

Dispatch and Interaction in a Service-Oriented Programming Language

Tim Walsh, David Lievens, William Harrison

Computer Science Department
Trinity College Dublin
Dublin 2, Ireland
{Tim.Walsh;David.Lievens;Bill.Harrison}@cs.tcd.ie

Abstract. Much of the object-oriented software produced today is written in programming languages where the client requests services from specific objects. Important areas of software development such as aspect-oriented, pervasive and grid computing require a more flexible model on how aspects or services are found and used. In this position paper we describe an architecture that separates clients, dispatchers and multiple service providers.

1 Motivation

The most common model for object-oriented software is that in which a method is implemented in the object identified by a particular reference at the point of call. This object, called the "target" of the call, provides an implementation to service the message that it is sent. This common interpretation is breaking down in the modern computing environment, and we can see symptoms of this breakdown emerging in a wide spectrum of variations, including aspect-oriented software [7], pervasive computing [1], service-oriented software [2], and Grid computing [3]. These models all share at least one important difference from the common object oriented model - they interpose or presume a dispatching intelligence between the message-sending client and the message-receiving servicer. While the emergence of these new models clearly highlights the need for such a dispatch intermediary, we can also see how its introduction would help to simplify the ongoing attempts to allow the separation of business logic from contextual concerns.

2 Malleability

The ability to rapidly adapt to changing environments is important in all of the domains from section 1. There is a critical need to be able to use clients and services from one source in conjunction with those of others. Software in which clients are aware of the structure of the service providers is too brittle to be redeployed in this manner. Continual reorganisation is one of the motivators for

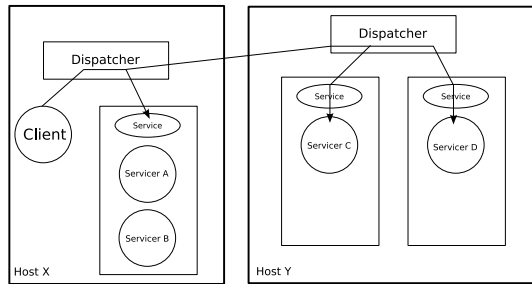


Fig. 1. Federated Service Dispatch

the growing popularity of both aspect-oriented technologies and service-oriented technologies.

A key aspect of the ability to write more malleable software, is reducing the dependency of a client on the way a service is implemented. In the current object-oriented model, a client and server share an interface. In Java, for example, the type checker enforces that a client is compiled with respect to the same interface a class is implementing. One step towards more flexibility, is allowing clients and servers to interact as long as they adhere to an interface that provides the required functionality even if the syntactic arrangement is different. To maximise flexibility, we envision the dispatcher being factored out of the language. In this way, dispatch may be contextualised by allowing declarative specifications of dispatching policies. At invocation time, any service-invocation may be connected to any aspect or service implementation that is deemed appropriate by the dispatcher.

3 Flexible Dispatch

Figure 1 illustrates the separation of client, dispatcher and services. The dispatcher must address both the selection of appropriate aspects of services *and* the co-ordination of those selected.

There may be many potentially suitable *servicers* and therefore, the dispatching problem becomes a matter of finding an appropriate candidate or candidates that will carry out the service in an appropriate manner. Malleable software must be capable of evolving to allow clients to make use of the most suitable services available at the time of use.

To address a client's request it can be necessary to use more than one service and the ordering of these service calls may be important. This is not a typical element of language specification but it is important here.

Conventionally, the interface guarantees all calls to the methods will succeed because the parameter variables refer to objects that implement the declared methods. The assertion has two elements:

1. the reference value provides assurance that the methods in its interface can be safely called, and

2. implementations of the methods in its interface are provided in the object referenced

The first is desirable, but the second is not. We plan to develop a language with the flexibility to allow assurances and implementations to come from any parameter supplied by the client and not necessarily the one that was expected when the service was written [4].

4 Modes of Service Interaction

As mentioned in the previous section we have deliberately introduced greater flexibility in the matching of services to clients. However, we now must reintroduce features that permeate the combination and use of the most appropriate services. We envision at least three ways of providing these capabilities: glossary references, protocols, and (dispatch) combination specifications.

4.1 Glossary References

The software developer needs a technique to discriminate between methods that appear equivalent in terms of only names and signatures. We propose to add attributes to methods that could be exploited to mechanically discriminate methods, and could act as semantic annotations for the use of programmers. For example, a method may be annotated with a tag indicating that a log is being updated on invocation. The tag may be exploited to rule out dispatching to this method if such behavior is undesired. This tag can be thought of as a link between developed software and a documentary specification whether formal or informal.

The presence of such tags can give rise to method description glossaries, or even ontologies that may then further be exploited to achieve some basic reasoning by the dispatching mechanism. For example, exploiting these tags may establish that the effect of invoking a certain service may also be achieved by invoking two or three other services in a particular order.

The glossary is also used to specify interaction characteristics of aspects or services such as whether they are mutually supportive or conflicting and can indicate dependencies among those available.

4.2 Protocols

Service-oriented software [2] requires orchestrating the participations of client and service activities so that the methods are invoked in the correct order and sequence. An example of such an orchestration languages is WSDL [5]. We feel it is important to provide such facilities in a manner that is integrated with the program language so that it becomes part of the mainstream software developer's way of thinking. As it is, these languages are treated as a separate specification used only for specialised environments such as web services. Better

description of the protocol for using provided services should form the basis for better engineering of software in general, rather than being seen as a something to be used only in specialised circumstances, like web services.

4.3 Combination Specifications

Aspect-oriented software [7] involves combining separately developed aspects or concerns by weaving the separate specification of behavior at points where the concerns cut across each other. Various aspect-oriented languages such as AspectJ [8] and HyperJ [6], specify the combination and ordering of logic from different concerns using terms like *before*, *after* and *around*, or by providing more complex ordering specifications. The pervasive [1] computing community also require the use of specification of complex dispatch. These specifications are intended to facilitate the inclusion of context information to find the appropriate implementation of services. In separating the dispatcher from the programming language we need to provide for dispatch specification that addresses the needs of both of these communities.

5 Summary

We are developing a language to address common needs of several domains. This language separates clients, dispatcher and multiple service providers. Such a system requires the ability to specify protocols for use of services, better semantic specification and their interaction, and specification of dispatch strategies, like prioritisation, for combining services.

References

1. Ubiquitous Computing overview available at http://en.wikipedia.org/wiki/Ubiquitous_computing.
2. Service Oriented Architecture overview available at http://en.wikipedia.org/wiki/Service-oriented_architecture.
3. Grid Computing overview available at http://en.wikipedia.org/wiki/Grid_computing.
4. Assurances vs. Capabilities as a Basis for Dispatch. Presentation at Foundations of Aspect-Oriented Languages workshop 06. slides available from <http://www.cs.iastate.edu/~leavens/FOAL/slides-2006/Harrison.pdf>.
5. Web Services Description Language (WSDL). Core Language. Available from <http://www.w3.org/TR/2006/CR-wsd20-20060327>.
6. HyperJ. Specification available from <http://www.alphaworks.ibm.com/tech/hyperj>.
7. Bader A, Elrad T, Filman R. E. Aspect-oriented programming: Introduction,. *Communications of the ACM*, 44(10), 2001.
8. Colyer et al. *Eclipse AspectJ*. Addison-Wesley, 2004.