

Integrating ServiceJ with The Web Service Invocation Framework

Sven De Labey Eric Steegmans

Report CW 503, October 2007



Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Integrating ServiceJ with The Web Service Invocation Framework

Sven De Labey Eric Steegmans

Report CW503, October 2007

Department of Computer Science, K.U.Leuven

Abstract

The Web Services paradigm is promising because it excels in uniting systems that were previously thought to be incompatible. Hence, a growing number of applications are built as *interconnected, cooperating services*, where the business logic of each service is typically implemented in an *object-oriented programming language*.

But current object-oriented programming languages do not integrate well with the Web Services paradigm. Various frameworks were proposed to increase transparency, but these APIs fail to provide appropriate abstractions for guiding service selection and for handling service failures. Hence, they make the programmer responsible for dealing with infrastructural technicalities, resulting in ad-hoc solutions that comprise error-prone and obfuscated code.

In this paper, we evaluate the Web Service Invocation Framework (WSIF), a popular API for writing web service clients in Java. We show how the weaknesses of WSIF can be solved by integrating it with *ServiceJ*, our Java extension that introduces language features for interacting with remote services. By directly integrating support for web services in object-oriented programming languages, we allow programmers to focus on the implementation of the business logic, rather than on technical challenges such as locating services, installing service communication protocols, and detecting service failures.

Keywords : Interoperability, Language Extension, Service Oriented Computing

Integrating ServiceJ with The Web Service Invocation Framework

Sven De Labey and Eric Steegmans

University of Leuven, Dept of Computer Science
200A Celestijnenlaan, B-3000 Leuven, Belgium
{svendl,eric}@cs.kuleuven.be

Abstract. The Web Services paradigm is promising because it excels in uniting systems that were previously thought to be incompatible. Hence, a growing number of applications are built as *interconnected, cooperating services*, where the business logic of each service is typically implemented in an *object-oriented programming language*.

But current object-oriented programming languages do not integrate well with the Web Services paradigm. Various frameworks were proposed to increase transparency, but these APIs fail to provide appropriate abstractions for guiding service selection and for handling service failures. Hence, they make the programmer responsible for dealing with infrastructural technicalities, resulting in ad-hoc solutions that comprise error-prone and obfuscated code.

In this paper, we evaluate the Web Service Invocation Framework (WSIF), a popular API for writing web service clients in Java. We show how the weaknesses of WSIF can be solved by integrating it with *ServiceJ*, our Java extension that introduces language features for interacting with remote services. By directly integrating support for web services in object-oriented programming languages, we allow programmers to focus on the implementation of the business logic, rather than on technical challenges such as locating services, installing service communication protocols, and detecting service failures.

1 Introduction

Enterprise systems rarely comprise a homogeneous combination of application services that were built for the *same platform* and that communicate via a *standardized protocol*. Instead, a service architecture is a network of interconnected services running on various platforms such as .Net [1] or the Java Enterprise Edition [2]. *Platform heterogeneity* is therefore increasingly becoming an issue. This kind of heterogeneity does not only occur during cross-enterprise operations, but also in intra-organizational enterprise systems that arose as the result of repeated mergers and acquisitions, where service architectures of acquired companies, like pieces of a puzzle, are combined so as to create a single, unified enterprise SOA.

The Web Service paradigm is currently receiving attention from the broad public as it excels in bringing together diversified platforms that were previously

thought to be incompatible. Based on standardized protocols (such as SOAP) and described by platform-independent documents (such as WSDL), web services allow different software applications to communicate and cooperate even though their technical foundations such as the runtime environment and the middleware may be very heterogeneous. From this point of view, the introduction of the Web Services paradigm can be seen as a successful attempt to hide the heterogeneity of enterprise computing platforms.

But there are other challenges as well. Next to platform heterogeneity, a vast *multitude of protocols* creates additional challenges for the implementors of service-oriented applications because each service must be invoked using its pre-defined protocol. This is either a well-known protocol such as CORBA, RMI/I-IOP and SOAP, or a specialized, vendor-specific protocol that further jeopardizes interoperability. The Web Services paradigm does not propose a solution for this problem as its underlying protocol, SOAP, is hardwired. Therefore, other frameworks such as the Web Services Invocation Framework [3] were created for hiding protocol heterogeneity.

Next to platform heterogeneity and protocol diversity, the *distributed and volatile nature of services* creates even more challenges for the developers of service applications. Network problems or unanticipated server outages, on one hand, may cause the service provider to become unreachable. Incompatible service updates and service migration, on the other hand, may break clients when they statically depend on the old version of that updated service.

The three challenges discussed here –platform heterogeneity, protocol diversity, and service volatility– create an impressive gap between *interactions with local objects* on one hand and *invocations on remote services* on the other hand. These challenges do not hold between local Java objects, for instance, because they reside in the same Java Virtual Machine. They are invoked using a native, transparent protocol and any incompatible updates to their public interface are immediately signaled as compile-time errors at those places in the code where operations are called that are no longer provided by the updated class. Interactions between local Java objects and *remote services*, on the other hand, are much more challenging because remote services have questionable availability, employ dissimilar protocols, and can be updated without warning their clients.

Currently, object-oriented programming languages such as Java are poorly integrated with the paradigm of Service-Oriented Computing. Java fails to provide appropriate support for dealing with the challenges that arise when interacting with services. This lack of specialized support implies that the programmer is made responsible for hardcoding ad-hoc strategies for solving these challenges each time a service is to be invoked. Programmers are forced to devise their own solutions for discovering and selecting services, handling protocol diversity, creating input and output messages, and anticipating problems that relate to the distributed and volatile nature of a remote service endpoint. Often, programmers must invoke operations on *port types* in the same way read/write operations had to be executed on *sockets* years ago. This results in a considerable amount of boiler-

erplate code in order to invoke a remote endpoint, thus obfuscating the business logic of an application. The resulting code entanglement decreases the comprehensibility of the source code, as such prolonging maintenance cycles as well as increasing the odds for introducing bugs in either the business logic or in the code that deals with infrastructural SOA-related technicalities.

The problems resulting from a too low level of transparency explain the growing need for a programming model that creates a cleaner separation between the *middleware* and the *application*. That programming model should equip the implementors of the business logic with the right level of abstraction to build and interact with services, leaving all infrastructural challenges to the compiler and the middleware.

In this paper, we build a programming model that integrates object-oriented programming languages and Web Service programming. We start from the Web Service Invocation Framework (WSIF) to show how Java can be used to interact with web services. We discuss how this framework frees the programmer from having to deal with protocol-specific and platform-dependent service invocations. Then, we show how the Web Service Invocation Framework can be integrated into *ServiceJ*, our Java dialect that provides specialized concepts for interacting with services at a higher level of abstraction. Successful integration of Web Services and Object-Oriented programming leads to a higher level of abstraction, since programmers are able to focus on the business logic of the application, rather than on infrastructural issues related to web service invocations.

This paper is structured as follows. Section 2 enumerates the challenges of Service-Oriented Computing that are relevant for our integration project. Closely related to these challenges is a list of design goals, discussed in Section 3. Section 4 then describes the Web Service Invocation Framework and points out its strengths and weaknesses according to the aforementioned design goals. Section 5 shows how these weaknesses can be cured by integrating the WSIF project into *ServiceJ*, our Java dialect. Section 7 shows how *ServiceJ* is compiled to Java. Section 8 illustrates how *ServiceJ* and WSIF join forces in order to increase the level of abstraction for programmers. Finally, Section 9 describes related work and Section 10 concludes.

2 The Challenges of Service Oriented Computing

Every new programming paradigm brings its own technical challenges. The Web Services paradigm is not different from this point of view, and as this paper proposes abstractions for *interacting with remote services*, we consider only those challenges that are relevant for successfully interacting with a service endpoint or a *service provider*:

- **Distributed.** Services are modular components that typically live in distributed environments. This is particularly true for *web services*, as this paradigm was devised for executing cross-enterprise operations. This creates a major challenge for the developers of *service clients*. Such a client application can never take for granted the availability of the remote service since

various problems can occur that are not necessarily under the control of the client or the service provider. These are some typical examples:

- *Link failures.* Somewhere between the client and the service, a network link may be broken, as such obstructing message exchanges between both participants. This gives rise to *timeouts* at the service requester, so programmers must insert code for dealing with such failures.
 - *Network Partitioning.* Severe router crashes and link outages may cause network partitioning, leaving the client and the provider in different partitions. Typically, the client will be able to communicate with some of its partners, where other interactions result in request timeouts.
 - *Server downtime.* Even in the absence of network problems the service provider may be unreachable because its server is offline. On one hand, these downtimes can be anticipated, for example, when a server needs maintenance actions. But even such expected suspensions are not communicated to business partners, so these outages cannot be predicted by external service clients. Additionally servers may go down due to external attacks such as hostile takeovers or Denial of Service attacks. Obviously, such outages cannot be anticipated by either the service provider or the consumer.
- **Volatile.** Service oriented computing advocates a programming paradigm in which different business partners cooperate by exposing business processes as services. This creates a multi-enterprise service architecture that is inherently decentralized. Every business partner decides how its own services evolve over time, and such decisions may deeply affect the proper working of other business partners. Below are some examples of how a service provider can affect the proper working of its services' consumers:
- *Incompatible Service Updates.* A classical problem in multilateral cooperations is that of incompatible updates. This occurs when one service provider decides to update a given service, making that newer version *incompatible* with the previous version. This older version is typically discarded shortly after the newer service becomes operational, and this breaks all clients that statically depend on the older version. If these clients do not have a built-in failover mechanism, they will remain broken until someone manually fixes the problem.
 - *Service Migration.* In a distributed environment, services may be relocated from one system to the other. This can be done by system administrators, or autonomously by load-balancing algorithms. A more extreme example is when an enterprise decides to outsource or offshore those processes that are not part of its core competencies, causing services to be migrated to different parts of the world. Service migration *changes the endpoint URL of the service* and this may lead to problems at the service requester. The latter typically downloads a WSDL document to contact the service, but once a service is moved to another location, that WSDL document is outdated. Thus, service migrations affects and possibly breaks those clients that rely on hardwired WSDL locations.

- *Service Deprecation.* The situations mentioned above occur when changes are made to a service, but at least that service remained operational. Problems worsen when a service is actually *removed*. This is not a rarity. Businesses may discontinue products or services when they are no longer profitable and this is inevitably reflected by the services that are running on the enterprise architecture. Typically, services will be discarded without notifying clients that depend on it, eventually causing these clients to break if they can't find an alternative service.
- **Competing Landscape.** Given the problems related to the distributed and volatile nature of services, it seems inappropriate for a service client to bet on one horse. Luckily, the main benefit of service-oriented computing is that it creates a *market* where competing providers offer similar services at different quality levels. This diversity allows service clients to fine-tune service selection. That is, they can select the service that best approximates a number of *functional and non-functional requirements*. Such functional requirements relate directly to the business logic of an application, whereas non-functional concerns or *Quality of Service* constraints talk about availability, reliability, and security, among others. Quality levels of services are as volatile as their location and compatibility, so another challenge for the developers of service clients is to allow that service client to *dynamically react* to changing quality levels.
- **Platform Heterogeneity.** Services execute on heterogeneous platforms such as .Net or the Java Enterprise Edition. The heterogeneity of the programming model and the runtime platform creates additional challenges for interoperability. Indeed, true interoperability is achieved only if all the technical, platform-dependent issues are successfully hidden for all cooperating partners. The Web Services Paradigm is an important example of an attempt to hide heterogeneity by relying heavily on vendor-neutral, XML-based documents (WSDL) and protocols (SOAP [4]). Service-oriented technologies such as Jini [5] and OSGi [6], on the other hand, still rely heavily on Java-based protocols. This compromises the degree of interoperability of an enterprise system: those components that are exposed as web services will be able to communicate with other runtime platforms; those exposing itself using Jini or OSGi interfaces will not.
- **Protocol Heterogeneity.** Platform heterogeneity is a well-known challenge for service-oriented computing. Less obvious is that the underlying *communication protocol* often causes similar integration problems. Indeed, although web services allow for platform independence, they still rely on a hardwired protocol, namely SOAP over HTTP. Other services may be exposed by means of other protocols that are incompatible with SOAP, and Web Service clients would not be able to communicate with these services. In other words, if a client wants to invoke a service that is offered by two providers (one using SOAP and the other using RMI/IIOP), it will need to use an entirely different protocol to contact each service. Worse yet, if the protocol at the service endpoint is changed to another one, the client will no longer be able to speak with that service. It is clear that the main chal-

lenge in this area is to hide the technicalities of protocol selection for the programmer. The Web Service Invocation Framework is a first step in that direction and it is discussed in Section 4.

This is not an exhaustive list of the challenges that come with service-oriented computing. Service orchestration and dynamic composition, for example, are other important challenges that are actively researched, but they are outside the domain of this paper as we are mainly interested in *client-service interactions*.

3 Design Goals for an Integrated Language

Our main objective is to design a programming model that integrates Web Services programming and Object-Oriented programming. This model must foster *rapid integration* of heterogeneous software components. Our objective can be split into a number of goals:

- **Dynamic Service Binding.** The challenges stemming from the distributed and volatile nature of services create a need for *late service binding*. The location of those services a client application cooperates with should not be hardwired in the source code. Hardwiring service references not only neglects the volatility of a service, it also obstructs the adaptability and extensibility of a service client. Indeed, reusing such clients in another context with different providers would be impossible without first revising every old service reference to make it point to the new provider. Dynamic service binding effectively avoids these problems by wiring service references *at runtime*. This is typically done by querying an external registry of service URLs. *Service volatility* is handled by updating the service address in the store, which does not require revisions to the source code. Store updates can be carried out by a system administrator or by an automated monitoring system.
- **Transparent Failover.** A programming model that lacks support for solving distribution problems forces programmers to insert code for handling invocation timeouts and for reinvoking failed operations on other (possibly replicated) service endpoints. As every remote service invocation must be decorated with failover code, the resulting source code becomes an obfuscated mix of business logic and code for non-functional concerns, as such reducing the comprehensibility of the application. This in turn increases the odds for introducing bugs during maintenance operations or software updates. Therefore, it is important that programmers are exonerated from having to deal with failures that are caused due to network problems or server outages. Such problems should be handled *transparently* and the instructions required for realizing this behaviour should be injected *by the compiler*, not by the programmer. For any given service reference that cannot reach its target web service, the middleware should transparently select another target service from a store of alternatives before reinvoking the operation. Programmers, on the other hand, should be able to write the business

logic as if every operation is executed locally, i.e., without having to face the technicalities and challenges of remote method invocation.

- **Constraint-driven Service Selection.** Section 2 pointed out that multiple providers often expose a similar service at various quality levels, and that these levels evolve over time. This creates a need for *constraint-driven* service selection. This is a selection mechanism that takes into account a number of quality constraints that were specified by the programmer. One major design goal of this selection mechanism is that the selection algorithm itself should be hidden for the programmer. Indeed, contacting services and checking whether they comply with predefined business goals is not a part of the business logic, but a middleware issue, so this code would only distract the programmer from his main task. Therefore, our programming model should enable programmers to specify their business constraints and non-functional requirements, whereas our compiler should inject the necessary instructions for selecting the most appropriate service based on the restrictions that were specified by the programmer.
- **Platform & Protocol Independence.** The applications developed using our programming model must be able to communicate with various services running on different platforms. The only requirement is that the service provider exposes a *WSDL definition* of that service at a suitable location. This does not restrict our programming model to *web services* alone. Every service, be it a plain Java object, a CORBA component, or an EJB session bean, that provides a WSDL definition can be used as a service endpoint. Similarly, the applications developed using our programming model must not depend on a specific communication protocol. Such dependencies cause clients to break when a service provider unexpectedly switches to a different protocol. Instead, our programming model must allow developers to focus on the business interactions with the service provider, leaving all protocol-dependent operations to the middleware.
- **Transparency.** In addition to the previous Critical Success Factors, our programming model must hide as much technical details as possible, and middleware interactions should be banned from the source code altogether. This implies that the programmer is completely shielded from the protocol that is selected to communicate with the remote service. Also, the programmer must be shielded from the technical details of the platform on which that remote service is running. Ideally, developers are only faced with implementing the business logic, whereas all technicalities and middleware interactions are injected by the compiler.
- **Higher Level of Abstraction.** Next to making middleware technicalities transparent, our programming model should also provide additional abstractions for implementing interactions with remote services. General purpose programming languages provide basic concepts for interacting with services, but these primitives cannot deal with the challenges mentioned in Section 2. The Java Enterprise Edition, for example, introduces the `@WebServiceRef` annotation to inject a web service endpoint into a variable, but this service reference is hardwired, making the code less reusable and more vulnerable for

distribution problems and volatility issues. Specialized language constructs can solve these problems because they can trigger the compiler to inject additional code at those places where service interactions occur. Ideally, the programmer is given declarative language concepts for specifying functional and non-functional *service constraints*, as well as concepts for further *optimizing service selection*.

The challenges of Section 2 led to the design goals discussed in this section. The following section provides an overview of the Web Service Invocation Framework along with an evaluation on how it realizes some of these design goals. Section 5 then shows how shortcomings can be healed by integrating WSIF with ServiceJ, our Java extension for service interactions.

4 Web Service Invocation Framework

The Web Service Invocation Framework (WSIF) [3] was initially developed by IBM and has later been donated to Apache as an open source project. WSIF's main objective is to introduce a clear and simple API to invoke web services in a *protocol-independent way*. Doing so, WSIF tries to combine protocol independence with the platform independence property of web services. WSIF uses the term “web services” in a very broad sense: every resource that describes its functionality in a *WSDL document* can be invoked by the framework. Thus, it is possible to take any existing legacy component that was not intended to be used as a web service, extract a WSDL definition from it, and contact it using generic WSIF operations. In Section 4.1, we show how WSIF allows programmers to build protocol-independent service clients. Section 4.2 then evaluates the degree of integration between WSIF-driven Java applications and the Web Services paradigm.

4.1 Achieving Protocol-Independence with WSIF

Figure 1 shows a simple service architecture not using the Web Service Invocation Framework. A Java client interacts with two remote services and is forced to use different communication protocols to invoke operations on each service. The first service provider, an EJB Session Bean, must be invoked using the native RMI/IIOP protocol, whereas the second service provider, the BPEL process, requires message exchanges over the SOAP protocol. The main problem with this protocol-dependent approach is that programmers must be familiar with both protocols. They must know about JNDI lookups so as to get a remote reference to the Session Bean, and they need to learn an API (such as Apache Axis) for interacting with services that expose their interface using WSDL. A second problem is that the Java client breaks if the service provider switches to another protocol (or protocol version). This is because the protocol used for invoking operations on a service endpoint is hardwired in the business logic of the service client. Such protocol changes are problematic in the context of cross-enterprise operations, where internal decisions to update the communication infrastructure are not necessarily communicated back to all the business partners (the clients).

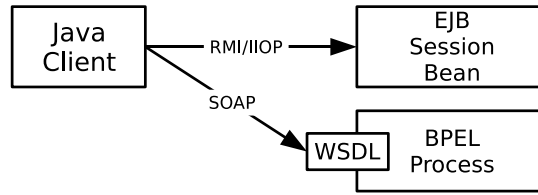


Fig. 1. Basic service architecture *not using WSIF*

Stub-based Invocation. Figure 2 shows the same service architecture, now using the WSIF framework for implementing service invocations. The communication protocol that is used to invoke remote services, is completely hidden behind a *stub*. This stub is created using a WSDL2Java stub compiler that comes with the WSIF framework. Stubs allow for type-safe invocations on remote services and they provide a higher level of abstraction by hiding all protocol-specific details inside a generated stub, which can typically be left unchanged by client programmers. These WSIF stubs can thus be compared to RMI stubs with the added benefit of protocol independence.

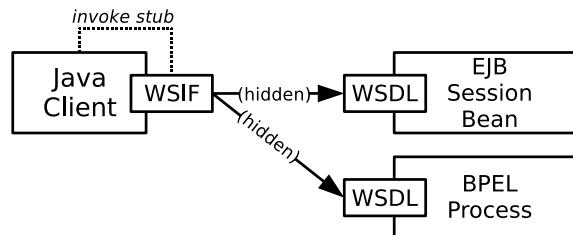


Fig. 2. Basic service architecture *using WSIF*

Example. Listing 1.1 presents an example of a service client using WSIF's stub-based approach for web service invocation. It shows the code for invoking the `getQuote` operation that is published by the `StockQuote` web service. Interactions with that `StockQuote` service are carried out in a protocol-independent way, and they can be programmed using the following steps. First, a service factory instance is created (line 3). This factory is then used to create a `WSIFService` instance (lines 6–13). This is a local object representing the remote web service endpoint (it can be seen as a *reified* version of that web service). The `WSIFService` object is created by the `WSIFServiceFactory` instance based on the WSDL definition of the service endpoint and some additional information. Using their `WSIFService` object, programmers request a *stub* (line 16), which is used to interact with the web service (line 18). Note also that two exceptions can

```

1 try {
2     // --1-- Create a service factory
3     WSIFServiceFactory factory = WSIFServiceFactory.newInstance();
4
5     // --2-- Parse WSDL
6     WSIFService service =
7         factory.getService(
8             "/example/StockQuote.wsdl", // WSDL location
9             null,                       // Service Namespace
10            null,                       // Service Name
11            "http://...url...",        // PortType Namespace
12            "StockQuotePortType"      // PortType Name
13        );
14
15    // --3-- Get the stub
16    StockQuote stub = (StockQuote)service.getStub(StockQuote.class);
17
18    // --4-- interact with the service
19    float quote = stub.getQuote(aQuote);
20    System.out.println(quote);
21 }
22 catch (WSIFException we) {
23     we.printStackTrace();
24 }
25 catch (RemoteException re) {
26     re.printStackTrace();
27 }

```

Listing 1.1. Invocation of a `StockQuote` web service using WSIF stubs

be thrown during this scenario. A `RemoteException` is thrown when something goes wrong at invocation time. A `WSIFException` may occur when something in the preparation phase (e.g. WSDL parsing) fails.

Dynamic Protocol Provisioning. The technical details of the service invocation are hidden for the programmers of the service client. In order to invoke operations on remote services in a transparent, protocol-independent way, WSIF relies on *dynamic providers*. Each of these providers understands one protocol and they can be added or removed at runtime. Thus, such a pluggable provider architecture enables to dynamically extend or reduce the number of protocols that the service client can use when interacting with a remote service, at the same time minimizing programmer intervention.

Summarizing, even though stubs in WSIF are created statically, the protocol that is eventually used to invoke the service is determined at runtime, hence allowing programmers to build protocol-independent service clients.

Stubless Invocation. Next to stub-based invocation, WSIF also supports *stubless* invocation. This approach allows programmers to *dynamically create and execute* a web service invocation using functionality provided by the WSIF API. The construction of a dynamic service call is similar to (though slightly more complex than) how dynamic method calls are built using the Java Reflection API. Stubless invocation is not explained in this paper, as it firmly decreases the level of abstraction by requiring programmers to write a considerable amount of interactions with the WSIF API before a service can be dynamically invoked. Additionally, dynamic invocation provides *less compile-time guarantees*. Similar to the Reflection API, method calls are represented as *string objects* such that the compiler is unable to check whether the invocation target supports that operation.

4.2 Evaluation of WSIF

In this section, we evaluate the Web Service Invocation Framework according to the challenges and design goals that were discussed in Sections 2–3. We examine how WSIF adds to the integration between object-oriented programming and service-oriented computing. Shortcomings to this integration attempt are then tackled in subsequent sections.

Strengths. The main objective of the Web Service Invocation Framework is to provide for protocol-independent web service invocations. Hence, the major strengths of WSIF are geared for protocol *independence* and *transparency*:

- *Protocol Independence.* WSIF allows programmers to develop clients independent of the wire protocol that is used for contacting a remote service. Being independent of that protocol increases the reusability of the client and allows for deployment in heterogeneous service environments that communicate using other protocols without requiring changes to the source code of WSIF clients. This is an important feature in the world of service-oriented computing, where services from different origins are assembled in heterogeneous service architectures.
- *Protocol Transparency.* Service clients are independent of the wire protocol, and an additional advantage of WSIF is that this protocol is entirely *transparent* for programmers. This implies that the source code of the client contains less traces of infrastructure-dependent technicalities, and this allows programmers to concentrate on the implementation of the business logic, leading to a cleaner separation between functional and non-functional concerns. This in turn leads to increased code comprehension and lower chances for introducing bugs.
- *Platform Independence.* WSIF allows interactions with remote resources that expose their functionality by means of WSDL descriptions. As such, WSIF achieves a high degree of platform-independence. This goes even further than

the Web Services paradigm, which also relies on WSDL documents, but hardwires the communication protocol (SOAP) for all client-service interactions.

Weaknesses. WSIF is very effective when it comes to automated protocol provisioning in heterogeneous service architectures, but other important challenges were not dealt with. Currently no other frameworks exist that enrich WSIF with solutions for the problems that these challenges bring forward. The following is a list of shortcomings that refrain WSIF from realizing the objectives enumerated in Section 3:

- **Hardwired Service Locations.** WSIF hides the underlying communication protocol for client developers, but the framework cannot hide the *location* of the services. Developers must provide the `WSIFServiceFactory` instance with the WSDL location of the web service when invoking the `getService` operation (as shown on line 8 in Listing 1.1). Thus, WSIF cannot cope with challenges originating from the *distributed and volatile nature* of services, and this leads to a number of problems:
 - *Volatility problems.* WSIF can handle protocol volatility, but it cannot deal with varying Quality of Service levels, service migration and service deprecation. Thus, programmers must insert this non-functional code each time a remote service is invoked. This shortcoming overshadows the reduction in code size that was achieved by the benefit of protocol transparency because programmers are still forced to write code for dealing with infrastructural issues.
 - *Distribution problems.* Clients using WSIF become very vulnerable for network problems or server outages as they statically rely on hardwired service endpoints. Failover is not transparently supported by WSIF-based service interactions, so the programmer must *decorate* every remote invocation with the necessary code for invoking backup services. Obviously, such backup service references would in turn rely on the WSIF framework, so their endpoint URLs would be hardwired as well.
 - *Service Selection problems.* Relying on a single service endpoint obstructs every possibility for dynamically selecting a service among a number of alternatives. This implies that the WSIF framework cannot deal with dynamically changing Quality of Service levels, nor can it cope with the multitude of substitutable, competing services. The user must decide statically which service to invoke, and then hardwire its endpoint URL in the source code of the service client. If more interesting services are added to the service architecture, then older clients will not find these services; they will continue to use their hardwired, outdated services.
- **Lack of Transparency.** Although stubs hide most of the details of protocol heterogeneity behind a clean Java interface, WSIF is not fully transparent. As shown in Listing 1.1, programmers still need to invoke API operations in order to initialize the stub. This involves (1) creating a `WSIFServiceFactory` instance, (2) using that factory to create a `WSIFService`, and (3) requesting

a stub from that reified web service. By forcing programmer to write these API interactions, WSIF decreases the level of transparency between service interactions and local object invocations, and this again overshadows its attempt for transparent service invocations.

- **Compilation.** Every stub must be compiled separately using the WSDL2Java compiler before the application can be compiled. Also, changes to the WSDL document require manual recompilation of the stub.

In summary, the benefits of WSIF focus on providing protocol-independent invocations of web services that execute in heterogeneous environments. But the framework suffers from integration problems as well. The level of transparency is still too low, and programmers are still forced to decorate service invocations with code for handling infrastructural technicalities. In order to solve these problems, we have integrated the Web Service Invocation Framework with *ServiceJ* [7], our Java dialect with specialized concepts for web service interactions. The next section describes how ServiceJ extends Java with new language concepts. Section 7 explains how WSIF and ServiceJ are integrated, and how WSIF operations can be injected during the compilation of ServiceJ to Java.

5 ServiceJ. Abstractions for Client-Service Interactions

Driven by the need for a high-level language to implement client-service interactions, we have designed a Java extension, called ServiceJ. This dialect extends Java in two ways. *Type qualifiers* (Section 5.2) signal the compiler to inject code for service retrieval and for handling communication failures. *Declarative operations* (Section 5.3) provide programmers with high-level abstractions to fine-tune service selection according to predefined *quality attributes*, such as functional requirements or Quality of Service constraints.

But first, we motivate the need for these language concepts by pointing out how Java fails to provide adequate support for programming service interactions.

5.1 Java does not Adequately Support Service Interactions

Service references differ much from local references because most of the challenges mentioned in Section 2 only hold for those fields referencing *remote* services and not for fields pointing to *local* objects. Since Java lacks special concepts for distinguishing between both kinds of fields, their representation in the source code is the same, as shown in Listing 1.2. The first field, `employee`, is a reference to a local Java object whereas the second field, `defaultPrinter`, points to a remote `Printer` service. Java does not provide special features for separating local references from remote service references, so both declarations look the same. The main advantage of being unable to distinguish between both kinds of fields is the increase in local/remote transparency that programmers get when invoking operations on objects.

But this increased transparency is only beneficial in theory. Section 2 pointed out that the invocation of a remote service comes with much more challenges

```

1   Person employee = ...; //init field with local object reference
2   employee.getSalary();
3
4
5   Printer defaultPrinter = ...; //init field with service reference
6   try{
7       defaultPrinter.print(aFile);
8   }
9   catch(...){}

```

Listing 1.2. Java does not distinguish between fields pointing to local objects (lines 1–2) and fields referencing remote services (lines 5–9)

than merely invoking operations on a local object. These challenges are not dealt with by Java, so they must be handled by the programmer. The use of remote objects is therefore not transparent at all: every remote invocation must be *made robust manually*. Code for service selection, service binding, and service failover must be added so as to create a service client that can react to dynamic changes in an SOA. This idea is illustrated in pseudocode in Listing 1.3, which shows that remote service invocations lead to code explosion (lines 7–9 and 14) due to a lack of support offered by the programming language and its accompanying compiler.

```

1   // --1-- Local object invocation.
2   employee.getSalary();
3
4
5   // --2-- Remote service invocation.
6   // Volatility & distribution create uncertainty
7   // 1: contact a repository, ask for available Printer services
8   // 2: consider quality attributes of each Printer service
9   // 3: bind remote reference to the most appropriate alternative
10  try{
11      defaultPrinterService.print(aFile);
12  }
13  // 4: add retry logic in case of distribution problems
14  catch(RemoteException ex){
15      //try another service, or signal failure
16  }

```

Listing 1.3. Dealing with infrastructural challenges obfuscates the source code

It is obvious that this code is not elegant. The programmer is led away from his main task –implementing the business logic. Also, the comprehensibility of the source code is severely compromised, which naturally leads to error-prone code as well as drawn-out maintenance and update cycles. The Web Service Invocation Framework is an attempt to increase this low level of transparency, but as Section 4.2 has shown, it is definitely not a one-size-fits-all solution. The

next Section explains how we extend Java with specialized concepts for service interactions in an attempt to further increase the level of abstraction.

5.2 ServiceJ extends the Java Type System with Type Qualifiers

We have designed a Java extension, ServiceJ, that introduces specialized language concepts for client-service interactions such as the example given in Listing 1.3. This Section discusses how *type qualifiers* hide the technicalities of client-service interactions. Section 5.3 elaborates on this by describing how service selection can be fine-tuned at a high level of abstraction.

Type Qualifiers. ServiceJ introduces a special type qualifier to distinguish external service references from local object references. A field referring to a service reference is declared using the **pool** qualifier. An example is shown below:

```
pool PrinterService printer;
printer.print(aFile);
```

This code declares a field **printer** of type **PrinterService** decorated with the **pool** qualifier. That qualifier signals to the compiler that the **printer** field may refer to *any service* of which the *type* is compatible with the **PrinterService** type. In other words, by using the **pool** qualifier, programmers allow the ServiceJ middleware to use a *non-deterministic* service selection algorithm; they consider all service endpoints to be equal.

This service selection algorithm is executed at the middleware level, so there are no physical service endpoint references in the source code of the client. Therefore, a **pool** field can be seen as a *stub* pointing to a *single virtual service*. This service is called *virtual* because programmers are never confronted with the physical service object. This is very different from service interactions with WSIF, where programmers were forced to invoke the **getService** method of a **WSIFServiceFactory** instance in order to create a **WSIFService** object which eventually delivered the *stub*. ServiceJ hides these operations for programmers and directly provides the stub in the form of an instance field that is qualified with the **pool** qualifier. Such *pool fields* exhibit the following characteristics:

- *Transparent initialization.* Fields carrying the **pool** qualifier are never initialized by programmers. This would hardwire the field to a single service endpoint, leading to the problems that were encountered in WSIF (see Section 4.2). Instead, initialization is now the responsibility of the compiler, which injects instructions for interacting with the service selection algorithms of ServiceJ in order to inject an appropriate service reference into the **pool** field. This is similar to the Inversion of Control principle that is used in the Java Enterprise Edition in order to inject resources in *annotated* fields [2]: using these annotated fields, the EJB container is able to transparently inject the proper resource reference without forcing programmers to write low-level JNDI interactions.

- *Transparency.* Type qualifiers provide a high level of abstraction for the developers of service clients, as such exonerating developers from a lot of infrastructural tasks. Interactions for *service lookup* are now transparently injected during compilation, *service selection* is carried out in a non-deterministic way, and *service binding* is performed at runtime. These tasks can still be fine-tuned by developers, as will be discussed in Section 5.3, but programmers are no longer confronted with code for reacting to infrastructural problems.

The `pool` qualifier allows the compiler to distinguish between fields pointing to local objects and fields referring to services. This separation allows the compiler to insert special instructions at those places where services are to be invoked. We discuss three benefits that ServiceJ code achieves by making this difference: (1) solving service availability problems (2) handling service volatility and (3) dealing with service plurality.

Solving Distribution Problems. WSIF cannot handle distribution problems so programmers are forced to decorate each service invocation with failure handling logic, as such obfuscating the business logic. ServiceJ solves this problem by integrating support for *service failover* with the `pool` qualifier. If such a `pool` field becomes the target of a method invocation, the necessary logic for dealing with distribution problems is injected by the compiler. Solving an availability problem typically comprises (1) the selection of another service that conforms to the static type of the pool field, (2) binding the new service reference to that field, and (3) reinvoking the operation that was unsuccessfully invoked on the unavailable service. These instructions are injected by the ServiceJ compiler for all method invocations on fields that were qualified as `pool` fields.

Solving Volatility Problems. As discussed in Section 4.2, WSIF cannot deal with service volatility. ServiceJ provides a solution to this problem by removing from the source code all infrastructure-specific service information. Programmers declare a single *virtual service* using a `pool` field *without initialization code*. This approach increases reusability and transparency, leads to better code comprehension, and thus reduces the possibility for introducing bugs.

The non-deterministic service binding strategy of the `pool` qualifier increases load balancing and enables service failover. Often, however, developers will want to fine-tune service selection, limiting the set of assignable services to those services that exhibit specific functional or non-functional characteristics. This level of configurability is not supported by the `pool` qualifier. ServiceJ therefore introduces special operations for further constraining service selection, as discussed in the next section.

5.3 Declarative Operations for Fine-tuning Service Selection

Section 2 pointed out that service architectures often contain multiple services that provide a similar functionality at different *quality levels*. This allows devel-

opers of service clients to finetune service selection, limiting the set of invocation targets to those services that provide the appropriate Quality of Service. Object-oriented programming languages do not provide language abstractions for expressing such *selection constraints* and neither does WSIF. They put the burden of implementing service selection strategies on the programmer. This results in complex iterations over sets of compatible services, where each service is contacted so as to retrieve information about its quality level. After that information has been retrieved, programmers have to write algorithms for selecting the service that most closely approximates the requirements and preferences of the service client.

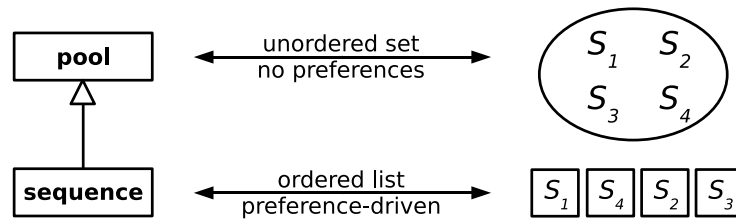


Fig. 3. A hierarchy of type qualifiers

Imposing Constraints. Service selection can be finetuned by imposing service constraints. These constraints can be *functional requirements* or *non-functional requirements*. Functional requirements relate directly to the business logic of an application, whereas non-functional requirements or *Quality of Service constraints* relate to security, availability, response times, etcetera. ServiceJ allows programmers to specify a service selection policy using a declarative operation, **where**, which accepts a *boolean condition* representing the service selection constraint. can be invoked on fields that were declared using the `pool` qualifier. For example, creating a service pool of those `Printer` services that support color printing can be written as:

```
pool Printer printer where printer.supportsColorPrinting();
```

Every boolean expression can be a service selection policy. For example, a more complex example is that of a service pool containing only those printers at the second floor that support color printing with a speed of at least twenty pages per minute can be defined as follows:

```
pool Printer printer
  where printer.supportsColorPrinting()
    && printer.getFloor().getNumber()==2
    && printer.getPagesPerMinute()>=20;
```

By introducing the declarative **where** operation, programmers are exonerated from having to search for a service that complies with their expectations. The **where** operation automates the following tasks:

- *Requesting service characteristics.* The service selection constraints (i.e. the argument of the **where** operation) is used to transparently query the service candidates about their service level. Services that are unable to fulfill this condition are removed from the candidate set.
- *Dynamic service binding.* After a service has been found that complies with the selection constraint, the **pool** field is transparently bound to that service endpoint.

Next to selection policies, a *preference order* may exist among those services that satisfy the selection constraint. Support for preferences is not supported by the **where** operation, since it only deals with boolean expressions. Therefore, ServiceJ introduces a second declarative operation, which is discussed next.

Dealing with Client Preferences. The **pool** qualifier is used to state that invocations may be sent to *any* compatible service. The **where** operation allows to constrain that set of candidate services, but both concepts rely on *non-deterministic service selection*. They assume that all services that reside in the (constrained) service set are interchangeable.

```

1 // unsorted collection, no preferences
2 pool Printer printer
3     where printer.supportsColorPrinting()
4         && printer.getFloor().getNumber()==2;
5
6 // sorted list, preference-driven
7 sequence Printer printer
8     where printer.supportColorPrinting()
9     orderby printer.getPriceFor(aFile);

```

Listing 1.4. Declarative operations for fine-tuning service selection

Sometimes, however, some services are preferred above others based on service-specific characteristics. ServiceJ introduces the **orderby** operation to specify such *client preferences*. This radically changes the service selection strategy: in stead of arbitrarily selecting a service from a constrained set, now a *deterministic* service selection policy is required. Indeed, the specification of preferences induces a *partial order* on the candidate set, where some services are preferred above others. To distinguish between both selection policies, ServiceJ introduces a second type qualifier, **sequence**, which is a *subtype* of the **pool** qualifier. Technically, a **sequence** can be seen as a *sorted list* of services, whereas a **pool** corresponds to an *unordered collection* of services, as shown in Figure 3. The following sample code selects the **Printer** service that minimizes the cost for printing a file that is referenced by the variable **aFile**:

```
sequence Printer printer orderby printer.getPriceFor(aFile);
```

Both declarative operations can be combined, yielding an expressive mechanism for guiding the service selection strategy without forcing programmers to deal with the infrastructural technicalities of service selection. The following field represents a sequence of color printers sorted by the price they ask for printing the document referenced by the variable `aFile`:

```
sequence Printer printer where printer.supportColorPrinting()
                                orderby printer.getPriceFor(aFile);
```

6 Formal Development of ServiceJ

We have proved a type soundness theorem for ServiceJ as a Java extension. We started from Featherweight Java (FJ) [8], which is a lightweight, functional calculus for Java. We have extended the formal construct of a Java *type* to a combination of a *class name* and a *type qualifier*. The class name refers to a basic Java type, whereas the type qualifier can be a `pool`, a `sequence`, or a `singleton`. The `singleton` qualifier exists only in the formal model and is used to refer to regular Java variables. A second extension to the FJ syntax was the introduction of the `where` and `orderby` expressions.

Given this extended syntax, we have defined a number of *typing rules* and *reduction rules*. Eventually, we have used these rules to prove that ServiceJ, as a Java extension, is type sound. The details of this formal development are discussed in a previous technical report [9].

7 Implementation Issues for Integrating ServiceJ & WSIF

A Java compiler translates Java source code to bytecode, which can be interpreted by the Java Virtual Machine. Our goal is to reuse both the Java compiler and the Java Virtual Machine for the development and execution of ServiceJ applications. Being able to reuse both tools is beneficial for two reasons. By reusing the Java compiler, we are able to optimize ServiceJ applications in the same way regular Java code is optimized at compile time. Additionally, by reusing the standard JVM, ServiceJ applications become interoperable with all other Java applications that run on the standard JVM. This means that existing Java code, such as libraries and APIs, can be reused in ServiceJ programs without requiring source code modification.

Obviously, simply reusing both tools to run ServiceJ programs is insufficient, as the Java compiler is unable to recognize our newly introduced language features. Therefore, we introduce a *preprocessing step* in which ServiceJ applications are *transformed* to Java applications. This transformation injects middleware interactions for guiding the integrated ServiceJ and WSIF middleware during service interactions. The overall process of compiling ServiceJ to Java is shown in Figure 4. First, the ServiceJ source code is read by a parser so as to build an abstract syntax tree (AST) represented as a ServiceJ *metamodel instance*, as described in Section 7.1. This ServiceJ metamodel instance is then transformed to a Java metamodel instance before it is compiled, as discussed in Section 7.2.

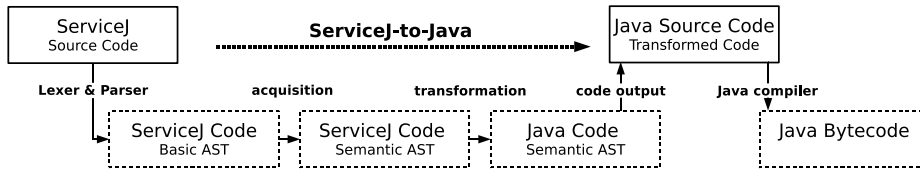


Fig. 4. Overview of the ServiceJ-to-Java transformation process

7.1 Representing ServiceJ applications in the Java Metamodel

ServiceJ applications are read by a ServiceJ Lexer and a ServiceJ Parser. Both the lexer and the parser were generated by ANTLR after the ServiceJ grammar was specified. The ServiceJ parser reads ServiceJ source code and builds an *abstract syntax tree* (AST) representation of the ServiceJ program. This AST lacks semantic information about the program, so it is very hard to transform it to an AST representing an equivalent Java program. Therefore, this basic AST is transformed to a *semantic AST*. This is an AST in which each node is a reified version of an element that occurs in the source code. An instance method, for example, is represented by an instance of the `RegularMethod` class, which can be queried for its access modifier, its formal parameters, and even its implementation. Classes, on the other hand, are represented by a class that can be queried for instance fields, methods, etcetera.

The ServiceJ metamodel is built as a specialization of *Jnome*, our metamodel for Java. The ServiceJ metamodel extends *Jnome* with special classes representing our newly introduced language concepts. These additions can be divided into two categories:

- *Additional variable modifiers.* An example of a variable modifier is the `final` modifier, which is represented by the metamodel class `Final`. This class is already defined at the level of the Java metamodel, *Jnome*, because the `final` modifier is defined in Java. ServiceJ introduces two additional variable modifiers: `pool` and `sequence`. These modifiers are represented at the level of the ServiceJ metamodel by the `Pool` and `Sequence` classes. The tool that transforms the basic AST into a semantic AST is responsible for attaching an instance of `Pool` or `Sequence` to the those metamodel elements representing fields that were qualified as `pool` or `sequence` fields.
- *Additional built-in operations.* The ServiceJ metamodel also specializes the Java metamodel by introducing two additional classes for representing the `where` and `orderby` operations that can be invoked on `pool` and `sequence` fields. These classes are subtypes of the general `Expression` type. Both declarative operations can be seen as *binary operations* accepting two expressions (the *pool target* and the *pool membership constraint*) and returning a non-void result (a reference to the *constrained service pool*).

By extending the Java metamodel, we can represent ServiceJ applications as instances of the ServiceJ metamodel. These instances contain semantic information about the program and provide operations for querying and manipulating

the metamodel. This functionality makes a metamodel instance eligible for *metamodel transformation*. In the next section, we show how a transformation from the ServiceJ metamodel to the Java metamodel is carried out.

7.2 Compiling ServiceJ to Java with WSIF instructions

While the previous section showed how ServiceJ code is represented by a metamodel instance, this section explains how such an instance is transformed to a Java metamodel, which is eventually written out as Java code before it is compiled by the Java compiler.

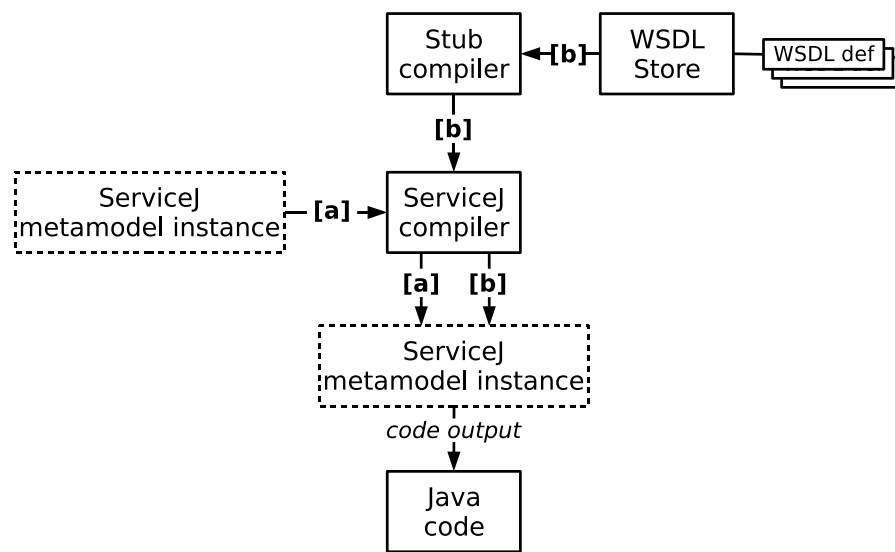


Fig. 5. Compiling ServiceJ and WSIF to regular Java code

The tool that transforms the ServiceJ metamodel instance to a Java metamodel instance is commonly referred to as *the ServiceJ compiler*. It accepts information from two sources, as shown by traces *a* and *b* in Fig. 5:

- (a) The first input is the *instance of the ServiceJ metamodel* representing the ServiceJ source code, as obtained after adding semantic information to the basic AST. This process was explained in Section 7.1.
- (b) As a second input, the ServiceJ compiler needs precompiled *service endpoint stubs* that were compiled using a WSDL-to-Java compiler (e.g. the WSDL2Java compiler provided by Apache Axis). These classes are part of a ServiceJ application, so they must be added as *types* to the ServiceJ metamodel instance.

The ServiceJ compiler starts by iterating over the ServiceJ metamodel instance, transforming each node (i.e. program element) to its corresponding Java construct. Some transformations are straightforward because they represent the same concept in both ServiceJ and Java. Classes from ServiceJ, for example, are transformed to classes in Java. Similarly, instance fields from ServiceJ are transformed to instance fields in Java. But other than Java, ServiceJ distinguishes between different *kinds of fields*. It uses a different transformation strategy for interactions with *regular instance fields* on one hand, and interactions with *pool or sequence fields* on the other hand. Method invocations on regular fields are left unmodified during the transformation whereas interactions with pool or sequence fields require the ServiceJ compiler to inject *additional instructions* for dealing with the challenges that were enumerated in Section 2. The following list gives an overview of challenges that are dealt with by code that is injected during this transformation process.

- *Service discovery.* Instructions are added to discover services of which the type is compatible to the static type of the pool field. Method invocations on a field declared with type `Printer`, for instance, will be decorated with code for discovering services that are compatible to that `Printer` type.
- *Service selection.* Programmers may have constrained the candidate set of services to those services that exhibit special characteristics using a `where` or `orderby` operation. The ServiceJ compiler injects code for constraining the set of discovered set of services using these expressions. It will query each member of the candidate set and check whether it conforms with the constraints that were specified by the programmer. Only those services that meet that quality level are retained.
- *Service binding.* The ServiceJ compiler initializes the pool field by injecting instructions for dynamically binding it to a service. This protocol depends on the *variable modifier* (`pool` or `sequence`) that was used when the field was declared. If the `pool` qualifier is used, then a *non-deterministic* service binding algorithm is used, which merely binds the field to a random service from the constrained candidate set. On the other hand, if the field is declared using the `sequence` qualifier, then a *deterministic* service binding algorithm is used, which binds the field to the service of which the characteristics most closely approximate the preferences that were specified using the `orderby` clause. Whatever protocol is used, it is important to note that service binding is done *just-in-time*. This is because service quality levels may be subject to continuous change, so it is important to use the most recent data when selecting and binding a service.
- *Service failover.* Given the distributed nature of services, all service interactions must deal with service unavailability problems. The ServiceJ compiler therefore decorates every service interaction with code for handling such failures. When confronted with remote problems, this injected code will search for another compatible service from the (constrained) candidate set and then reinvoke the operation on that new service endpoint.

The transformation from a ServiceJ metamodel instance to a Java metamodel instance is the main value-adding factor in the compilation process of a ServiceJ application. After the Java metamodel instance has been obtained, a *codewriter* turns the metamodel instance into a set of Java source files, which are eventually compiled with the Java compiler to Java bytecode. This concludes the process that is shown in Figure 5.

The above may look like a complex compilation process, but it is not our goal to build a highly optimized compiler; rather, we want to show that it is possible to extend Java with the concepts that are introduced by ServiceJ. Also, note that programmers are completely shielded from the complexity of the compilation process. The entire process can be executed without user intervention, and it is straightforward to automate the process by writing a script that automates the entire process.

8 Illustration

In this section, we provide a detailed example of how the responsibilities are divided between the developer, the ServiceJ middleware, and the WSIF middleware. We start by focusing on the task of the programmer in Section 8.1. Section 8.2 then shows how this code, after being compiled with the ServiceJ compiler, triggers the necessary actions at the ServiceJ middleware, which cooperates tightly with the WSIF middleware in order to successfully execute the operation.

8.1 The Task of the Developer

The task of the programmer in this example is to implement a service interaction for printing a file, represented by the variable `aFile`. This file must be printed on a *color printer* that offers a throughput of at least *20 pages per minute*. Multiple printers may satisfy these constraints, so the file must be printed on the printer with the lowest number of queued jobs. As shown in Figure 6, the programmer can easily implement this service interaction by using the language constructs that ServiceJ provides. A pool field `printer` is declared along with a `where` operation representing the *service constraints*, and an `orderby` operation representing the *preferences* of the service client. The resulting code is small, elegant, and statically type-checkable. Also, the programmer works at a high level of abstraction, since operations are invoked on a *virtual service endpoint*, not on a real service.

Contrary to existing Java frameworks, programmers are no longer responsible for locating physical services, nor are they forced to insert instructions for service selection, service binding, and service failover. These tasks are transparently dealt with by the ServiceJ middleware, which interacts with WSIF, as explained in the next section.

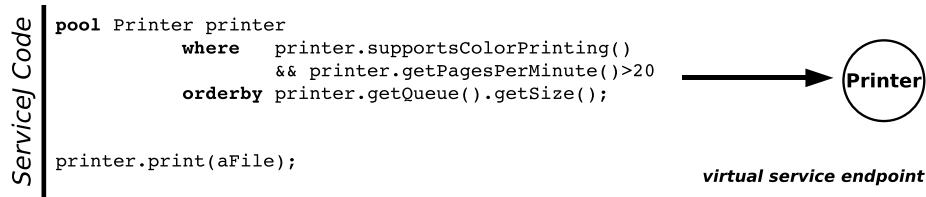


Fig. 6. Implementation of the business logic using ServiceJ language concepts

8.2 The Task of the Integrated ServiceJ-WSIF Middleware

Under the hood, ServiceJ and WSIF cooperate tightly to enable successful service invocation. These interactions are shown in figure 7. Before the `print` operation from Section 8.1 can be successfully invoked on a remote service, the ServiceJ middleware has to search for services that satisfy all the constraints that were imposed when the pool field was declared. As discussed in Section 7.2, the ServiceJ compiler is responsible for injecting middleware interactions that select the appropriate service before invoking the `print` operation.

First, a ServiceJ registry is contacted, requesting the list of URLs that publish the WSDL definitions of those services that conform to the static service type (`Printer`, in this example). ServiceJ passes these URLs to the WSIF middleware. It does so by creating a `WSIFServiceFactory`, which creates a `WSIFService` instance representing a *reified* version of each physical service endpoint (step 2 in Fig. 7). That `WSIFService` is then used to request a *stub* for contacting the service (step 3). All these stubs are returned to the ServiceJ middleware layer, where they are added to a *service pool* (step 4).

Steps 1–4 enforce that each remaining service conforms to the static type of `printer`, but there are no guarantees about whether they satisfy the constraints that were specified in the `where` clause of the `printer` field. Therefore, steps 5–7 further constrain the service pool. Each service is contacted in order to check whether it satisfies the requirements of the user. These service interactions must be executed using the protocol that the service provider speaks. To do that, ServiceJ leans on the WSIF middleware, which provides protocol-independent service invocations through the use of *dynamic service providers*. At runtime, the proper service protocol is installed, and each service is contacted using its own language. When the required information is retrieved, the ServiceJ middleware is able to constrain the service pool, retaining only those services that comply with the constraint as represented by the `where` operation. A similar interaction is needed to enforce the *preference* as represented by the `orderby` operation, but this is not shown in Figure 7. Once the service pool is constrained using these declarative operations, the remote service can be invoked in a robust way using the built-in failover mechanism.

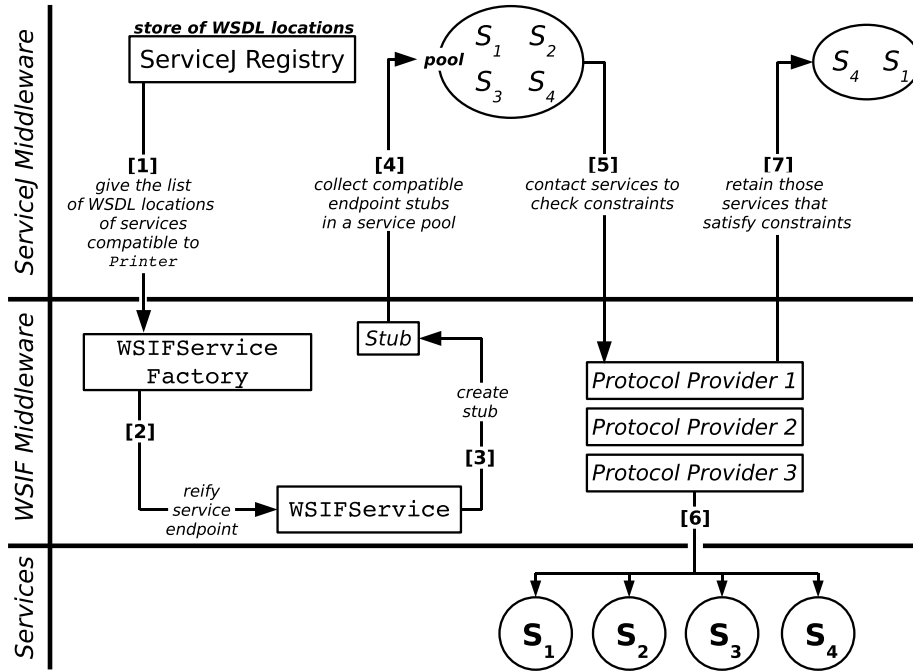


Fig. 7. The service selection procedure is transparently executed by the integrated middleware of ServiceJ and WSIF.

9 Related Work

Language constructs for handling connectivity problems in web computing were first introduced by Cardelli et al. in [10]. They use *service combinators* such as the *sequential execution operator* (denoted “S1?S2”) for stating that S2 must be contacted if S1 is unreachable. These Service combinators are used in object-oriented languages such as WebOz [11], WebL [12], and XL [13] to enable web service failover. Their dependence on hardwired service references, however, does not capture the volatility of web service architectures and it disables dynamic service discovery. Also, these languages lack support for defining constraints and selection policies.

JR [14] is a Java extension with specialized concepts for distributed programming. Similar to ServiceJ, JR programs are translated to Java before being compiled with the Java compiler. The concepts for JR are more geared towards language support for synchronization primitives in distributed computing and this makes JR less interesting for implementing the *business logic* of a service application. One major drawback of the language is that it expects each service provider to install a JR-specific virtual machine. These assumptions cannot be made in the context of cross-enterprise operations.

Jini [5] and JavaSpaces [15] are Java-based technologies aimed at implementing service-oriented applications. They use a public registry as a common place where providers register their services and where clients search for the service they need. Services can be registered along with a number of *entry objects*, where each entry object represents a characteristic of that particular service. A `PrinterService`, for example, may have an entry object denoting the location where that printer is found in a building. One drawback is that Jini forces the programmer to write a considerable amount of boilerplate code for retrieving services [16].

BPEL is a language for defining orchestrations of services, but this language lacks support for dynamic service binding. Service endpoints are hard-wired in executable processes, so network problems and service volatility may break client processes. This problem is partially solved in [17] by introducing *flow parametrization*. Parametrized flows allow to invoke operations on service references that were obtained as the result of a query, so they enable dynamic service binding. GPSL [18] is another language with similar concepts. These languages provide service selection, but they fail to support protocol independence because every process must communicate by exchanging WSDL documents. This makes it hard to integrate GPSL processes with services that don't understand the underlying protocol.

The GridRPC protocol, a communication protocol for scientific computational grids, is implemented using web service protocols in [19]. The main reason is that existing GridRPC-based systems rely on vendor-specific protocols, thus jeopardizing interoperability. Designing such a web services layer is a step in the good direction, since interoperability is improved. But other issues, such as service volatility and distribution problems are not tackled. One solution could be to incorporate a GridRPC-to-WSDL transformer into our ServiceJ compiler. That way, the applications would remain interoperable thanks to WSIF, and they would be more robust in a distributed setting thanks to ServiceJ.

The Java Commodity Grid Kit (CoG) [20, 21] is a framework for implementing Java applications for service grids. The framework offers classes for accessing specialized information and security services, as well as management services for configuring the grid. By abstracting these interactions in a framework, the programmer is able to approach these issues at a higher level. The main advantage of commodity kits like Java CoG is that they enable rapid development of sophisticated client-server applications. One drawback when compared with ServiceJ is that programmers need to insert a large amount of framework interactions in their source code. A lot of these API calls are also based on the Java Reflection API, e.g. passing class names as String objects, which gives programmers less static guarantees than our language extension.

ServiceJ extends the Java type system with type qualifiers. These qualifiers can be traced throughout the source code and thus allow the compiler to inject middleware interactions whenever a service is to be invoked. Similar extensions were proposed in [22], which adds type qualifiers for creating transparent proxies

for *future objects* and in [23], which adds qualifiers for supporting *reference immutability* in Java.

10 Conclusion and Future Work

ServiceJ introduces special language concepts that allow programmers to focus on the business logic, at the same time allowing them to fine-tune service selection in a declarative, type-safe way. We have shown how the ServiceJ compiler translates ServiceJ to Java, a process during which WSIF's protocol-independent service interactions and ServiceJ's infrastructure-specific service selection algorithms cooperate so as to dynamically bind the optimal service to each pool field occurring in a ServiceJ program.

There are two major future directions for the ServiceJ research project. First, we plan to extend its type system with *semantic information*. Currently, ServiceJ relies on *explicit* type conformance, so it cannot detect that a `PrinterService` and a `PrintingService` provide similar functionality if both types are not related in a type hierarchy. The injection of semantic information in the type system provides a solution to this problem. Second, we will define abstractions for *implicit service interactions*. Such interactions often occur in event driven service architectures, but currently, ServiceJ only integrates support for *explicit* client-service interactions.

References

1. C# Spec.: <http://www.ecma-international.org/>
2. Java EE 5.0 Spec.: <http://www.jcp.org/en/jsr/detail?id=244>
3. M. Duftter et al.: Web Service Invocation Framework <http://www.research.ibm.com/people/b/bth/OOWS2001>
4. SOAP Specification: <http://www.w3.org/TR/soap/>
5. Jini Architecture Specification: <http://www.jini.org>
6. Open Services Gateway Initiative: <http://www.osgi.org>
7. S. De Labey, M. van Dooren, and E. Steegmans: ServiceJ. A Java Extension for Programming Web Service Interactions. In: Proceedings of the 5th International Conference on Web Services, Salt Lake City, Utah (2007)
8. Igarashi, A., et al.: Featherweight Java: A Minimal Core Calculus for Java and GJ. In: OOPSLA99. Volume 34(10). (1999) 132-146
9. De Labey, S., van Dooren, M., and Steegmans, E.: ServiceJ. Service-Oriented Programming in Java. Technical Report CW451 K.U.Leuven <http://www.cs.kuleuven.be/publicaties/reports/CW/2006/>
10. Cardelli, L., Davies, R.: Service Combinators for Web Computing. In: Trans. on Softw. Engineering **25**(3) (1999)
11. Hadim, M., et al.: Service Combinators for Web Computing in Distributed Oz. In: Proc. Intl. Conference on Parallel and Distributed Processing Techniques and Appl. (2000)
12. Kistler, T., Marais, H.: WebL - A Programming Language for the Web. In: 7th Intl. Conference on the WWW. (1998)

13. Florescu, D., Gruenhagen, A., Kossmann, D.: XL: A Platform for Web Services. In: Proceedings of the First Biennial Conference on Innovative Data Systems Research. (2003)
14. Keen, A.W., Ge, T., Maris, J.T., Olsson, R.A.: Jr: Flexible distributed programming in an extended java. *ACM Trans. Program. Lang. Syst.* **26**(3) (2004) 578–608
15. Mamoud, Q.: JavaSpaces Technology: Beyond Conventional Distributed Programming Paradigms – www.javaspaces.org. (2005)
16. S. De Labey and E. Steegmans: Advanced Language Constructs for Developing Intra-organizational Service Architectures. In: Proceedings of the 2nd European Young Researchers Workshop on Service Oriented Computing (YRSOC'07). (2007)
17. D. Karastoyanova, et al.: An Approach to Parametrizing Web Service Flows. In: Proceedings of the 3rd International Conference on Service Oriented Computing. ICSOC05. (2005)
18. D. Cooney, et al.: Programming and Compiling Web Services with GPSL. In: Proceedings of the 3rd International Conference on Service Oriented Computing. ICSOC05. (2005)
19. Shirasuna, S., Nakada, H., Sekiguchi, S.: Evaluating web services based implementations of gridrpc. In: HPDC '02: Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing, Washington, DC, USA, IEEE Computer Society (2002) 237
20. von Laszewski, G., Gawor, J., Lane, P., Rehn, N., Russell, M., Jackson, K.: Features of the Java Commodity Grid Kit. *Concurrency and Computation: Practice and Experience* **14** (2002) 1045–1055
21. von Laszewski, G., Foster, I., Gawor, J., Smith, W., Tuecke, S.: CoG Kits: A Bridge between Commodity Distributed Computing and High-Performance Grids. In: ACM Java Grande 2000 Conference. (2000) 97–106
22. Pratikakis, P., Spacco, J., Hicks, M.: Transparent Proxies for Java Futures. In: OOPSLA04. (2004) 206–223
23. Tschantz, M.S., et al.: Javari: adding reference immutability to Java. In: OOPSLA '05: Proc. of the 20th Conf. on OOP, Systems, Languages, and Applications. (2005)