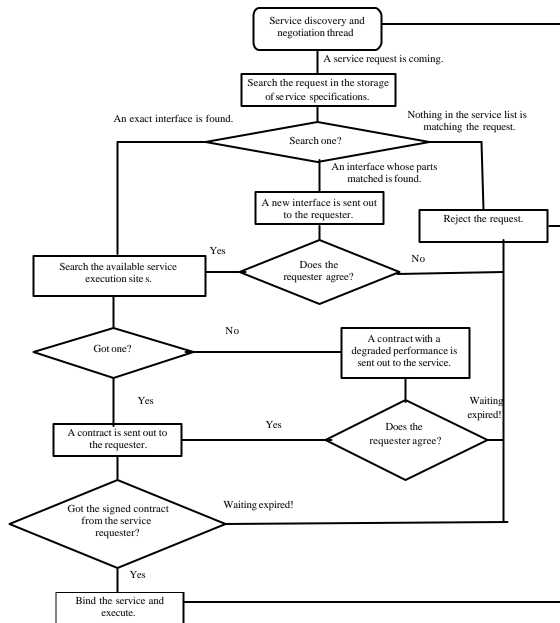

Fig. 5 The thread of service discovery and negotiation

With the above requirement, we compose a contract model. A contract for a service is defined as $\chi ::= < S_{spec}, t, c_s, c_p >$, where, S_{spec} is a service specification; t is the performance requirement to complete the service; c_s is the cost of the service; and c_p is the penalty to the service provider's failing to meet the performance requirement. To make negotiation simple, one contract has only one service specification. A procedure of service negotiation is described as shown in Fig. 5. It can be used for all the architectures we discussed at a service center or registry.

4.3 Service Binding

After a service is discovered, binding should be done before execution. The first three architectures we discussed in section 3 provide services with definite identifications for service centers or service providers. Logically, it is a traditional style in composing distributed applications, such as Remote Procedure Call (RPC), remote objects, Common Object Request Broker Architecture (CORBA) and Remote Method Invocation (RMI). These are called identity-based services [11]. By identity-based we mean that a service request is sent to a service center or a service provider and the application knows exactly where the service is.

If we use the fourth architecture discussed in section 3, the application developers do not need to know the concrete service providers. We can locate a service provider randomly. In this style, when an application needs a service, it may broadcast its request. It receives the bids, compares and selects one from them to sign a contract and to finally request to the service provider.

When an application is working, there might still be new service providers joining the service network. The service requester should be able to apply the new service if it costs less or provides better performance. Here, we use "static" to mean services are determined before the application is running and "dynamic" means that services are determined when the application is running. By "dynamic", we have another meaning that the services might be removed and replaced with a new service. To complete this, we need a facility to constantly contact the service providers and regularly compare and select the most economic services. As for the architectures, the first three architectures are easy to accommodate static binding. For the architecture of distributed service providers and service registries, every application might be able to find an available service dynamically. Fig. 6 shows the flowchart of a thread to listen to new services.

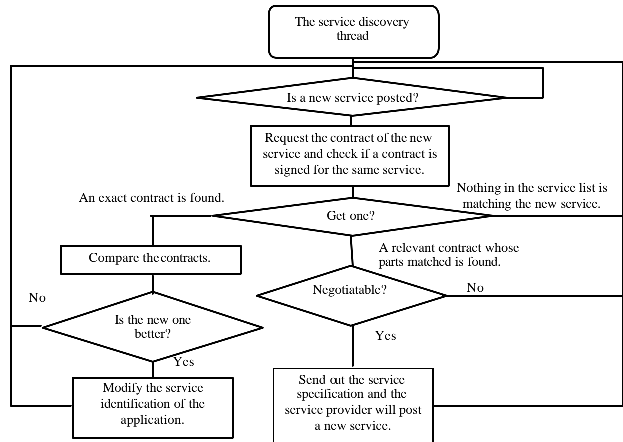

Fig. 6 The thread that listens to new services

4.4 Service Execution

In traditional procedural programming, there are normally two different procedure-calling mechanisms, i.e., call-by value and call-by-reference. When the data to be processed by a procedure is small, we generally use call-by-value, i.e., copy the values of the actual parameters to the formal parameters. When the data is too large, we

generally use the mechanism of call-by-reference to avoid a large quantity of copying. In the remote procedure call (RPC) mechanism, it is ordinarily accepted that call-by-reference is not practical because there is a need for the remote procedure to access remotely located data.

In a grid of services, how should we execute the service request and reply? If we consider the address space of all the grid interconnected computers as one logical data space, we could also use the same ideas in the procedural programming such as call-by-value or call-by-reference. That is to say, when the data is small, we can directly send out a service request accompanied with the data copy to the service provider. If the input data is too large, we can only send out a service request accompanied with the data reference including the address of the data (URL and the internal data address), the access rights to the data to the service provider.

In our daily lives, we normally ask for a service and the service providers generally do not explicitly require us to send out data. In current web communities, a person's information is distributed in different databases widely spread on the internet. For example, a credit check service does not require the requesters to provide the data of the person to be checked. Therefore, in SOA, there would not be a requirement for an application developer to send out all the data to be processed. There are two possible situations for service execution: one is that the service must reside on the computers of the service provider (case (2)) and the other is that the service is able to be transferred to other computers (case (1)).

5 Conclusion

A service as a reusable unit is more appropriate for reusing software, because the logic of a service is generally so complicated that it is much easier for the service requesters to understand the contract than to implement the service by themselves. In this way, service requesters do not care and cannot care about the details of service implementations. This is a large step towards reusable components.

SOA brought new chances to improve the development of reusable components. However, there are still many challenges that need to be overcome. We have proposed some initial solutions to make services more reusable. There are still many potential research topics to make services reusable, as follows:

- Standardize service specification languages;
- Provide more practical mechanisms for service publications;
- Implement high performance service registry and service discovery;
- Provide rigid contract negotiation tools;
- Implement dynamic service binding; and
- Develop strategies where the service execution should be located.

6 ACKNOWLEDGMENTS

This research is partly supported by the IBM Eclipse Innovation Grant Funding. Thanks to Georgia Irwin and Pierre Seguin for their proofreading of this article.

References

[1] Brown, A., Johnston, S., and Kelly, K., "Using Service-Oriented Architecture and Component-Based Development to Build Web Service Applications", *A Rational Software White Paper*, Rational Software Corporation, 2002.

[2] Budd, T., 2002. *An Introduction to Object-Oriented Programming (3rd Ed.)*, Addison-Wesley, 2002.

[3] Constantine, L. L., *The Peopleware Papers: Notes on Human side of Software.* Prentice Hall Inc., 2001.

[4] Coulouris, G., Dollimore, J., Kindberg, T., *Distributed Systems: Concepts and Design (*3rd Ed.*)*, Addison Wesley, 2001.

[5] Dijkman, R.M., Quartel, D.A.C., Ferreira Pires, L., van Sinderen, M.J., "The State of the Art in Service-Oriented Computing and Design", ArCo/WP1/T1/V1.00, University of Twente, Enschede, The Netherlands, July 2003.

[6] Frakes, W.B., and Fox, C.J., "Sixteen questions about software reuse", *Communications of the ACM*, vol. 38 , no. 6 , June 1995, pp. 75-87.

[7] Handorean, R. and Roman, G. C., "Service Provision in Ad Hoc Networks", *Proc. of the 5th International Conference on Coordination Models and Languages, York, UK, April 811, 2002,* pp. 207-219.

[8] Jeng, J.J., "An Approach to Designing Reusable Service Framework via Virtual Service Machine", *SSR'01*, May 2001, Toronto, Ontario, Canada, pp. 58-66.

[9] Krueger, C. W., "Software Reuse", *ACM Computing Survey*, vol. 24, no. 2, June 1992, pp. 131-183.

[10] Richard, G.G., "Service Advertisement and Discovery: Enabling Universal Device Cooperation", *IEEE Internet Computing*, vol. 4, no. 5, Sep/Oct 2000, pp. 18-26.

[11] Sarukkai, S., "Identity-Based Service-Oriented Architecture", *Web Service Journal*, Feb. 2005, http://www.sys-con.com/story/?storyid=48038&DE=1 .

[12] Zhu, H. and Chen, H., "Composing Software Components Based on the Mechanisms of Smalltalk", *Computer Science*, vol. 8 (3), 1990 (in Chinese).

[13] Zhu, H. and Zhou, M. C., 2003. "Methodology First and Language Second: A Way to Teach Object-Oriented Programming", *Companion of the International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'03*), USA, Oct., 2003, pp. 140-147.