Adapter Patterns for Resolving Mismatches in Service Discovery*

Hyun Jung La and Soo Dong Kim

Department of Computer Science
Soongsil University
1-1 Sangdo-Dong, Dongjak-Ku, Seoul, Korea 156-743
hjla@otlab.ssu.ac.kr, sdkim777@gmail.com

Abstract. The theme of service-oriented computing is largely centered on reusing already existing services. Service providers model common features among potential applications, realize them as reusable services, and publish in service registries. Service consumers discover appropriate services and subscribe them. In developing application with reusable services, there exists a key technical problem, called mismatch problem which is a gap between the required feature and the feature of a candidate service. The adaptability of available services is a key factor in determining the reusability of the published services by resolving mismatches. Hence, we claim that the design of adapters should be an essential activity for developing service-oriented applications. In this paper, we identify recurring mismatch types in discovering services. And, we present four adapter patterns handling the mismatch problems. By using the adapter patterns, service providers could develop highly reusable services, and service consumers will be able to reuse more services available.

Keywords: service mismatch problem, adapter pattern.

1 Introduction

As an effective reuse paradigm, the theme of service-oriented computing is largely centered on reusing already existing services. Service providers model common features among potential applications in a domain, realize them as reusable services, and publish in service registries. Service consumers discover appropriate services and subscribe them in their systems. Hence, being able to reuse published services is the most fundamental underlying notion of service-oriented computing.

However, there exists a key technical problem in reusing the services; *mismatch problem*. Let $FEA_1, FEA_2 \ldots, FEA_n$, be the features needed in developing a target system. And, let $SVC_1, SVC_2 \ldots, SVC_m$, be the services available in services registries. Service consumers would try to locate the right service SVC_j for a feature FEA_i for the target system, and then the discovery will result in one of the following categories;

* This research was supported by the National IT Industry Promotion Agency (NIPA) under the program of Software Engineering Technologies Development.

A. Dan, F. Gittler, and F. Toumani (Eds.): ICSOC/ServiceWave 2009, LNCS 6275, pp. 498–508, 2010.

- Category 1) Finding SVC_j which fully fulfills FEA_i. This is the happy case that the located service can be included and reused in the target system, yielding a high reusability.
- Category 2) Finding no SVC_j which fully fulfills FEA_i. Since there is no fully matching service, the required feature FEA_i should be newly implemented.
- Category 3) Finding SVC_j which partially fulfills FEA_i. Due to the partial matching, the located service SVC_j may or may not be reusable in the target system depending on its adaptability. This is the motivation for our research.

Hence, *mismatch* is defined as the gap between the required feature FEA_i and the feature of a candidate service SVC_j. *Adaptation* is an activity of adapting such service SVC_j to make it fulfill the required feature FEA_i. Hence, the adaptability of available services is a key factor in determining the applicability and reusability of the published services. Without adaptation capability, services of partial fulfillment (in the category 3) could not be utilized in developing target systems. Hence, we claim that the design of adapters should be an essential activity for developing service-oriented applications.

Like the motivation for object-oriented design patterns [1], we observe recurring problems of mismatch in discovering services, i.e. typical patterns of service mismatches. Accordingly, we believe that developing patterns for designing adapters which can effectively resolve the recurring mismatch problems would be feasible.

In this paper, we first identify recurring mismatch types in discovering services, and define the roles of service providers and consumers to support adaptability. Then, we specify each adapter pattern. For the paper organization, we give a survey of related works in section 2. And, we present taxonomy of service mismatches, and roles of service providers and consumers in section 3. Each pattern is specified in details in section 4, and assessment work is presented in section 5. By using the adapter patterns, service providers could develop highly reusable services, and service consumers will be able to reuse more services available.

2 Related Works

Kongdenfha and his propose the aspect-oriented approaches to adapt mismatches in *interfaces* and *protocols* [2]. First, they define interface and protocol mismatch patterns including adaptation logic that resolves the mismatch. Then, they present applying aspect-oriented approach to modify the services. At runtime, adaptation logic is generated and woven into the adapted service. To do this, they define three mismatch patterns including *Signature Mismatch Pattern*, *Missing Message Pattern*, and *One-to-Many Pattern*. Although their approaches are presented in much detailed level to show the applicability and practicability, this work only focuses on the interface and workflow mismatches.

Benetallah's work proposes methods for developing adapters which can resolve mismatches in service interfaces and business processes [3]. A mismatch pattern specifies situation, information for adapter instantiation, and pseudo-code. Each pattern is provided for resolving mismatches of service interface and business process

level. This work proposes the mismatches confined to service component and business process and the steps to resolve mismatches without giving details for developing the adapters.

Sam's work proposes methods for identifying service mismatches and dynamically adapting the services [4]. The method for identifying mismatches utilizes *service configuration* which is a specification of syntactic aspect, semantic aspect and constraints of service interfaces. The method for adapting services utilizes *context association rules* to transform interfaces of published services to the service interfaces expected by consumers. This work mainly deals with interface mismatches and interface transformation.

Erl presents various kinds of patterns for realizing service-oriented architecture [5]. Most of the patterns are related to designing services, but they also address how to resolve differences in data formats, data models, and communication protocols by using three patterns; *Data Format Transformation*, *Data Model Transformation*, and *Protocol Bridging*. The mismatches are only for interface-level information, and the proposed patterns need to be enhanced in a practical manner.

3 Fundamentals of Service Mismatches

3.1 Types of Service Mismatches

Services are published in registries with their *Service Specification*, which becomes vital information for service discovery. Service consumers look up the specification and may find mismatches. Hence, we can derive typical mismatches from the key elements of the service specification. A service is commonly specified with its interface, functionality, and provided Quality of Service (QoS)[6].

Service Interface element of a service has *Provide Interface* and *Required Interface*. *Provide Interface* specifies the functionality delivered by the service, and *Required Interface* specifies external services or objects which should be plugged into the current service to make the service fully functional [7].

Service Functionality consists of *External Behavior* and *Internal Behavior* of operations in the service. The external behavior is typically specified with a functional overview, pre-condition, post-condition, invariant, and constraints. The internal behavior of a service operation specifies the logic, algorithm, rules, and any other internal details of the operation, and hence it is typically invisible to outside.

QoS consists of one or more quality attributes and their expected values at runtime, which is typically specified in *Service Level Agreement*. Hence, services are specified with certain levels of quality attributes.

Types of mismatches can be derived for each element in *Service Specification* by considering its plausible abnormality, i.e. *Interface Mismatch, Functionality Mismatch,* and *QoS Mismatch*. Furthermore, each mismatch type can be specified into more specialized mismatch types. For example, *Interface Mismatch* is specialized into *Signature Mismatch* and *Semantic Mismatch*. Fig. 1 represents all the mismatch types which are derived from the each element of the service specification.

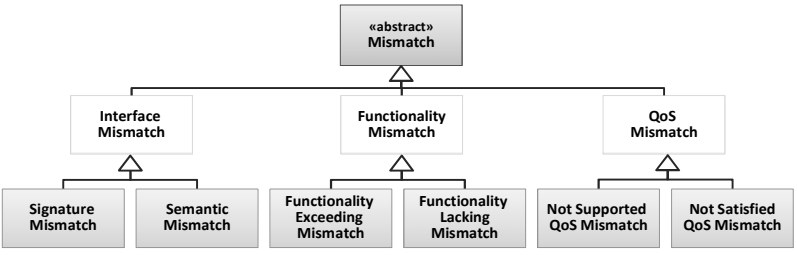

Fig. 1. Hierarchy of Partial Mismatches

The rest of the mismatch types are further specified within adaptation patterns in section 4.

3.2 Roles of Service Providers and Consumers for Adaptability

When developing applications with services, service consumers can face the mismatch problems so that they take responsibility for *adapting* the services. Therefore, service consumers develop static adapters by realizing the most appropriate *adapter pattern*. To do so, they identify the type of mismatches, design how to resolve the mismatches, and utilize the service after adapting it.

But, it is not always possible to resolve the mismatches since the service consumers should access enough information to develop the suitable adapters. That is, if service providers do not provide methods for static adaptation, service consumers have a limitation on adapting the services. Therefore, service providers *enable* the consumer to resolve the mismatches by providing sufficient service information. To enable the service adaptation, it is feasible for the service provider to define *required interface* which can assert consumer-specific variants [8].

4 Specification of Adapter Patterns

The mismatches identified in section 3.1 occur repeatedly when the services are invoked by various service consumers. To deal with these repeated mismatches, we propose several adapter patterns in terms of overview, applicable situation, structure, collaboration, consequence, and instruction.

Note that we do not cover the adapter patterns for resolving *Not Supported QoS Mismatch* and *Not Satisfied QoS Mismatch* since QoS cannot be managed at the structural and dynamic designs but the architecture design. Hence, we suggest utilizing architecture design approaches such as [9].

4.1 Signature Matching Pattern

Overview: This pattern is to adapt the interface of a candidate service so that the interface expected by a consumer and the interface of a provided service match.

Applicable Situation: When there is a candidate service but its interface do not fully match to the interface expected, this pattern can be used to resolve the mismatch.

In SOA, service interfaces are pre-defined by providers, and they cannot be re-defined by service consumers. There can be a situation where the interface of a candidate service does not fully match to the interface expected by a consumer application. We call this situation as *Signature Mismatch*.

Structure: The *Adapter* pattern such as [1] can be used. As shown in Fig. 2, *Caller* which is a class in an application defines a required interface, *op1( )*. *Callee* provides a candidate service having an interface, *op2( )*. Then, an *Adapter (*i.e. *adaptedClass)* resolves the mismatch by transforming signatures as shown in the note in the figure.

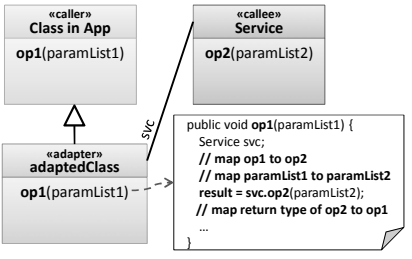

Fig. 2. Object Model for Signature Matching Pattern

Collaboration: At runtime, the participants in this pattern work together to resolve the signature mismatch as shown in Fig. 3. If there is more than one operation to transform the expected operation to provided one, there may be several self invocations in the instance of *adaptedClass*.

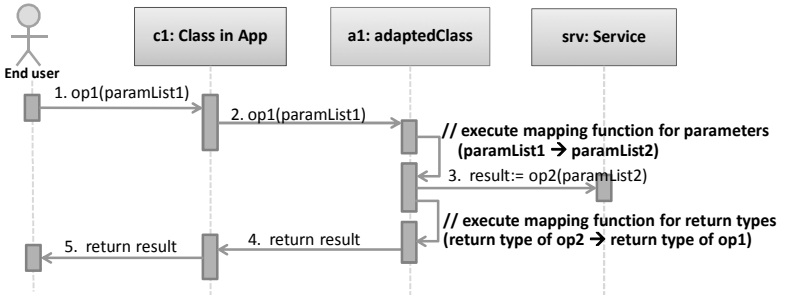

Fig. 3. Dynamic Model of Signature Matching Pattern

When the user invokes the method *op1(paramList1)* in the *Class in App*, its overridden method in *adaptedClass* is invoked by the principle of dynamic binding. As shown in the algorithm for the method in the figure, it adapts the interface mismatch by using mapping function for input parameter. Then, the method *op1(paramList1)* invokes the method in the *Service*, *op2(paramList2)*.

Consequence: Candidate services which have mismatches with services expected not becomes reusable, increasing the reusability of candidate services. There can be a minor performance penalty for having to invoke services through the *adapters*.

Instruction: To apply this pattern, first, we identify the mismatch between expected and provided interfaces which is *mismatch on the names of operations, mismatch on the datatypes of parameters and return value*, *mismatch on the orders of parameters*, and *mismatch on value ranges of parameters*. Second, we define a mapping rule for each mismatch. Mapping rules may vary from simple type casting to complex calculation. The next step is to add a code segment invoking the adapter in the application.

4.2 Semantic Matching Pattern

Overview: This pattern is to adapt a candidate service to make its semantic fit to the semantic of an expected service.

Applicable Situation: When developing the service-oriented applications, behavioral semantics of the service interface may not be satisfied with the consumer's expectation although the signature is compatible. Being different in semantics implies that there are minor gaps in *pre/post conditions* and *invariant*.

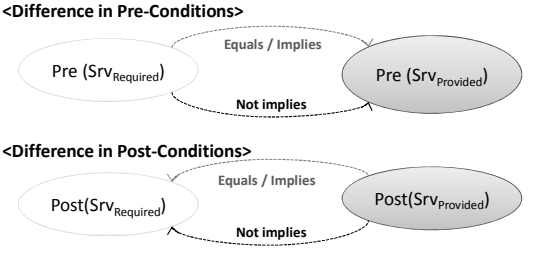

Fig. 4. Detailed View of Semantic Mismatches

Fig. 4 explains the applicable situations of this pattern in detail. Let $Pre(Srv_{Required})$ be the pre-condition of service $Srv_{Required}$ and $Post(Srv_{Required})$ be the post-condition of service $Srv_{Required.}$

When $Pre(Srv_{Required})$ is equal to or implies $Pre(Srv_{Provided})$, the target service can be applied. Otherwise, the service is not usable. Consider an example where $Pre(Srv_{Required})$ says A should be larger than 0, and $Pre(Srv_{Provided})$ says A is equal to or larger than 0. A problematic case is when A is 0, which requires an adaption.

Differences in the post-condition are opposite to ones in the pre-condition. Consider an example where $Post(Srv_{Required})$ says x should be equal to or larger than y, and $Post(Srv_{Provided})$ says x is larger than y. In this case, a problem occurs when x is equal to y, which requires an adaption.

As the third element describing the semantic, an invariant is a condition which must always hold while a service is being executed. Since services are provided as a black box, it would generally be not feasible to adapt invariants. This situation is called *Semantic Mismatch* as depicted in Fig. 1.

Structure: Adapting semantics of blackbox form of services is not trivial due to the limited accessibility and visibility. To mitigate gaps in the pre-conditions, adaptation will apply to both consumer and provider sides. For consumer side, the underlying mechanism of *Signature Matching Pattern* can be well applied. For provider side, we apply a plug-in mechanism [8]. This method can only be applicable when *required interface* is pre-defined. Through the required interface, service consumer can plug an external object which carries the consumer-specific logic, i.e. *variant*.

Unlike the pre conditions, the differences in the post conditions are only resolved on the provider side by using plug-in mechanism.

Fig. 5 depicts how plug-in mechanism is applied to this pattern. Service has a static attribute, *vp*, of *PlugType* type which is set through *plugIn* operation Class in App sends a message with the its own plug-in object, *myObj*.

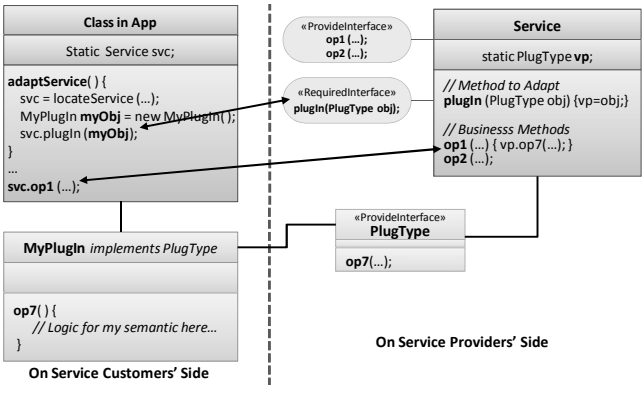

Fig. 5. Design of Semantic Matching Adapter with Plug-in Object

The Plug in object, *MyPlugIn*, embodies the adaptation functionality. To resolve this mismatch, the plug-in may perform additional functionality that reads the results provided by the service in reverse order. Note that in this plug-in mechanism, most of the functionality is fulfilled by the services provided and the small amount of the functionality is satisfied with the plug-in object.

Collaboration: When *Class in App* invokes the *Service*, the *Class in App* first creates its own plug-in object, an instance of *MyPlugIn* and then passes the object (i.e. *myObj*) as an argument by using the required interface, *plugIn(PlugType obj)*.

Then, on the service side, the argument *myObj* is copied to a static attribute, *vp,* for a variation point. Since then, whenever *op1()*gets invocation, the *Service* internally invokes op7() defined in *MyPlugIn* implementing consumer-specific functionality.

Consequences: Utilizing this pattern increases the reusability of candidate services, but there can be a minor performance penalty for invoking services after creating the plug-in objects. And, there may be some limitations to develop plug-in object since this pattern is only applicable when a service providers offers additional mechanism such as *Required Interface* which enables the object to be plugged in.

Instruction: To apply this pattern, we suggest the following process. First, we identify the mismatch the mismatch between expected and provided semantic. Second, we define a mapping relationship by applying semantic adaptation methods available such as [10] and [11]. When several semantic mismatches are adapted, some complications among the adapted items may occur.

The next step is to add a code segment invoking the adapter in the application. The place of the code segment for the adaption is different whether the semantic mismatches are about precondition or postcondition. In the former case, the code segments put before the actual service operation invocation. And in the latter case, the code is placed behind the service invocation.

4.3 Functionality Enhancing Pattern

Overview: This pattern is to supplement the functionality which is required by the service consumer but not offered as a service.

Applicable Situation: When service consumers look up the services based on their functional requirements, it is inevitable to find the service whose functionality does not fully match consumer's expectation. In this case, most of the functionalities are provided by the services but a little amount of functionality is not provided. This situation is called *Functionality Lacking Mismatch* as shown in Fig. 1.

Structure: To resolve *Functionality Lacking Mismatch*, we suggest using *Decorator pattern*[1] to append the needed functionality as shown in Fig. 6.

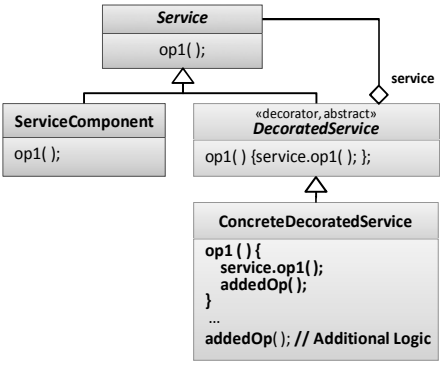

Fig. 6. Object Model of Functionality Enhancing Pattern

In the figure, the *Service* defines the interface for the *ServiceComponent* that provides its functionality but may have lack functionalities of the required ones. Hence, we define a service with an additional functionality, called *DecoratedService*. *DecoratedService* aggregates a reference to its superclass in recursive manner. At runtime, an instance of *ConcreteDecoratedService* substitutes its superclass object, and the *op1( )* in the class embeds an additional functionality in the method *addedOp( )*.

Collaboration: When the consumer invokes *op1()* of *ServiceComponent*, *Decorated-Servies* gets the invocation. In *op1()* defined in *DecoratedService*, it first invokes the *ServiceComponent* and then execute additional functionalities.

Consequence: The functionality-inappropriate service becomes useable so that the reusability of the service will be increased.

Instructions: To apply this pattern, we suggest the following process. First, we identify the lacking functionality by comparing consumer's *functional requirements* with *service functionality*. Second, we design the functionality. The next step is to add a code segment invoking the adapter in the application. The code segments for the adaptation are put after the actual service operation invocation.

4.4 Functionality Disabling Pattern

Overview: This pattern is to disable the unnecessary functionality of the service so that the expected functionality and the one of a provided service match.

Applicable Situation: This pattern is applied when the functionality of the service provides additional one required by the service consumer. Only if the accidental invocations of functionality create potential problems such as integrity violations and undesirable side-effects, this is problematic. This situation is called *Functionality Exceeding Mismatch* as shown in Fig. 1.

Structure: The *Proxy* pattern such as [1] can be used. As shown in Fig. 7, there are two subclasses of *ServiceInterface* implementing *op2()*. Functionalities of *op2* are almost similar, but *ServiceComponent* provides additional functionality. Hence, *ProxiedComponent* redefines the *op2()*, which *op2()* of *ServiceComponent* is first invoked and disables additional functionality by throwing exceptions.

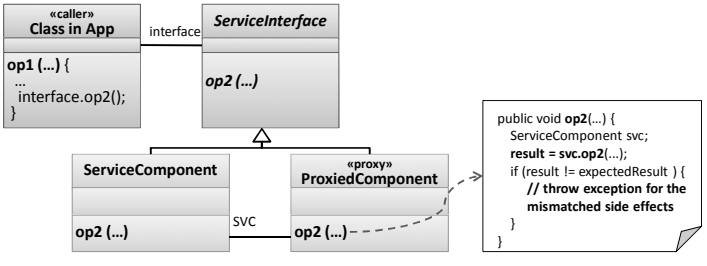

Fig. 7. Object Model of *Functionality Disabling Pattern*

Collaboration: When the user invokes the method *op2()* in the *op1()* of *Class in App*, its overridden method in *ProxiedComponent* is invoked. As shown in the algorithm in the figure, it internally invokes *op2()* in *ServiceComponent* and disables the additional functionality. Hence, additional functionality is no longer effective.

Consequence: The functionality-inappropriate service becomes useable so that the reusability of the service will be increased.

Instructions: To apply this pattern, we suggest the following process. First, we identify the exceeding functionality which is not required by the service consumer by comparing the functionality requirement and service functionality. Second, we check whether the exceeding functionality can be disabled or not. If not, such as side effects on the database, do not consider applying this pattern. Third, we design the way to disable the exceeding functionality such as treating as an exception. Then, we develop a proxy-like adapter which provides the same interface in the *ProxiedComponent*. The next step is to add a code segment invoking the adapter in the application.

5 Assessment and Conclusion

The adapter patterns proposed in this paper result in high reusability and profitability for the service providers and high ROI for the service consumers. We also should consider the performance penalty and additional efforts to develop adapters when adopting the adapter patterns. Hence, if a consumer's application imposes strict QoS requirements, it may be better not to use the service after adaptation. And, the consumers should analyze business cases by comparing the cost to develop application without services and with services and adapters.

In this paper, we identify six types of recurring mismatch in discovering services and define the roles of service providers and consumers to support adaptability. Each adapter pattern is specified in terms of its overview, applicable situation, structure, collaboration, consequence and instruction. By using the adapter patterns, service providers could develop highly reusable services, and service consumers will be able to reuse more services available.

References

[1] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley Professional, Reading (1994)

[2] Kongdenfha, W., Motahari-Nezhad, H.R., Benatallah, B., Casati, F., Saint-Paul, R.: Mismatch Patterns and Adaptation Aspects: A Foundation for Rapid Development of Web Service Adapters. IEEE Transactions on Services Computing 2(2), 94–107 (2009)

[3] Benatallah, B., Casati, F., Grigori, D., Motahari-Nezhad, H.R., Toumani, F.: Developing Adapters for Web Services Integration. In: Pastor, Ó., Falcão e Cunha, J. (eds.) CAiSE 2005. LNCS, vol. 3520, pp. 415–429. Springer, Heidelberg (2005)

[4] Sam, Y., Boucelma, O., Hacid, M.: Web Services Customization: A Composition-based Approach. In: In Proceedings of IEEE International Conference on Web Engineering (ICWE 2006), pp. 25–31 (July 2006)

[5] Erl, T.: SOA Design Patterns. Prentice Hall, Englewood Cliffs (June 2008)

[6] MacKenzie, C., Laskey, K., McCabe, F., Brown, P., Metz, R. (eds.): Reference Model for Service Oriented Architecture 1.0, OASIS Standard, October 12 (2006), `http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf` (accessed September 21, 2009)

[7] Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual, 2nd edn. Addison-Wesley, Reading (2005)

[8] Kim, S.D., Min, H.G., Rhew, S.Y.: Variability Design and Customization Mechanisms for COTS Components. In: Gervasi, O., Gavrilova, M.L., Kumar, V., Laganá, A., Lee, H.P., Mun, Y., Taniar, D., Tan, C.J.K. (eds.) ICCSA 2005. LNCS, vol. 3480, pp. 57–66. Springer, Heidelberg (2005)
[9] Rozanski, N., Woods, E.: Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives. Addison-Wesley, Reading (2005)
[10] Giunchiglia, F., Shvaiko, P.: Semantic Matching. The Knowledge Review 18(3), 265–280 (2003)
[11] Yeh, P.Z., Porter, B., Barker, K.: Using Transformations to Improve Semantic Matching. In: Proc. 2nd Int'l Conf. Knowledge Capture (K-CAP 2003), pp. 180–189 (2003)