Using Fail-stop Proxies for Enhancing Services Isolation in the OSGi Service Platform

Kiev Gama
University of Grenoble
LIG laboratory, ADELE team
France
Kiev.Gama@imag.fr

Walter Rudametkin
University of Grenoble
LIG laboratory, ADELE team
Bull SAS
France
Walter.Rudametkin@imag.fr

Didier Donsez
University of Grenoble
LIG laboratory, ADELE team
France
Didier.Donsez@imag.fr

ABSTRACT

The OSGi Service Platform is becoming the de facto middleware for deploying modularized Java applications. It is a dynamic platform that relies on a service oriented approach for loose coupling, but the absence of separate object spaces for isolating services of different modules cannot guarantee that service providers from uninstalled modules will no longer be referenced by active code. This may lead to memory retention and inconsistencies (e.g. a stale service that provides invalid cached data) that can introduce silent faults in the system by propagating invalid information.

We present our ongoing work where we introduce an isolation layer between service consumer and provider by using dynamic proxies for services. When the corresponding service becomes unregistered (i.e. uninstalled) the mechanism is able to: 1) Guarantee that no consumers directly refer to the service provider; 2) allow finding out the misreferencing consumer code by using a fail-stop mechanism. We have tested this mechanism in different OSGi based applications and benchmarked it against other approaches for accessing services in the OSGi platform.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Diagnostics*; D.2.8 [**Software Engineering**]: Metrics—*performance measures*

General Terms

Experimentation, Measurement

Keywords

Fail-stop services, OSGi, stale references, service isolation

1. INTRODUCTION

The OSGi Service Platform [14] plays an important role as universal middleware for modularized Java applications.

Its growing adoption in the sofware industry [10] [3] shows that it is becoming the *de facto* standard for such types of applications. The service oriented approach used in the OSGi platform enables a loosely coupled environment for the deploying of modules (called bundles in OSGi terminology) that may provide or consume services in OSGi. Bundles may be installed, stopped, updated or uninstalled during application execution without needing to restart the application. Services provided by such bundles may arrive and depart anytime. However, it is under the responsibility of the bundle developer to correctly handle the departure of services, which is notified by the framework. Since modules are in the same address space, there is no isolated container (e.g. separate object space) which can automatically handle the dereferencing of objects when a module is uninstalled. Thus we can say that OSGi has limited isolation mechanisms that are unable to enforce any sort of cross-module object reference rupture upon bundle departure.

The OSGi specification refers to a problem called Stale References, when objects of active bundles keep referencing objects from bundles that should have been unavailable. This problem leads to memory retention, since the deactivated bundle cannot be garbage collected, and can silently propagate inconsistent information due to the utilization of objects that should have not been used anymore. A tool [7] for the dynamic detection of stale references in OSGi based software proves that applications may actually present such misreferencing of objects upon the departure of bundles.

In this paper we present our ongoing work where we propose an approach to address service isolation issues in the OSGi platform. Our mechanism consists in introducing dynamically generated service proxies that will work as a layer of isolation between service consumers and service providers. Upon service unregistration, the proxy releases the reference to the actual service provider while the service consumer, even if misprogrammed, will reference only the proxy provided by the OSGi framework that is customized to carry our mechanism. Our proxies act as fail-stop services which in case of unregistration of the underlying service, they will intentionally throw an exception upon any method call, allowing us to identify the misusing of services. We have used Aspect Oriented Programming [13] to incorporate this approach into the OSGi framework and for achieving a portable solution across different OSGi implementations. This mechanism was tested in large OSGi-based applications and had its overhead benchmarked against other service utilization approaches. One interesting point that was also revealed

with our proxy based mechanism is that although developed towards a Service Oriented paradigm in the OSGi platform, some applications which we have tested do not exactly follow that principle every time, as described further in the text.

The remainder of this paper is organized as follows: Sect. 2 gives an overview of the OSGi platform and the problem concerning stale references; Sect. 3 presents our strategies and implementation of the proxied services and the fail-stop mechanism; Sect. 4 details the implementation and validation of the proposed mechanisms; related work is presented in Sect. 5 and, finally, Sect. 6 provides our conclusions and paths for future work.

2. THE OSGI SERVICE PLATFORM

The OSGi (formerly known as Open Services Gateway Initiative) service platform addresses modularization issues that are not natively[1] supported by the Java platform. OSGi is pure Java code that adds a middleware layer on top of the Java platform to fill some gaps (e.g. modularity) and to provide additional capabilities (e.g. information hiding at the package level in addition to standard modifiers in member and class level).

The OSGi framework is a dynamic service platform from the ground up where modules can be deployed, updated or removed from the platform with no application reboot. The deployment unit in the OSGi platform is usually referred as *bundle* which is a jar file that may contain components and services, and has special metadata in its manifest file to describe the module.

2.1 A Service Oriented Platform

Bundles may optionally provide services by registering them in the OSGi's service registry. In the service oriented (SOA) triad there are the service provider, service requestor and service catalogue which in the OSGi framework take the form of a bundle that provides a service, a bundle that requests a service, and the OSGi service registry, respectively.

Services in OSGi are usually registered under one or more Java interfaces into a shared service registry which is accessible through the BundleContext object provided to each bundle, as previously described. The implementations are decoupled by the usage of services. In OSGi, bundles may preferably communicate by means of services to enforce strong decoupling. Basically, whenever a bundle needs a service, it will ask the `BundleContext` for a given interface (and possibly some filtering parameters). A positive match would return a `ServiceReference` object, which encapsulates the metadata of a service object. This `ServiceReference` must be used to retrieve the actual service object by calling the `getService` method in the `BundleContext`, which will provide the corresponding servant object (i.e. the actual service). Other mechanisms for encapsulating this task are available, but we omit them here for brevity sake.

Upon service registration, modification or unregistration (either explicitly or implicitly due to the bundle being stopped) the framework notifies the subscribers of `ServiceEvent`s via a `ServiceListener` interface. Service requester code must be aware that a service departure means that the corresponding servant object must not be used anymore. Any

---

usage of the unregistered object may lead to inconsistency. Valid instances of the desired service would have to result from a new query to the BundleContext for a valid ServiceReference.

2.2 Namespace based isolation

The platform provides each bundle with its own class loader instance, which plays an important role in code isolation and is responsible for loading the resources and classes defined by the bundle as well as resolving classes provided outside the bundle (i.e. imported classes). OSGi specific class loading policies are able to handle the package level visibility defined in manifest entries of the bundle jar file. Class loaders can resolve classes imported from other bundles and perform runtime verifications to know if a bundle is able to instantiate classes provided by other bundles. Whenever a bundle tries to reference a type, its class loader will enforce if the visibility rules are followed.

2.3 Stale References

The previous subsection shows that in OSGi there is some level of isolation by means of individual class loaders. However, it concerns code isolation by controlling the visibility of types. Such isolation does not ensure that when a bundle is uninstalled, its objects (e.g. a provided service instance) will no longer be referenced. Although the OSGi platform provides events indicating the departure of services and bundles, if the service consumer code does not release correctly the object reference, this implies a problem referred in the OSGi platform as stale references is detailed in [6].

A custom diagnostics tool [7] is able to show that upon the departure of bundles OSGi based applications can present stale references, which are not visible in standard memory leak detection tools. Since an object is being referenced, its class loader and all class types loaded by the bundle will hang in memory. However, memory leaks are not the only problem concerning stale references. Inconsistencies, such as the utilization of a stale service instance which provides old cached data instead of using a new service that has replaced that instance. In situations like that, the system can silently propagate such inconsistent information.

3. ISOLATION OF SERVICES

For minimizing the impact of stale references, we introduce dynamically generated proxies as an isolation layer between service provider and service consumer. Our implementation can be seen in two distinct levels: The first one addresses isolation issues and avoids the retention of services in memory, while the second one increments the dynamic proxies by adding a fail-stop mechanism which would help identifying the code that is using stale services. That is, the first mechanism addresses memory retention while the second addresses the identification of client code that is misprogrammed and could potentially propagate inconsistent information. The next subsections detail both mechanisms.

3.1 Dynamic Service Proxies

In our approach, we use the proxy pattern [8] for creating a protection layer which would isolate the service provider from the service provider. Whenever an instance of a service provider is requested, a proxy for that object is dynamically generated and returned to the requester. Service consumers would no longer have direct access to the service provider
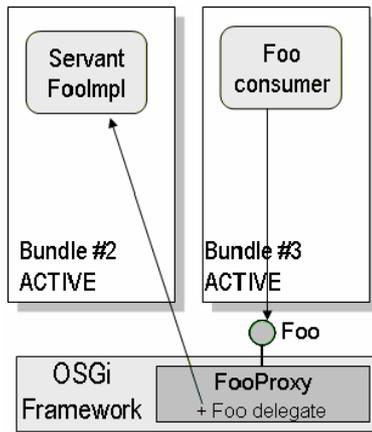
Figure 1: Dynamic proxy which delegates calls to acual service provider

instances. Any calls on the proxy object would be delegated to the actual service provider, as illustrated in Fig. 1.

This isolation layer does not intend to fix the misprogramming of service consumers but to prevent them of keep referencing unregistered service providers. Upon unregistration of the wrapped service provider, the proxy can release the reference that it keeps to that delegate so the service provider can be eligible for garbage collection (GC) if no other reachable objects refer to it.

It is also important to mention that this mechanism is limited to isolate objects which are used as services retrieved via the OSGi service registry. If a bundle that provides a service uses the service instance directly forwarding it to other objects, there is a potential risk of memory retention in case of code that does not handle the departure of services correctly. For example, a bundle may register in a JMX[2] server an object that points to a service provider that it registered in OSGi. Since JMX is not aware of the OSGi dynamics, if upon uninstallation the bundle does not remove the object it registered in the JMX server, the service instance will still be referenced.

3.1.1 Selective Mechanism

If need be, we can determine what service providers must not be proxied. This allows avoiding the overhead of proxies in some cases that are not necessary (e.g. services provided by the OSGi framework, component models). In our solution we have implemented a selective mechanism where we can use an exclude list which contains rules (e.g. implements, contains, subclassof) and class name patterns, defined in a file, to be matched against the service provider classes during the requests for service instances. By doing that it is possible to have more control and avoid the proxy overhead in the desired cases. This mechanism gives more flexibility to our dynamic service proxy mechanism and allows the necessary adjustments to enable some OSGi based applications to use the proxy approach, as described later.

3.2 Fail-stop services

A system that fails fast [17] should do so immediately

---

[2]Java Management Extentions provide tools for dynamically managing and monitoring applications
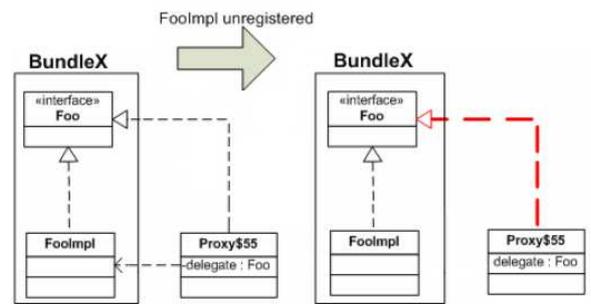

Figure 2: Object reference is released but reified type remains referenced by proxy

and visibly. By using such simplified failure model in OSGi to ensure that calls on unregistered services would fail, the portions of the application still using stale services would be explicit. If any calls to such services would result in a crash (an exception thrown) there would be no propagation of incorrect results, and bugs concerning the usage of stale services would be evident upon the crash.

We can consider the fact that the upon service unregistration the dynamic proxy will release the reference to the actual service provider, as in Fig. 2, which would supposely enable garbage collection. Thus subsequent calls to the proxy would lead to a `NullPointerException` when it tries to delegate the call to the service provider. However, the garbage collection is not guaranteed if other objects from active code directly reference the service object, like the JMX example given previously. We enforce service utilization validity in the proxy layer. Before delegating any call, the proxy verifies if the wrapped service provider has been unregistered. If so, the proxy fails by throwing a `RuntimeException` thus making the erroneous utilization evident.

4. IMPLEMENTATION AND VALIDATION

Developing the mechanisms described here is directly linked with the internals of any OSGi framework implementation. However we have used Aspect Oriented Programming to keep it as a separate concern which would enable easy portability across different OSGi implementations. We have successfully tested it in two implementations: Apache Felix [2] and Equinox [4]. A tool presented in [7] was used for verifying if service providers were garbage collected.

The proxies and the fail-stop mechanisms were initially validated in a scenario where we had developed a set of OSGi bundles intentionally misprogrammed. We have deployed them in an OSGi platform and observed the results of life cycle events on bundles that provided services used by our code. Upon the uninstallation or update of other bundles, our bundles retained object references to services that had been unregistered, thus presenting stale references. In the experiment with this set of bundles we proxied all the services and did not utilize the selective mechanism. After performing updates and uninstallation of bundles we could verify that the isolation through proxies allowed the garbage collection of the actual service providers (i.e. the delegates). We could also verify that the clients that were service consumers which were using stale services would fail, since they were getting an exception from the proxy. These results validate our concepts of isolation and fail-stop services.

Table 1: Garbage Collection of proxied stale services

		Newton	SIP Comm.
I	**Application**	Newton	SIP Comm.
II	**Version**	1.2.4	Alpha 3
III	**Lines of Code**	85000	125000
IV	**Total Stale Services**	76	21
V	**Proxied Stale Services**	35	9
VI	**Reclaimed Delegates**	28	5
VII	**GC percentage (VI/V)**	80%	56%

Table 2: Overhead compared to direct service calls

Method call type	Overhead
Fail-stop dynamic proxy	2.65
Static proxy	1.09
iPOJO	1.10
RMI	At least 200

However, these techniques needed to be validated against large applications. We have tested this mechanism in two OSGi-based open source applications: SIP Communicator, a multiprotocol instant messenger application; and Newton, a service component architecture. Tests have also been performed in JOnAS, a Java EE Application Server, but our mechanism presented incompatibilities with the component model used in several JOnAS bundles. The errors concerned type visibility, and needed further investigation.

In this test we used the selective mechanism to avoid proxying some services due to explicit typecasts on code that consumed them. A script performed the update of bundles, forcing dynamic events that implied in the departure and arrival of bundles and services, resulting in a set of stale services in both applications. Table 1 shows the garbage collection effectiveness of service providers whose proxy have become stale referenced. Line IV shows the total stale services, either proxied and non-proxied. Line V shows the total proxied stale services, while VI shows that not all proxied service has been garbage collected, that is, they are still reachable. The proxies avoid service consumers referencing service providers directly. For instance, a bundle B that provides a proxied service X may have another non-proxied service Y which has become a stale service during bundle uninstallation or update, but if Y references X directly, X will not be eligible for GC as long as Y is reachable.

In SIP Communicator the fail-stop mechanism allowed us to identify some functionality that was still being used. By analyzing the regular error output of the system, we could see which class and line of code was using the stale service. However, in order to get more results with this fail-stop mechanism we would need to use the tested applications thoroughly to verify if any utilization of the system presents errors due to calls on unregistered services.

Although the module updates that we have performed in the tested applications may not reflect their actual evolution, we can see that even applications constructed by experienced developers may present mishandling of events and service departure concerning OSGi, and may not be completely ready to fully take advantage of the platform's dynamicity.

4.1 Benchmark

We have chosen to perform a benchmark[3] for measuring the overhead of our dynamic proxy mechanism. It was compared with other approaches in the OSGi platform as well with Java Remote Method Invocation (RMI) for establishing communication with a JVM running in other process. The experiment measured the invocation time of a parameterless method in a service registered in the OSGi platform.

[3] OSGi Platform: Apache Felix 1.0.4. JVM: Sun Hotspot/JRE 1.6.0_07. OS: Windows XP SP2. Hardware: Pentium 1.7 GHz 1GB RAM

Optimizations, which are not under our control, made during compile time and execution time were apparently affecting our initial measurements. At each benchmark execution the overhead of the dynamic proxies was exponentially growing whenever we increased the amount of times the method should be called. As an alternative for avoiding such optimizations we have altered the benchmark code in a way that some sorts of optimizations (e.g. method inlining) could not be done. We used Java reflection to perform the method invocations that were being measured. As a result, the overhead remained aproximately the same if we varied the number of method calls performed in the benchmark.

This was a simple benchmark, based on the code from [16] where we did not make a deep statistical analysis. The deviation was not analyzed and we present only set of results that carried the minimum values of the benchmark, which have ocurred repeatedly in different executions. Our intention by measuring that is to have a rough idea of the cost introduced by the dynamic proxies and compare it with other alternatives. In Table 2 we compare the dynamically generated proxies with other approaches in OSGi: direct service calls; static proxies as services (i.e. indirect service calls) and calls to services injected by iPOJO [5], a component model that performs bytecode manipulation for handling, among other things, service dependencies.

By comparing the results of the benchmark we can see that our approach adds a method invocation time overhead that is fairly acceptable and provides a very low cost (less than 3 times of a direct service call) to achieve service isolation if compared to RMI (an alternative for isolating services in different JVMs). iPOJO has a low overhead slightly higher than static proxies, which are a good alternative to dynamic proxies and follow the same principle of the proxy design pattern. We have manually generated the static proxy as a class that implements the tested interface and delegates the calls to the target object, held as an attribute, which implements the same type. We wanted to position its performance so we could analyze the gains for future implementations of our mechanism. The static proxy mechanism has a better performance than all other approaches. Apparently, we can cut the overhead by more than half if we reimplement the dynamic fail-stop proxies as static proxies generated during runtime.

4.2 Drawbacks

Since this is an ongoing work, we are still evaluating the drawbacks imposed by the utilization of our approach. We have already identified a few issues which we detail here.

In [8] we can find an important principle of reuse in object-oriented design: *"program to an interface, not an implementation"*, which is a basic concept for decoupling in Service Oriented Computing. This principle flows naturally in SOC environments like XML Web Services, where typical scenarios are service consumer and provider located in different

```
//Service registered under the MyService interface
ServiceReference ref =
ctx.getServiceReference("xyz.MyService");
/* The code below "knows" the actual registered
/* MyService implementation, and does a typecast */
MyServiceImpl serv = (MyServi-
ceImpl)ctx.getService(ref);
/* The typecast in above line would not work with our
/* proxy approach, which would return a proxy for
/* MyServiceImpl instead of the actual object */
```

Figure 3: Typecasting that would break our code.

machines which communicate using well defined contracts (interfaces) but communication layers are hidden behind proxies that transparently provide the same contract. In that case, the client consumer does not have access to the actual object that provides the service. In OSGi based applications, bundles should communicate using such contracts as well but service consumer code may violate that principle by doing explicit typecasts on interfaces. Since the service consumer and provider execute in the same memory space, and there is no proxying or marshalling mechanism, programmers assume that they can perform typecastings of services to concrete types when they know what actual implementation is behind the service interface, which usually works.

The usage of our fail stop mechanism works only on applications that consume services as a contract defined by interfaces. This is what we expect in service-oriented applications. However, implementations that choose not to blindly rely on interfaces and try to use typecasts to concrete implementations, as in the example on Fig. 3, would compromise the functioning of our proxy approach. A workaround for that problem was to include such services in the exclude list of the selective mechanism explained in Sect. 3.

Other issue concerning synchronization may arrive if the bundle that provides the service uses any lock directly on the servant object. If consumer code tries to synchronize access to that service, they will not have exclusive access guaranteed since they are synchronizing on the proxy object. However if the bundle that provides the service accesses it via the OSGi framework, the synchronization point will be the proxy which is used by all consumers of that service.

As we have presented, our approach addresses: 1) memory retention (services and class loaders); and 2) the identification of unregistered services (stale services) still being used by service consumers. While the latter is easily addressed with the fail-stop approach which will accuse which service consumers are still using unregistered services, the first issue is partially solved. We can guarantee that service consumers will not reference the service providers, thus the service object can be reclaimed by the garbage collector but the class loader used by our dynamic proxy is the same of the service provider. Thus, as long as service consumers keep referencing the proxies the class loader will still be referenced even if the corresponding bundle has been uninstalled.

5. RELATED WORK

Another work [1], also based on proxied services, deals with fault tolerance concerning service availability in OSGi. However, that approach does not prevent the stale service from being called. Their proxy solution is responsible for dynamically locating the best service implementation, and in case of faults it tries to locate another service.

In [11] we can see a service failure approach which presents a fail-stop solution to handle faults in the composition of services in SOA environments where consumers of a service must anticipate that any service provider will fail (crash) from time to time.

Spring Dynamic Modules [18] provide a transparent proxy utilization in the OSGi platform which is similar to our approach, but we different objectives. They provide a dependency injection approach for objects from spring modules that would consume OSGi services. Instead of injecting the reference to the service, they inject a proxy to that service which would handle the departure and arrival of services and some rules for locating services that may depart, similar to the approach mentioned in [1]. While in Spring they act only on Spring specific modules, which need to use its API and specific metadata, our approach targets any requested service registered on the OSGi service platform.

The previously mentioned approaches use proxies for working in a dynamic service level agreement fashion. We also investigated other approaches which concern environments for enforcing isolation, like the JSR 121 [12] provides a specification for a Java environment where application isolation can be done by means of Isolates, which are application units which resemble lightweight processes. Applications are isolated in different object spaces but they can share some resources like runtime libraries. Communication between isolates can be done through Java RMI (Remote Method Invocation) based or equivalent mechanisms [15] which imply in marshalling objects across contexts. A hybrid mechanism described in [9] combines the JSR121 with an extensible virtual machine. Their solution presents a concept of JVM domains which allows ligthweight isolation between them.

The Microsoft .NET platform utilizes application domains, referred by [19] as "lightweight address spaces", which can isolate applications that run inside the same Common Language Runtime (CLR). A single CLR process can run several .NET applications by loading them in separate application domains. It is possible to have a multi-application environment without the overhead of process context switching. Application domains are isolated but they reside in the same CLR process space, communication across domains is possible via marshalling of objects using .NET Remoting, which is the inter-process communication approach in .NET.

6. CONCLUSIONS AND FUTURE WORK

The widespread adoption of the OSGi Service Platform as an environment for modularized applications is evident. Its service oriented approach allows loose coupling between bundles, but the dynamicity of that platform may lead to inconsistent service referring if the departure of modules are mishandled. As a consequence of that, combined with object isolation limitations on OSGi, applications may present memory leaks and propagate inconsistent results. We present a mechanism which minimizes and detects such problems by introducing an isolation layer between service consumer and service provider.

The implementation consists of dynamic proxies providing a fail-stop mechanism upon service unregistration. The approach was used to avoid the retention of unregistered services and to invalidate method calls on stale services, avoiding the propagation of incorrect results and facilitating the search for stale services being used in the application. Although some improvements need to be done to the mecha-

nism in order for it to run on any type of OSGi application, we have proven its feasibility and interest for applications.

The tests that have validated the fail-stop mechanism were initially performed in a controlled environment where the bundles were intentionally misprogrammed and presented stale references upon the departure of bundles. The mechanism worked as expected: service usage was not affected except when unregistered services were improperly called. The benchmarking showed that this approach is expensive, but we see it as a low-cost implementation to enhance isolation aspects in OSGi.

This mechanism was also tested in other OSGi based applications which allowed us to see that some applications targeting the OSGi platform do not always follow the "programming to interface" principle, which caused our approach to fail when bundles performed typecasts to concrete types forcing us to add such cases to our exclude list that determined the non-proxied services. We have analyzed the results of experiments in two OSGi based applications of significant size, which we have observed the garbage collection of unregistered services and by using the fail-stop mechanism we could identify portions of the code that kept using such unregistered services.

In a simple benchmark we analyzed the invocation cost of a dynamic proxy comparing it with RMI calls and also two other service usage approaches in the OSGi platform: static proxies (indirect service calls) and iPOJO. The dynamic proxies have a very acceptable cost if we compare it with with the isolation cost of RMI (i.e. isolation in different VMs). The utilization of a static proxy proved to have a better performance than all other approaches.

In the continuation of our work we will evaluate the impacts of dynamic proxies in the OSGi environment, as well as the issues that such mechanism may bring. We will replace the dynamic proxy implementation by static proxies. With the help of bytecode generation libraries the static proxy classes are going to be generated and dynamically loaded during runtime. The proxies of the new approach would work exactly as the dynamic proxies. Although there would be the initial cost of bytecode generation when registering the services, we would take advantage of faster execution with the static proxies during application lifetime. We also plan to execute tests in more realistic scenarios that actually reflect usage and evolution of OSGi based applications instead of using scripts for batch updates of modules.

7. REFERENCES

[1] H. Ahn, H. Oh, and C. O. Sung. Towards reliable OSGi framework and applications. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1456–1461, New York, NY, USA, 2006. ACM.

[2] Apache Felix. http://felix.apache.org.

[3] M. Desertot, D. Donsez, and P. Lalanda. A Dynamic Service-Oriented Implementation for Java EE Servers. In *SCC '06: Proceedings of the IEEE International Conference on Services Computing*, pages 159–166, Washington, DC, USA, 2006. IEEE Computer Society.

[4] Equinox. http://www.eclipse.org/equinox.

[5] C. Escoffier, R. S. Hall, and P. Lalanda. ipojo: an extensible service-oriented component framework. In *Services Computing, 2007. SCC 2007. IEEE International Conference on*, pages 474–481, 2007.

[6] K. Gama and D. Donsez. A Practical Approach for Finding Stale References in a Dynamic Service Platform. In *11th International Symposium on Component Based Software Engineering (CBSE 2008)*. Springer LNCS, 2008.

[7] K. Gama and D. Donsez. Service Coroner: A Diagnostic Tool for finding OSGi Stale References. In *Proceedings of the 34th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 2008.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison-Wesley, January 1995.

[9] N. Geoffray, G. Thomas, C. Clement, and B. Folliot. Towards a new Isolation Abstraction for OSGi. In *Proceedings of the First Workshop on Isolation and Integration in Embedded Systems (IIES 2008)*, pages 41–45, Glasgow, Scotland, UK, April 2008.

[10] O. Gruber, B. J. Hargrave, J. McAffer, P. Rapicault, and T. Watson. The eclipse 3.0 platform: adopting osgi technology. *IBM Syst. J.*, 44(2):289–299, 2005.

[11] C. Hobbs, H. Becha, and D. Amyot. Failure Semantics in a SOA Environment. *Montreal Conference on e-Technologies*, 0:116–121, 2008.

[12] Java Community Process. JSR 121: Application Isolation API Specification. http://jcp.org/en/jsr/detail?id=121.

[13] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[14] OSGi Alliance. OSGi Service Platform. http://www.osgi.org.

[15] K. Palacz, J. Vitek, G. Czajkowski, and L. Daynes. Incommunicado: efficient communication for isolates. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 262–274, Seattle, Washington, USA, 2002. ACM.

[16] L. Seinturier, N. Pessemier, C. Escoffier, and D. Donsez. Towards a Reference Model for Implementing the Fractal Specifications for Java and the .NET Platform. In *5th Fractal Workshop at ECOOP'06*, July 2006.

[17] J. Shore. Fail fast. *IEEE Software*, 21(5):21–25, 2004.

[18] Spring Dynamic Modules for OSGi Service Platforms. http://www.springframework.org/osgi.

[19] D. Stutz, T. Neward, and G. Shilling. *Shared Source CLI Essentials.* O'Reilly Media, March 2003.