Assimilating Service-finding into Object-Oriented Languages

Dominik Dahlem, William Harrison

Department of Computer Science
Trinity College
Dublin 2, Ireland*
(+353) 1-896 8556
{Dominik.Dahlem | Bill.Harrison}@cs.tcd.ie

* This research is supported by a grant from the Science Foundation Ireland.

© Dominik Dahlem, William Harrison, 2007. Permission to make digital or hard copies of all or part of this work for personal use is granted provided that that copies bear this notice and the full citation on the first page.
Assimilating Service-finding into Object-Oriented Languages

Dominik Dahlem and Bill Harrison
Software Structure Group,
Computer Science Department,
Trinity College Dublin,
Dublin, Ireland
{dominik.dahlem,bill.harrison}@cs.tcd.ie

Abstract

As software has increased in volume and complexity, software developers have increasingly exploited practices, tools support, and language features to modularise software and to loosen the coupling between the modules. But it is still generally true that developers assembling components into larger bodies of software use a great deal of latent information in choosing the components. Manifest specification of more information about interfaces than is needed for compilers to properly generate code, is often seen as useful for “secondary” concerns, like verification. In contrast, the popularity of service-oriented components places emphasis on using dynamic service-finding facilities, often called “service brokers”. The involvement of brokers to provide a service having certain characteristics implies that no assumptions can be made about latent information. The broker may supply any one of the services that provide the characteristics being sought. Component specifications for use in such a context must make more information manifest, information like the rules that govern use of its elements. The process of making more information manifest not only benefits coarse-grained business-to-business service interaction, but also benefits more flexible and dynamic use of components for solving in-house software needs, as might be used in the construction of software lines. This paper investigates a novel language construct to provide the effect of a service broker, applying the “new” operator to interfaces as well as classes. This allows conventional object-oriented software to derive the advantages that accrue to more flexible component selection being adopted for service-oriented software.

1 Introduction

Software designs are often faced with structural complexity. In most cases, this is due to the fact that several generations of software engineering, growth in the availability of alternative choices, independent development, often with different techniques, went into the construction of such a system. Selection of implementation modules is made based on the set of modules available at the time a client is written, where the reality is that, over time, greater choice becomes available (e.g. Hashtable to HashMap). And over time, collaborating modules will manifest a growing number of apparent dependencies, and will actually have a growing number of latent ones. Ongoing re-evaluation and devotion to use better alternatives is rare, and if the developers of these modules are responsible for resolving their dependencies an ongoing maintenance challenge is posed to the development team. In response, software developers have increasingly exploited practices [2, 8, 9, 20], tool support, and language features [11, 13] to modularise software and loosen the coupling between the modules.

The successful development of large and complex software systems in a timely fashion is mostly based on the pillars of modular programming, decomposability, composability, and adaptability [13, 17]. The criteria used to divide a system into smaller manageable parts and enable isolated and parallel development range from behavioural requirements to complexity in design. Assembling the individual parts into a larger body of work allows developers to balance flexibility against comprehensibility. A software system with few or no intra- or inter-module dependencies offers a higher degree of comprehensibility, and highly modular and hierarchical systems offer the flexibility to adapt to different environments and requirements easily with the additional advantage of reusable parts. The impact of reuse on productivity, quality, and defect density is a major motivation to further advance the development process in supporting loosely-coupled system design and engineering [1, 3, 7].
However, it is generally true that developers assembling components into larger bodies of software use a great deal of latent information in choosing the components. Developers take the risk of over-specifying the dependence on a component by declaring more methods than are used or needed in the interface they bind to and thus impede on best practices to avoid highly coupled software structures.

In a similar movement to those software development efforts to reduce coupling in software applications, enterprise software development experienced a shift towards service-oriented computing paradigms driven by the motivation to provide more flexible, robust, and resilient software systems [10, 16, 18]. Typically, in a service-oriented computing environment the interface of a component is structured into a number of sub-interfaces corresponding to specific services. Formally, services are usually identified with behavioural building blocks, i.e., define partial behaviours, while components define total behaviour [4]. This approach of dividing total behaviours into individual services can be considered a partial view on some component and thereby reducing the risk of over-specifying component interaction. In addition to the over-specification of component interaction, the realm of service-oriented computing places places an emphasis on dynamic service-finding component and thereby reducing the risk of over-specifying component interaction. In addition to the over-specification of component interaction, the realm of service-oriented computing places places an emphasis on dynamic service-finding facilities, often called “service brokers”. Those service brokers are “outsourced” entities from the language runtime and act on behalf of the service consumer to find a service that conforms to certain characteristics, such as functional requirements, provided by the consumer. Consequently, no assumptions can be made on latent information, such as protocol or assumptions about objects that are passed into a service. Instead the consumer relies on the manifest information provided by the interface. Moreover, as a result of loosely-coupled service interaction, a mediating entity can be devised to gather statistical information about service interactions and optimise the choice of service-binding based on the past history and future predictions of quality of service parameters, such as response time, reliability, among others.

This way of thinking not only benefits coarse-grained business-to-business service interaction, but also benefits more flexible and dynamic use of components for in-house solutions, for example in constructing product lines. Our approach marries the advantages of loosely-coupled service interaction from the service-oriented domain with a language construct to provide the effect of a service broker by applying the “new” operator to interfaces as well as classes. We believe that by adopting more flexible component selection mechanisms and expressing them into programming-language terms will encourage developers to apply the philosophy of “software as a service” [19] with the direct benefit of more loosely-coupled software that exhibits key characteristics such as low rigidity, low fragility, and low immobility [15].

Thus, the immediate objectives of this paper are as follows. First we set out current approaches to late-binding in Section 2 and discuss design patterns that can be derived from it. Second we raise issues related to implementing our approach and present possible solutions in Section 3. Third we motivate the need to align service-finding and late-binding as a general-purpose software development artifact in Section 4. Finally we conclude with future work in supporting our approach, among others, into a new language called continuum in Section 5.

2 Current Approaches to Late-Binding

Modern-day computing systems driven by component-oriented software development methodology and network-based service deployment created the need to bind to services at runtime to maintain loosely-coupled dependencies across organisational boundaries. Concomitant to this trend agile software structures have been devised to support fundamental design philosophies, such as loose-coupling and high-cohesion. Service-oriented computing emerged as a promising paradigm to facilitate agile structures represented by the premise that the choice of the implementor of a service is left until runtime, allowing the potential pool of service providers to change over time and according to context. The implications of this shift to loose-coupling in the enterprise software domain are two-fold. On the one hand, details about the internal object hierarchies are hidden from the client software and thus do not interfere with the client’s ability to choose among different service providers. On the other hand, client software cannot be deeply tangled with vendor-specific application programming interfaces for service invocation. Much effort has been put into modularisation in order to insulate client code from service-provider-specific APIs [5, 6]. The Composite Pattern of Web Services Invocation (CPWSI) by Buhler et. al. is a good example of dealing with abstractions of invocation engines [5]. In the following section we discuss a similar path to Buhler et. al., but with an emphasis on service finding and binding service implementations to given abstractions. In this vein, both patterns, the Bridge and the Factory Method pattern are applicable with some modifications whereby the Bridge pattern is collapsed into a simple interface/implementation class relationship.

The advantages of a service-finding approach can be used to address the problem of better modularising component software, and to accrue more flexibility and better behavior as software evolves. Not only does this flexibility incur a huge advantage for service providers who rely on third party services, but it also adds benefits to mainstream software development by moving the lessons learned through in software design into the language runtime.
2.1 Pattern for Service Finding

Design patterns have been used extensively to communicate elegant solutions to common software design problems in a formalised way and have influenced architects and application developers at the same time. The factory method pattern is of particular interest, because it offers a high degree of flexibility and independence both for the client software and the service provider.

Figure 1 depicts a client depending on a Service interface used in association with a ServiceFactory to create concrete implementations of the Service interface. Here ServiceFactoryA and ServiceFactoryB implement the interface ServiceFactory and provide implementations to create the respective services, ServiceProviderA and ServiceProviderB. Several implementation strategies for the concrete factories can be employed in order to align the creation of service implementations to the specific environment. For example, in a service-oriented environment a factory could contact a service broker, generate stub code, and bind to the endpoint accordingly. A different approach could be based on a local configuration whereby the fully-qualified interface name is mapped to the required fully-qualified implementation class name.

In general, the factory method pattern provides a transparent solution to instantiating, or if applicable finding a correct implementation of a given interface. The next section discusses the details on a new language construct to provide the effect of a service broker facility by applying the “new” operator to interfaces as well as classes.

3 A Language Construct for Late-Binding

To introduce the advantages of service-finding, we propose a Java language extension allowing the “new” operator specify an interface, instead of being restricted to classes. Generally, service-finding deals with providing service instances to a service consumer. A service broker relies on information made manifest to it in order to identify and resolve to the most appropriate service that is semantically and syntactically satisfactory. In the Service-Oriented Architecture (SOA) realm, manifest information mostly is associated with the interface specification, which is supplied by the service consuming client. In a Java Virtual Machine (JVM) the facility most analogous to service-finding is that of a class loader. The class loader is called by the Java runtime when one class references another one. More specifically, the “new” instruction identifies a symbolic reference to an interface, class, or an array type in the runtime constant pool. The responsibility of the class loader then is to resolve this symbolic reference and load the class into memory. The class has to pass all verification rules, including the integrity of the class file and correct inheritance hierarchy [12, 14].

However, the standard Java runtimes do not implement the feature of dynamic service-finding. This would allow a reference to an interface, which is less tightly bound than a class reference. Effectively, the runtime would provide an object implementing this interface, deferring the decision to make use of available resources. This feature can be seen as an extension to the class loader, embedding additional logic to resolve cross-references from an interface to one or more classes. However, the class resolution process becomes more complex if parameters to interfaces can be specified. These impose additional constraints on a suitable class by requiring it to implement a constructor that accepts the given parameters. There are several approaches to accommodating additional search criteria.
The most flexible solution is to use an implicit approach, taking the constructor specification from the actual parameters supplied to the “new” operator. This gives the class loader the greatest flexibility in exploring which implementation classes are suitable. This flexibility comes with the risk that the client code may not be what was intended; there is not the kind of redundancy usually provided by a declaration that can be checked against the use at compile time.

To gain error checking at the cost of flexibility, strong-typing can be provided by extending interface specifications to include the constructors that a class must implement. In contrast to the first solution, this is a more intrusive change to the language, because the Java virtual machine specification does not allow constructors to be declared in interface specifications. With even greater language impact, initialisation could be separated from allocation, treating the initialisation of objects as conventional methods, and allowing methods to be bound by a protocol using setter methods that accept those parameters. This protocol would form part of an interface’s specification. Without having a protocol specification, using conventional methods for initialisation has the undesirable side-effect of increasing the amount of latent information involved in the service call.

4 Implementing Late-Binding of Class Choice

SOAs have emphasised the availability of a service-finding capability to satisfy several needs. Service-finding allows services to be found by the infrastructure when needed at run-time rather than being found by the developer at the time the client code is being written. It simultaneously provides for a better separation of concerns and a better modular structure, for the use of newer or better services when they are made available, and for the selection of services based on context such as load, availability, or performance. To gain similar advantages for a wider class of applications, service-finding in general software development is exploited. This can be done by providing a language construct that supports late-binding of the classes used to implement objects. In the previous section we discussed an approach based on interfaces. This section explores the issues in supporting that language extension in the JVM.

By analogy to the service-brokers used in service-oriented computing, we envision a ServiceFinder as an external entity to the Java runtime that can be configured with any standard JVM. The ServiceFinder is a specialised class loader that selects one class implementing an interface, e.g., `java.util.Map`, from the several possible classes present on the classpath, e.g., `java.util.Hashtable` and `java.util.HashMap`. It passes this class to the default `defineClass()` method of the parent class loader, which constructs the appropriate runtime representation of a given class. To manage the set of classes from which to select, all class resolution requests are inspected by the ServiceFinder. If a class is identified to implement an interface, the information can be recorded in a JVM cross-reference repository. This repository is a persistent store that can be configured for each Java application and therefore isolates the cross-reference tables from un-related applications. It can be initially established by examining information available on the classpath or other sources at JVM start-up. If no class can be matched to an interface lookup request a `ClassNotFoundException` will be thrown indicating a missing class to the application. There are several options to service-lookup to support late-binding of class choice:

- **Initial generation of cross-reference information.** Static cross-reference tables are the simplest option for an interface-to-implementation class mapping. However, this requires the mapping to be established when the JVM is started. Moreover, this approach does not take into account changes to the classpath contents at runtime of an application.

- **Advance generation of cross-reference information.** As the size of a classpath increases, the lookup procedure becomes more resource intensive. In order to rectify this situation, a tool can be provided that inspects JAR files and records interface/implementation class mappings into the JAR manifest file. The ServiceFinder uses this information from the manifest files in order to reduce the overhead of exhaustively searching all classes in the jars on the classpath.

- **Dynamic generation of cross-reference information.** A more flexible approach to reducing the initialisation overhead is to generate the cross-reference information when the JVM loads classes, leveraging the class-loading mechanism provided by the Java platform. The ServiceFinder is configured with the JVM on the command-line forming part of the class loading hierarchy of the Java runtime. When the JVM starts up, the customised class loader passes through the classpath to establish all possible interface-to-implementation class mappings. Upon receiving an interface resolution request the ServiceFinder looks up its cross-reference table whether it can satisfy the request by loading the appropriate implementation class. The look-up process can greedily select among the most recently used or the loaded classes first, before inspecting the look-up table and thereby reducing the initialisation performance.

```java
java.util.HashMap
java.util.Map
```
However, the first match may not necessarily be the best fit. Consequently, implementations of a ServiceFinder have to balance exploitation versus exploration. Different software applications also may have different requirements on the service-finding facility. Therefore, our approach allows maximum flexibility and allows the extended class-loading mechanism to be configured to suite the needs of the specific application.

- **Use of a repository of dynamically updated cross-reference information.** In order to take advantage of the cross-reference information gathered in one JVM session, the look-up tables can be made persistent for future sessions and thereby reduces the initialisation costs significantly. But to avoid stale information in the look-up tables, the ServiceFinder masks failures to resolve implementation classes until the reference list is exhausted and removes those that do not resolve successfully. Additionally, the repository can be used to store statistics of past interactions with the services. As a result, the class-choice can be improved by selecting those that have been considered as being most successful.

5 Conclusion and Future Works

We have presented an issue, proposed a solution, and discussed implementation alternatives for introducing the advantages of late-binding of class-choice specifying “new” on interfaces as well as classes.

The advantage of using class loaders to put late-binding of class choice into the language is to reduce the dependence on a particular implementation class in favour of a dynamic, loosely-coupled dependency on an interface. The software developer delegates the responsibility of resolving to a particular implementation class to the Java runtime and thereby is able to exploit newly created or locally preferable implementations that satisfy the requirements. The creation of objects by specifying their interface eliminates the possibility that the developer will use latent information in choosing and using a particular implementation, thereby improving coupling metrics as well. In the face of an ever more dynamic and evolvable application software base, our approach is able to discover and exploit new additions to the classpath to consider those for potential interface/implementation class mappings.

The solutions are compatible with the class-file specification and format which permits interfaces to be specified as arguments to “new”. And modifying a typical Java compiler to accept the new language construct is a small matter.

Putting our work into the context of the question on “how dark should a component blackbox be?”, we believe the pursuit of the language construct we introduced in this paper would enable developers to treat their dependencies as abstract constructs, allowing the Java runtime to reason about the best match. But although today’s developer relies on much latent information, service-finding ultimately requires this information to become manifest information, exposing the dependencies. Expressing dependencies purely in terms of the methods required by a client class, the component can be considered quite black, because the software developer does in fact not know which implementation class will be providing the service to the client. In standard Java, manifest information is embodied by the interface specification. Java made a deliberate choice to use “nominal” interfaces - allowing much latent information to be embedded in the uniqueness of an interface name. Two interfaces with different names are considered to be different precisely to prevent violation of the latent contract. But, although Java interfaces are intended to encapsulate many latent specifications, they do not mandate an interaction protocol. In today’s software, unspecified latent information may therefore sneak into the selection of an implementation chosen for an interface. The software developer is using additional knowledge over what is explicitly stated. This presents a mismatch between the service implementor and the service consumer and consequently the component blackbox cannot be considered black. Restoring the blackness will ultimately require inclusion of more complete specification of behaviour and protocol, as is done in service-oriented implementation architectures.

The focus of future work aims at balancing and continuing to explore trade-offs in implementing a service-finding class loader extension for Java. Also, we are looking into an autonomous and adaptive service selection mechanism that takes into account statistics about past interactions in order to improve future class or service selections. The work presented here is the precursor to a new language, called Continuum, that will include protocol specification into the service interface, which will allow reduce the risk of relying on latent information in service interaction.

References


