# Preparing for Service-Oriented Computing: a composite design pattern for stubless Web service invocation

Paul A. Buhler[1], Christopher Starr[1], William H. Schroder[1], and José M. Vidal[2]

[1] College of Charleston, Dept. of Computer Science,
66 George Street, Charleston, SC 29424, USA
{buhlerp, starrc, schroderw}@cofc.edu

[2] University of South Carolina, Computer Science and Engineering
Columbia, SC 29208, USA
vidal@sc.edu

**Abstract.** The ability to dynamically bind to Web services at runtime is becoming increasingly important as the era of Service-Oriented Computing (SOC) emerges. With SOC selection and invocation of Web service partners will occur in software at run-time, rather than by software developers at design and compile time. Unfortunately, the marketplace has yet to yield a predominate applications programming interface for the invocation of Web services. This results in software that is deeply ingrained with vendor-specific calls. This is problematic because Web service technology is changing at a rapid pace. In order to leverage the latest developments, code often needs to be heavily refactored to account for changing invocation interfaces. This paper explores the mitigation of this problem through the application of software design patterns. Specifically, it details how a Web service architectural pattern, based upon the composition of software design patterns, provides for implementations that insulate the application code from the peculiarities of any specific vendor's interface.

## 1 Introduction

Distributed computing is undergoing revolutionary change as the worlds of Service-Oriented Computing (SOC), Multiagent Systems (MAS), and Business Process Management (BPM) converge. Web services will be the foundational technology that will underpin future distributed, internet-based computing systems. As the Semantic Web matures, Web services will routinely advertise a semantically rich description of their capabilities. These descriptions will likely be encoded in OWL-S, a semantic markup language designed for Web services [1]. Exploitation of these trends will require agile software structures that support the loosely coupled interaction of services that are found and bound at run-time. Much theoretical and practical work remains to transform this vision into reality, as change needs to occur at both the infrastructure and application levels.

It has been said that the only thing constant is change. The only consistency in the Web services space is provided by the reliance on a core set of open standards: HTTP, SOAP, WSDL, and UDDI. Fortunately, the activities of most application developers do not occur at the level of the standards themselves. Software developers generate robust software systems by harnessing the power of software toolkits, which insulate them from the standards. Unfortunately, the marketplace has yet to yield a predominate toolkit represented by a standard Applications Programming Interface (API) for dynamic, fully stubless invocation of Web services.

The lack of a standard API results in software that is deeply ingrained with vendor-specific API's. Generally, when working with stable technologies this does not present a problem; however, Web service technology is changing at a rapid pace with new tools and techniques frequently becoming available. In order to take advantage of these latest developments, code often needs to be significantly refactored to account for changing interfaces. To compensate for this problem, we have used software engineering principles and software design patterns to create an composite pattern, which insulates Web service client code from the peculiarities of any specific vendor interface. This approach enhances code stability while providing the flexibility to experiment with various approaches to the dynamic invocation of Web services.

This paper motivates the discussion of our design with an example of a typical dynamic invocation interface. It continues by providing a brief introduction to software design patterns, followed by a discussion of their applicability to Web service technologies. Interestingly, just as meta-services can be achieved through the aggregation of more primitive Web services, software design patterns can also be composed to form larger abstractions. These aggregates can be recursively composed and analyzed, resulting in resilient software architectures. This background information is used to facilitate a discussion of our prudent use of software design patterns in a successful attempt at increasing the genericity of Web service client code. The paper proceeds with an examination of an instantiation of our architectural pattern and briefly reports on the use of Axis, WSIF, and JROM as a collective Web service invocation toolkit. Finally, the conclusion summarizes our contribution and discusses the pedagogical utility and future directions for this work.

## 2 A Motivating Example

Web service invocation follows the traditional Remote Procedure Invocation (RPI) integration pattern as described in [2]. When viewed generically, RPI is an integration style that achieves Application to Application (A2A) integration by allowing one application to invoke a function published by a second application. The function in application two, appears as a local function to application one. The underlying mechanism which generates this transparency is based upon providing a function stub to application one, that when called accesses a middleware layer which transports the call and its associated data to application two. The generation of stub functions is typically automated, with tools consuming an interface description of the target function and creating the stub veneer. From a Java Web service perspective, the interface description is the WSDL file and the generation of stubs occurs with a tool such as

WSDL2JAVA. The stubs are typically generated during the coding stage of application development. The reason for this is intuitive; the stubs are called directly from the application code and need to be resolved at compile time.

SOC is based upon an underlying Service-Oriented Architecture (SOA). Figure 1 provides a description of the relationship amongst the major components of a SOA. The SOA provides the mechanisms for Web service partners to be located and invoked at run-time. This obviously requires that more flexible integration styles be developed to support the dynamic publish-find-bind pathways. Functionally, most Web service toolkits provide some capability for late, run-time binding to Web services. For example, the Glue toolkit from WebMethods [3] provides an IProxy class that can bind to a WSDL description and invoke operations. Similarly, the Web Services Invocation Framework from the Apache project [4] allows for dynamic invocation, as does the JAX-RPC package which is part of the J2EE Web services Developer Package [5].

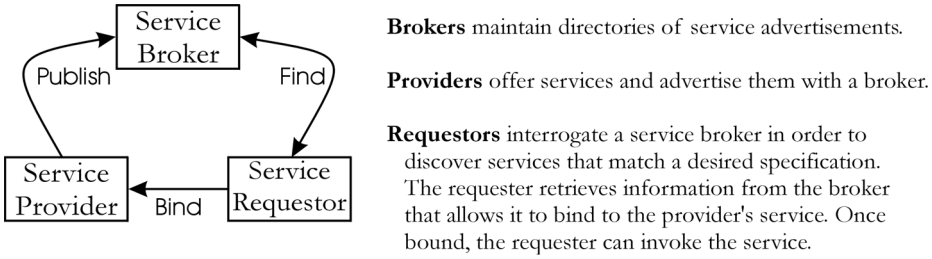


**Fig. 1.** An illustration of the major functional components of a Service Oriented Architecture.

Unfortunately, seamless dynamic invocation is beyond the capability provided by these toolkits for the simple reason that they are incapable of handling complex types returned from the invoked service. This limitation is due to the fact that the returned data must be unmarshalled from the SOAP message, which in Java is not possible without having a compatible class that implements the serializable interface. In the absence of appropriate classes, the Java run-time environment generates a java.rmi.UnmarshalException. Ironically, the stub generation tools that are not required for dynamic invocation provide these missing classes.

By way of example, a publicly available Web service provides the current weather forecast for a US zip code via a complex type [6]. Without generating a stub to hold the returned WeatherInfo, code that invokes the Web service will generate an unmarshall exception. Ideally, there would be a uniform mechanism for handling this problem; however, each toolkit has its own workaround. A singular solution will not be developed until there is broad realization of this problem. Statements such as, “The benefits of using dynamic proxies instead of generated stubs are not clear – it’s probably best to stick with generated stubs”[7, pg 339] only exacerbate the situation. Ultimately, the net result is that Web service client code becomes highly dependent upon specific toolkit APIs, which leads to tighter coupling between the code and toolkit than is desirable. The following Java code snippet illustrates a typical workaround utilizing the webMethods Glue toolkit.

```
public class dynamicInvocationExample
{
  public Document dynamicInvocationWithGlue()
    throws Throwable {
    String wsdlName =
      "http://www.ejse.com/WeatherService/Service.asmx?WSDL";
    String operation = "GetWeatherInfo";
    String args[] = { "29424" };

    // create a SOAP interceptor
    SOAPInterceptor responseHandler = new SOAPInterceptor();

    // register the interceptor to catch incoming responses
    ApplicationContext.addInboundSoapResponseInterceptor(
       (ISOAPInterceptor)responseHandler );

    try {
      // obtain a proxy to the Web service via its WSDL
      IProxy proxy = Registry.bind( wsdlName );

      // stubless invoke of the operation
      proxy.invoke( operation, args );
    }
    catch( java.rmi.UnmarshalException e ) {
      // do nothing, the UnmarshalException is expected
    }

    // generate an XML document containing the SOAP body
    return new Document( responseHandler.getResponse() );
  }
}

class SOAPInterceptor implements ISOAPInterceptor {
  private Element soapBody;

  public void intercept( SOAPMessage message,
     Context messageContext ) {
    try {
       soapBody = message.getBody();
    }
    catch( Exception e ){
      System.err.println( e.toString());
    }
  }

  public Element getResponse(){
    return soapBody;
  }
}
```

In the above code sample, it can be seen that the code negates the effect of the unmarshall exception by catching it. The SOAP interceptor captures the result of the Web service invocation. The dynamicInvocationWithGlue() method, returns a standalone XML document which contains the body of the SOAP response message. The application can access the returned data via standard XML processing functions by

loading the document into a DOM tree. It is worth noting that this sample Glue-based client uses the toolkit at two levels of abstraction, the Glue level for the dynamic invocation, and the SOAP level for the capture and handling of the response. It should be apparent that this code is not portable to another toolkit and would need to be heavily refactored if a toolkit change was required.

In order to complete the explanation of what transpires, a sample of a returned XML document follows:

```
<?xml version='1.0' encoding='UTF-8'?>
<soap:Body>
  <GetWeatherInfoResponse
    xmlns='http://ejse.com/WeatherService/'>
    <GetWeatherInfoResult>
      <Location>Charleston, SC</Location>
      <IconIndex>11</IconIndex>
      <Temprature>43°F</Temprature>
      <FeelsLike>35°F</FeelsLike>
      <Forecast>Light Rain</Forecast>
      <Visibility>Unlimited </Visibility>
      <Pressure>29.99 inches and falling</Pressure>
      <DewPoint>40°F</DewPoint>
      <UVIndex>2 Minimal</UVIndex>
      <Humidity>89%</Humidity>
      <Wind>From the Northeast at 18 gusting to 23 mph</Wind>
      <ReportedAt>Charleston, SC</ReportedAt>
      <LastUpdated>Wednesday, February 25, 2004, at 2:56 PM
          Eastern Standard Time.</LastUpdated>
    </GetWeatherInfoResult>
  </GetWeatherInfoResponse>
</soap:Body>
```

Since the application needs to handle the XML directly, it is pertinent to question how the application knows what types are represented by the returned data. The answer to this question is found in the Web service's WSDL file, which provides an XSD definition for the response message. The definition of the of the GetWeatherInfoResult is found below:

```
<s:complexType name="WeatherInfo">
  <s:sequence>
     <s:element name="Location" type="s:string" />
     <s:elementname="IconIndex" type="s:int" />
     <s:element name="Temprature" type="s:string" />
     <s:element name="FeelsLike" type="s:string" />
     <s:element name="Forecast" type="s:string" />
     <s:element name="Visibility" type="s:string" />
     <s:element name="Pressure" type="s:string" />
     <s:element name="DewPoint" type="s:string" />
     <s:element name="UVIndex" type="s:string" />
     <s:element name="Humidity" type="s:string" />
     <s:element name="Wind" type="s:string" />
     <s:element name="ReportedAt" type="s:string" />
     <s:element name="LastUpdated" type="s:string" />
  </s:sequence>
</s:complexType>
```

This example illustrates the need for a Java-typed, in-memory representation for instances of XML Schema typed data. The Java Record Object Model (JROM) [8] provides an intermediate in-memory representation that is not object-based, provides more convenience than dealing with XML directly, and most importantly supports Java's type system. As pointed out in [9] JROM can be used to hold XSD-defined XML data in memory. JROM utilizes a tree structure and is able to accommodate both simple and complex schema definitions. Simple JROM values map Schema types onto Java types, similar to the Java Architecture for XML Binding (JAXB). To illustrate the intermediate nature of a JROM representation, consider that a JROM-FloatValue is used to link the Schema type float to Java's float primitive-type. Thus the left-hand side of an assignment operator needs to be typed as a Java primitive float when storing the contents of a JROMFloatValue.

All complex types are mapped to the JROMComplexValue, a tree structure which contains simple JROM values at its leaves. The key to leveraging JROM within the context of dynamic Web service invocation is to ensure the appropriate serializers and deserializers are available to transform input and output data to JROM representation. To extend the example, the XML document containing the SOAP response message is mapped to a JROM structure, which contains Java typed data extracted from the response with the aid of the WeatherInfo XML Schema definition. At the design level the example illustrates the need for an architectural pattern that decouples the call-response system from the client invoking the Web service.

## 3 Software Design Patterns

Object oriented design patterns are design solutions to commonly occurring software design problems, codified and cataloged for reuse. The design pattern specifically names, abstracts, and identifies the key aspects of a common design problem that make it useful for creating a re-usable software component design. Patterns identify the participating classes or objects, their roles and collaborations, and the distribution of responsibilities with a focus on a particular object-oriented design problem or issue.

Design patterns are discovered through the experiences of well-practiced architects and designers, who in collaboration engage in the process of documenting and cataloging new patterns to be shared with the software community. In doing so, the set of software design patterns continues to grow, capturing elegant solutions and forming a richly expressive language for communicating designs.

The software design pattern movement was inspired by the civil architect, Christopher Alexander [10] in the early works of Eric Gamma and subsequently in the now revered Gang of Four text by Eric Gamma, Ralph Johnson, John Vlissides and Richard Helm, Design Patterns: Elements of Reusable Object Oriented Software [11]. The design patterns community has proliferated and continues to expand as people continue to discover and document both new design patterns and find new abstractions in which to apply the patterns concept.

Design patterns are intended to be reusable and adaptable to new applications in which similar design problems exist. As such, design patterns could be incorrectly

construed as reusable software components, which they are not. Both their elegance and their practical value are attributable to the abstraction of a salient solution from the source code that inspired the discovery of the pattern. The documentation of a new pattern describes the solution it provides without the details of a specific application or a programming language.

In addition to the utility of design patterns as solutions to commonly occurring design problems, design patterns tend to support certain design philosophies, which tend to be a reflection of the current consensus of the design community. Within the original 23 design patterns cataloged by the Gang of Four, the design philosophies of low coupling, high cohesion, pure typing, polymorphism over inheritance and the class substitution principle pervade. In some instances, the consequences of using a design pattern can bring about conflicts among competing design goals. For example the use of a design pattern can require the user to select between object transparency and object safety. The conflicts and tension in both design solutions and design philosophies exposed by the use of design patterns should be relished as an added benefit of patterns and not as an unnecessarily complicating issue.

Structurally an object oriented design pattern is a collection of interrelated classes or objects that provide a possible solution to a set of problems that match the intent of the pattern. The intent of the pattern provides the meaning for the pattern's structure and the realization of the pattern's motivation. The codification of each design pattern follows a template to clearly document, classify and communicate the pattern to the software community. A process of mining and polishing new design patterns is well documented and used regularly by members of the pattern community at both the PloP (Pattern Languages of Programs) and EuropeanPLoP conferences. The Hillside Group also provides a distribution mechanism for software design patterns to the user community [12].

Since their introduction to the software community as software design patterns, the pattern concept has been applied with coarser granularity at the architectural level as architectural patterns [13] and the application domain level as framework patterns [14]. The application of patterns to software development process has also seen in analysis patterns [15] and in the POAD method [16].

The utility of software design patterns for Web service design has matched particular patterns such as Adapter, Façade and Proxy to the Web service design issues of control, availability, performance, security and scalability [17]. There is an underlying assumption that design patterns can allow developers to reduce the gap between development and deployment during these times of rapid technology change.

## 4 Trends Toward Composition

Just as Web services are components intended to be composed, there is a growing realization that design patterns can be composed or aggregated into larger units. As early as 1997, it was shown that composite patterns could possess a set of characteristics that exceeded those of the individual contributing patterns [18].

In the composition of two design patterns, one pattern may use or refine the second to form a desired composite behavior, else it is said to conflict with the other pattern.

The Uses Relation, Refines Relation and Conflicts Relation, form a classification of the relationships in pattern composition [19]. It is expected that the majority of successful pattern compositions will exploit the Uses Relation in the composition of larger design elements.

For an engineering approach to pattern composition, a methodology for pattern composition was needed to systematically produce composites to leverage composite patterns as the building blocks of design artifacts. Behavioral and structural approaches to design pattern composition have been suggested, providing methodologies for the composition of patterns [16, p 19]. The behavioral approach is accomplished using object interaction specifications. The structural approach uses the class diagrams for each pattern to be composed. In both methods, care must be taken to ensure that the result of the composition does not introduce unexpected behaviors from the composite pattern. A correctness proof of composition can demonstrate that the expected behaviors of the contributing patterns exist in the composite and that the composition does not introduce new and unexpected behaviors [20].

Examples of the trend toward composition include enterprise integration patterns, which emphasize the architecture and design of enterprise information systems from collections of patterns relevant to that domain, and application frameworks, which contain multiple patterns with extension points for application development [21]. In these and other examples, the use of patterns and pattern compositions has catalyzed a shift in the design paradigm from OOD to a pattern-oriented design.

### 4.1 Pattern-Oriented Design

The composition of patterns provides flexible and arguably elegant design solutions, which can be used as primary design components and not just refinement for solutions during refactoring. The Pattern Oriented Technique is an object oriented design approach which introduces patterns into the design process after designers recognize in the class diagram the need for the solution a pattern will provide [22]. The notion that patterns and pattern compositions can be used as premier design components for design creation, changes the design approach to use larger-grained design components. This approach is used in the Pattern-Oriented Analysis and Design (POAD) technique [16].

### 4.2 Web Service Patterns

Design patterns specific to Web service architectures are emerging to form the basis for a POAD approach within the emerging paradigm of SOC. Specific patterns for Web services and Web service architectures have been documented to facilitate the rapid deployment of Web services in the support the communication pathways of “publish-find-bind” [23]. Addressing the problems of communication in service oriented applications is the major contribution of the Web service pattern collection. The Architecture Adapter pattern, solveing the communication problem, is one such pattern that creates a connection between layered (tiered) and service-oriented architectures.

Using an Architecture Adapter it is possible to decouple the interface dependencies between a Web service and the users of that service. The deployment of Architecture Adapters is also trivial using tools such as WSDL2Java, relieving the designer from specifying a hand-coded solution to convert the WSDL interface and methods to handle the conversion of a language-specific request to a SOAP request. [23, p69] The Architecture Adapter demonstrates the utility of design patterns in achieving loosely coupled designs in a Web service environment.

### 4.3 Stubless Web Service Invocation Architecture

The development of a truly dynamic Web service invocation implementation has been driven by the design goals of simplicity, flexibility, and extensibility. It is desirable to develop this capability based upon existing, open industry-standard Internet technologies without adding unnecessary layers of complexity. To isolate the application interface from its implementation, a Composite Pattern for Web service Invocation (CPWSI) was constructed using a composition of design patterns.

The CPWSI was designed, then initially implemented in Java as a thin-layer atop the Axis engine encapsulating the engine's client invocation details and providing a simple interface to make dynamic service calls. Knowing that the Axis platform is only one of many different platforms in the Web services space, it was important to separate this prototype's interfaces from its implementation so that various other Web service APIs could be plugged-in as needed to maximize its flexibility. To function as a generic interface for Web service invocation, the CPWSI was designed to be extensible to accommodate changing Web service invocation APIs without compile time dependencies for the application.

To achieve the desired level of flexibility and extensibility for the CPWSI, the overall class structure of the design Java code is provided by the Bridge pattern. The Bridge is a structural pattern with the intent to decouple an abstraction from its implementation so the two can vary independently [11]. The decoupling avoids a permanent binding between an abstraction and its implementation allowing the implementation to be selected or switched at runtime. In the Bridge the abstraction and the implementation are maintained in independent generalizations (inheritance hierarchies), allowing each to be arbitrarily extended without impacting the other. The Bridge provides the structure for run-time selection of Web service invocation for existing and future API implementations, bringing design stability to Web service applications in a rapidly changing technology environment.

Figure 2 illustrates the classes, associations and dependencies of the Bridge pattern, separating interface from implementation. The two distinct inheritance hierarchies are rooted by the Abstraction and the Implementor classes. The Abstraction is represents the public interface to the entity being implemented. The client class deals only with this Abstraction. As shown, it is possible to extend and refine the abstraction as needed. The Abstraction can be represented through an abstract class which provides a mechanism to aggregate references to one or more implementations provided by the Implementor class. The Abstraction's methods may be public, protected or abstract depending upon the specific context. The Implementor encapsulates

within its hierarchy all the specifics of a particular implementation of the abstraction. The implementation remains encapsulated but supports the semantics of the public interfaces of the abstraction. The flexibility of this approach is clear as several Implementors could exist and be switched in and out at runtime as needed. The Implementor can be represented through an interface with possibly many other classes realizing that interface with concrete implementations.

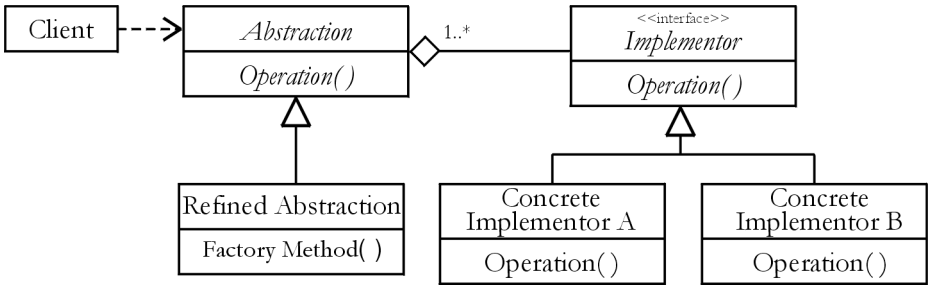


**Fig. 2.** The Bridge pattern separates the abstraction of the Web service invocation from its implementation, eliminating compile time dependencies of the APIs from the Web service application. The Bridge provides the core architectural element for the Web service invocation architecture pattern.

Additional flexibility can be derived by introducing a Factory Method pattern, illustrated in Figure 3. The Factory Method is a creational design pattern which defers the instantiation of a product to subclasses localized external to the client application [11]. The pattern has applicability when the client cannot anticipate the class of objects it needs until runtime, which is integral to the CPWSI.

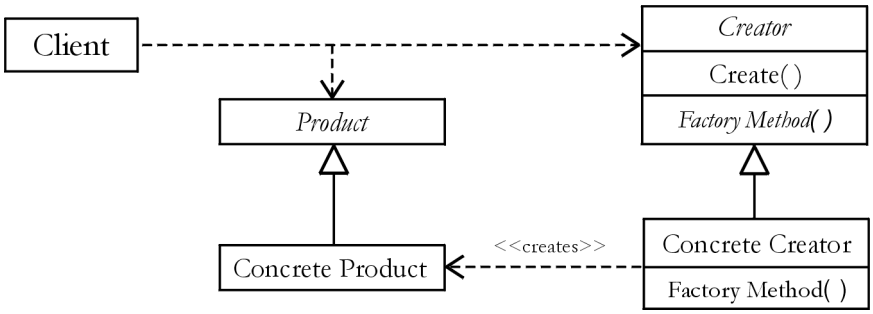


**Fig. 3.** The Factory Method pattern delegates responsibility for class creation to subclasses external to the client application, allowing the client to defer binding to a particular Web service invocation class until runtime.

The composition of the Factory Method and the Bridge further decouples clients from the concrete implementations and enables the choice of implementation to be

made based upon some runtime condition. Figure 4 shows the composite of the Bridge and Factory Method patterns. The Bridge provides the separation of responsibilities of interface specification from a potential set of implementations, allowing each to vary independently. The Factory Method provides the mechanism for isolating the instantiation mechanism of each implementation class so the client application can remain compile-time independence from all implementations, requesting a particular implementation class at runtime.

In the composition the Implementor interface of the Bridge pattern is analogous to the Product interface of Factory Method since the products of the Factory Method pattern are the implementations of the interaction represented in the Bridge pattern. Without the Factory Method the client would be more tightly bound to the implementation selection of the Abstraction. The Factory Method pattern isolates the client from the concrete implementors.

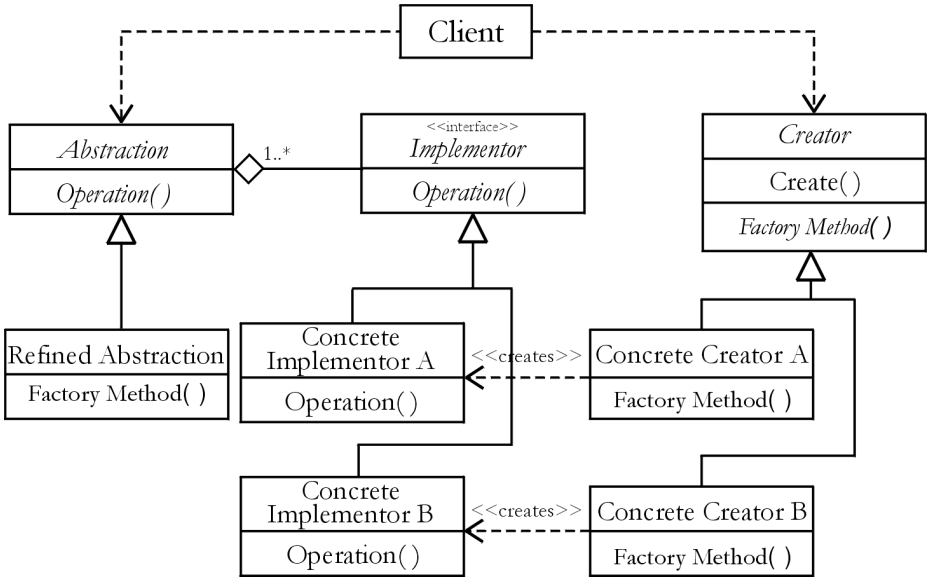


**Fig. 4.** The composition of the Bridge and Factory Method patterns produce the design architecture for stubless Web service invocation. The Bridge separates the choice of Web service APIs from a client interface for Web service invocation. The Factory Method separates the instantiation classes from the client for runtime dispatching of a particular API binding.

The structural composition of Factory Method into the Bridge also exploits the parallel class hierarchies. The class hierarchy, Implementer, of the Bridge Pattern defines the implementation classes. The class hierarchy, Creator, of the Factory Method selects the Factory Method for instantiating a particular implementation of the Product abstraction. The behavioral composition of the Factory Method with the Bridge decouples the client from the concrete implementations and enables the choice

of implementation to be made based upon some runtime condition. The create method of creator would be parameterized for the selection of a particular implementation.

## 5 An Architectural WS Invocation Pattern

The resulting composite of the Bridge and Factory Method patterns provides an agile software design for the decoupling of the invocation services that are found and bound at run-time. The composition maintains the separation between the Web service invocation technologies and the client application. Figure 5 illustrates the realization of the CPWSI.

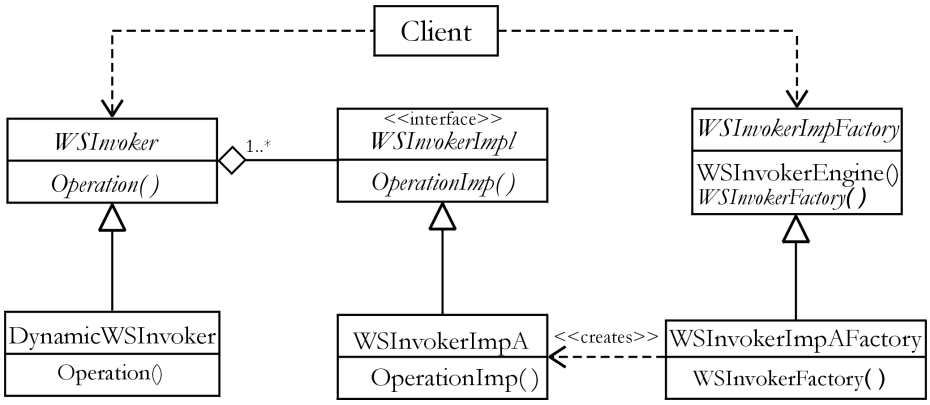


Fig. 5. The composite design pattern for stubless Web service invocation.

### 5.1 WS Invocation Pattern Instantiation

The abstract class *WSInvoker* fulfills the role of the Abstraction in the Bridge pattern with the concrete class *DynamicWSInvoker* as the RefinedAbstraction by extending *WSInvoker*. The interface *WSInvokerImpl* forms the root of the implementation hierarchy fulfilling the role of the Implementor in the Bridge pattern. For this particular prototype, the concrete class *WSInvokerImplA*, which implements the *WSInvokerImpl* interface, provides a Concrete Implementor. An abstract Factory Method class, *WSInvokerImplFactory*, encapsulates the creational details of the specific implementation, decoupling *DynamicWSInvoker* from the *WSInvokerImplA*. The *WSInvokerImplFactory* also provides a Concrete Creator class to dynamically select among a set of possible factory methods to bind one of potentially many *WSInvokerImpl* implementations. The *WSInvokerImplFactory* requires an instance of a Properties bundle, which it uses to lookup the desired implementation at runtime.

The client code need only concern itself with the public interface of *DynamicWSInvoker*. Clients which utilize the CPWSI provide a Properties instance that contains the property WSI_IMPL, which specifies the desired runtime implementation. The client is also responsible for supplying the URL for the WSDL of the desired Web service, the Service Name, Port Name, and Operation Name. These last three parameters form a tuple, which uniquely identify a specific service endpoint within the given WSDL specification. The last input to the *DynamicWSInvoker* is an array of Objects that contains the input parameters to the Web service. The order of the Objects in the array is defined by the parameterOrder attribute in the WSDL file. Upon return, the client is presented with a JROM structure that contains the Web service response. The CPWSI has been successfully used with Axis, WSIF, and JROM as a collective Web service invocation toolkit.

## 6 Conclusion

This paper has presented an engineered solution for completely stubless Web service invocation. As discussed, stubless Web service invocation will become more prevalent as the SOC paradigm is widely adopted. The CPWSI described in the paper has been used for rpc/literal Web services; work remains to be done to support doc/literal service partners. During the design process, a variety of Web service toolkits were examined in an effort to construct a minimal, yet inclusive interface for the *DynamicWSInvoker*. We anticipate that the composite design pattern will prove durable as technologies change underneath it.

From a pedagogical perspective, the CPWSI is appropriate case study material for a graduate-level software design patterns course. The CPWSI leverages the latest research on the composition of design pattern and presents a reasoned solution to an easily demonstrated problem. Our experience indicates that students are more apt to become engaged in discussions of software design patterns when they are grounded in practical problems.

Lastly, we anticipate integrating our development efforts with others who are working in the semantic web services community. To date, the activities of this group have been top-down and focused on generating the mechanisms for encoding semantically- rich descriptions of Web services. As more robust semantic matching capability emerges, the need for a stubless invocation mechanism will become obvious.

## 7 Acknowledgements

This work is supported by the U.S. National Science Foundation under grant IIS 0092593 (CAREER award).

## References


[1] The OWL Services Coalition. OWL-S: Semantic Markup for Web Services, http://www.daml.org/services/owl-s/1.0/owl-s.pdf.

[2] Hohpe, G. and Woolf, B. *Enterprise integration patterns : designing, building, and deploying messaging solutions*. Addison-Wesley, Boston, 2003.

[3] webMethods, Inc. Glue Overview, http://www.webmethods.com/solutions/wM_Glue/.

[4] Apache <Web Services /> Project. Introduction to WSIF, http://ws.apache.org/wsif/.

[5] Sun Microsystems, Inc. Java Web Services Developer Pack, http://java.sun.com/webservices/webservicespack.html.

[6] EJSE, Inc. Weather XML Web services, http://www.ejse.com/services/weather_xml_web_services.htm.

[7] Monson-Haefel, R. *J2EE Web services*. Addison-Wesley, Boston, 2004.

[8] IBM Alphaworks. Java Record Object Model (JROM), http://www.alphaworks.ibm.com/tech/jrom.

[9] Mukhi, N., Khalaf, R. and Fremantle, P. Multi-protocol Web Services for enterprises and the Grid. In *Proceeding of Euroweb 2002*, 2002.

[10] Alexander, C. *The timeless way of building*. Oxford University Press, New York, 1979.

[11] Gamma, E. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley, Reading, Mass., 1995.

[12] The Hillside Group. Patterns Library, http://hillside.net/patterns.

[13] Buschmann, F. *Pattern-oriented software architecture : a system of patterns*. Wiley, Chichester ; New York, 1996.

[14] Johnson, R.E. Documenting frameworks using patterns. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, ACM Press, 63-70, 1992.

[15] Fowler, M. *Analysis patterns : reusable object models*. Addison Wesley, Menlo Park, Calif., 1997.

[16] Yacoub, S.M. and Ammar, H.H. *Pattern oriented analysis and design : composing patterns to design software systems*. Addison-Wesley, Boston, MA, 2004.

[17] Hewlett-Packard. Applying Design Issues and Patterns in Web Services, http://www.devx.com/enterprise/Article/10397.

[18] Riehle, D. Composite Design Patterns. In *Proceedgins of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, ACM Press, 218-228, 1997.

[19] Noble, J. Classifying relationships between object-oriented desgin patterns. In *Australian Software Engineering Conference (ASWEC)*, 1998.

[20] Dong, J. Representing the Applications and Composition of Design Patterns in UML. In *Proceedings of the 2003 ACM Symposium on Applied Computing*, ACM Press, 1092-1098, 2003.

[21] Larsen, G. Designing component-baed frameworks using patterns in the UML. *Communications fo the ACM*, *42*(10):38-45, 1999.

[22] Ram, D.J., Raman, K.N.A. and Gururasad, K.N. A pattern oriented technique for software desgin. *ACM SIGSOFT Software Engineering Notes*, *22*(4):70-73, 1997.

[23] Monday, P.B. *Web Service Patterns: Java Edition*. APRESS, 2003.