Applying Dependability Aspects on Top of "Aspectized" Software Layers

Kiev Gama and Didier Donsez
LIG laboratory, University of Grenoble
Bat. C, 220 rue de la Chimie, Domaine Universitaire
Grenoble, France
{kiev.gama, didier.donsez}@imag.fr

ABSTRACT

Dynamic platforms where components can be loaded at runtime can introduce risks to applications stability if components are not previously known before deployment. It may be needed anyway to execute such untrustworthy code, even if it is not malicious. The OSGi platform presents such a scenario where components can be installed, started, stopped, updated or uninstalled during application execution. In this paper we describe how we introduced dependability and monitoring as crosscutting concerns in the OSGi platform for improving applications resistance to such risks. These concerns crosscut different software layers which are well defined in the OSGi specification but scattered over different places in the OSGi API. We also created a level of indirection by representing software layers as aspects, enhancing the API's modularity as well as reuse by avoiding redundant pointcut definitions. The dependability aspects helped us validating the layer aspect abstraction reuse. Since the aspects targeted the OSGi API, it was possible to weave our solution into distinct versions of three different OSGi implementations, namely Apache Felix, Equinox and Knopflerfish. We validate our approach on all of the woven platforms in a simulation of an RFID and sensor-based application that uses untrustworthy components.

Categories and Subject Descriptors
D.1.m [**Software**]: Programming techniques – *miscellaneous*

General Terms
Reliability, Experimentation

1. INTRODUCTION

The ability to dynamically load components during application execution in platforms such as Java and .NET introduces a lot of flexibility gained with late binding but at the same time, it may introduce potential risks for applications. Testing becomes difficult if the runtime deployed assemblies are not known in advance. Performing tests against all sets of possible components from which the application may be composed is a hard task that is not cost effective. Dynamically loaded code —even if it is not intentionally malicious— that is not known in advance may introduce risks to applications since the behavior of such code in the application has not been previously tested. Java and .NET platforms run managed and type safe code, having features such as bounds checking and garbage collection (preventing errors such as buffer overflows and memory leaks, respectively). It minimizes a range of errors, but applications and components are not free from naïve programming errors that under certain circumstances could lead to problems like excessive memory or CPU consumption. Also, the need to load native libraries into such managed environments opens breaches that can lead to application crashes in case of severe errors caused by the underlying native library. Centralized component-based frameworks do not provide isolation boundaries that ensure fault containment, rendering an application vulnerable to such threats, which are not necessarily malicious. Besides such boundaries, purging a misbehaving component and restarting it without stopping the application is an interesting feature to have.

The possibility of dynamically loading components at runtime can be found, for example, in DynamicTAO [25], targeting CORBA, as well as in SOFA/DCUP [34] and OSGi [31][30], for the Java platform. Our study focuses on OSGi, which is currently seen by industry as a *de facto* standard for constructing modular and dynamic Java applications. In OSGi, components can be dynamically installed, started, stopped, updated and uninstalled during application execution, but since all objects share the same memory space, a crash or malfunction in a component may affect the whole application. In previous work [14], we have developed a self-healing sandbox for hosting third-party OSGi components in order to enhance applications dependability. In our solution we allow the transparent deployment of components into an isolated sandbox, based on isolation policies dynamically checked against components (e.g., components provided by an unknown vendor should be placed in the sandbox). OSGi components communicate via service objects, and our sandbox mechanism still allows the components in the main application to communicate with isolated sandboxed components (and vice-versa) by using dynamic proxies that seamlessly utilize inter-process communication (IPC).

Our solution was initially coded as a set of patches to Apache Felix[1] version 1.4, which is an open source implementation of the OSGi specification. Attempts to port that solution to a more recent version of Apache Felix would require manual work of copying

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
AOSD'11, March 21–25, 2011, Pernambuco, Brazil.
Copyright 2011 ACM 978-1-4503-0605-8/11/03...$10.00.

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

and pasting the patches that are scattered across different classes. Migrating to another OSGi implementation (e.g., Eclipse Equinox[2], Knopflerfish[3]) requires a deep analysis of the target implementation source code and migration of the patches. To ease the burden of applying such patches manually, we have extracted and refactored them into aspects, which was a good choice for modularizing the dependability crosscutting concerns. This refactoring approach enables better code evolution and can also be found in other domains such as software product lines [2]. In our case, it allows us to easily apply the extracted crosscutting concerns over two dimensions: 1) across different versions of a given OSGi implementation, and 2) across different OSGi implementations (i.e., different vendors), thus enhancing the maintainability of our solution and its applicability.

We have identified the points of interest of the OSGi API where our dependability aspects should be applied. The API is standardized and the common point to all implementations, therefore the aspects targeting the API are applicable to any of the implementations. However, during our work we have noted that useful concepts described in the OSGi specification are not well represented in its API, making it difficult to distinguish abstract concepts in the specification from their counterparts in the API. For instance, the software layers specified by OSGi are scattered over different interfaces, which accumulate roles from different layers. There are no single entities to describe individual layers neither there is a single access point for accessing the services of each layer. Software layers are abstractions to enhance modular design. Therefore, if such layer concept is lost when a specification is translated into an API, we lose modularity as well.

Providing reusable abstractions for these concepts improves modularity and allows better comprehension of the API from an architectural point of view. We have analyzed the OSGi API and used aspects to reify these abstract software layers, distributing the resulting code in the form of an aspects library. Layers can be crosscut by different concerns which are aspects of more specific purpose (e.g. logging, dependability). In this case, instead of applying the specific aspects directly to the OSGi API, we add another level of indirection through layers that are "aspectized". The specific aspects can reuse the pointcut definitions of these layer aspects, giving us two advantages: better readability with a clear understanding of which layers are crosscut by which aspects; and reuse of pointcut definitions, which need be define only once in the layer aspect thus avoiding redundancy. We demonstrate such reuse by refactoring our OSGi dependability patches as aspects that reuse these new layer abstractions. Although we concentrate on dependability aspects this approach could use the same strategy for introducing any aspect addressing the OSGi framework by means of the layer aspects.

To the best of our knowledge no previous studies have refactored software layers as aspects or addressed dependability as aspects in the context of dynamic component platforms. The main contribution of this paper lies on the usage of aspects for disentangling software layers from an API, representing them as aspects and allowing their pointcuts to be reused by other aspects that provide more specific crosscutting concerns, like logging or dependability, improving modularity for better abstractions and reuse. A secondary contribution of this paper is the usage of aspects for enhancing dependability in the OSGi platform. This is demonstrated by providing dependability aspects constructed on top of this newly introduced abstraction of layers. Although the solution is specific to the OSGi platform, our approach is useful in any application framework that has similar conceptual gaps between specification and implementation.

The remainder of the paper is organized as follows: section 2 gives background and motivations, followed by the abstractions of OSGi layers in section 3. Implementations of the dependability aspects on top of the layer aspects are explained in section 4 and the validation described and discussed in section 5. Section 6 comments on related work followed by conclusions and perspectives in section 7.

2. BACKGROUND AND MOTIVATIONS

The next subsections provide a brief overview on the OSGi Service platform and the issues, especially those resulted from component dynamism, that motivate us to introduce dependability cross-cutting concerns in that platform.

2.1 OSGi

The OSGi Service Platform targets the construction of dynamic and modular Java applications, allowing strong decoupling between components in a service-oriented fashion. The OSGi specification defines a framework that allows the dynamic deployment and undeployment of components and services. It leverages Java's dynamic class loading feature for enhancing modularization and also introduces a runtime where software components can be installed, started, stopped, updated or uninstalled without halting the application.

The deployment unit in OSGi is called *bundle*, which is an ordinary compressed jar file containing classes, resources and an extended manifest file. This manifest contains OSGi specific attributes providing metadata about the bundle dependencies (e.g., a list of imported and exported class packages). A bundle can be dynamically loaded or unloaded on the OSGi framework and can optionally provide or consume services, which are ordinary Java objects. Services need to be registered in the OSGi service registry as providers of the specified interfaces. Service-oriented principles provide strong decoupling between components in OSGi. As described in [33], in a basic Service-Oriented Architecture (SOA) there are three types of participants: service provider, service discovery agency and the service requestor (i.e. a client). Their interactions involve publish, find and bind operations. In the case of the OSGi framework, those participants take the form of a bundle that provides a service, the OSGi service registry and a bundle that requests a service, respectively. The interactions are centered on the service registry which notifies interested parties about service publications or withdrawals.

The dynamic composition mechanisms rely on a service-oriented composition approach. Different service-based component models have been constructed on top of the OSGi service registry helping manage the complexity and minimize the burden of service registration and unregistration that govern the service dependencies and bindings. However, such models are not enough for guaranteeing the mishandling of references. Stale references [12] are a typical problem in OSGi applications when the

[2] Eclipse Equinox. http://eclipse.org/equinox/

[3] Knopflerfish OSGi. http://www.knopflerfish.org

dynamicity of bundle substitution or uninstallation is mishandled. It is characterized by references to services that should no longer be available (i.e., they are unregistered), and generally by objects provided by the classloader of a bundle that has been stopped or uninstalled still being referenced. The usage of unregistered services may introduce silent faults in the application (e.g., inconsistent cached data from a stale service) that are hard to find and may propagate throughout the system.

The functionality provided by the OSGi framework is divided in different layers as illustrated by the gray boxes in Figure 1, which is based on a picture from the OSGi specification [32]. The module layer provides rules for sharing packages between bundles; the lifecycle layer provides a runtime model for bundles; the service layer specifies the programming model that ensures loose decoupling between bundles; and the bundles layer are the actual OSGi components to be deployed on the framework. The security layer is based on Java mechanisms with some extensions, but it is an optional layer in OSGi.

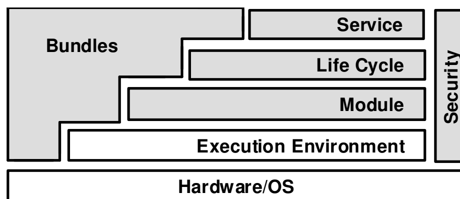

Figure 1. OSGi layers detailed in gray.

2.2 Issues with Third-Party Code Execution

The dynamic replacement of bundles in OSGi is not always effective since the correct removal of a bundle (e.g., uninstall) requires the dereferencing of its objects and types from other bundles, which is not ensured by the framework but is based on good programming practices. For instance, if the framework is told to update a bundle, it will roughly: send a stop notification to BundleListeners, stop the bundle's execution (i.e., call its BundleActivator stop method), reload the bundle, send a start notification to BundleListeners, and then start the new bundle version. There is no command or method for actually purging a component from memory. In order to clear a bundle's set of objects, classes and its classloader from memory, good practices call for the following: the stopping bundle must terminate its spawned threads, it must remove objects from registries external to OSGi (e.g., removing a JMX[4] MBean from an MBeanserver), consumers of the bundle's services must release references to the servant objects from that bundle. Such component removal limitations occur, in part, due to the fact that the only isolation provided to objects is that of namespace, which is done using individual per-bundle classloaders.

This limitation in isolation also has consequences when a component crashes the application (e.g., due to an unstable native driver). In fact, these risks are common to most centralized component-based applications where all objects share the same memory space. Problems may include mishandling dynamicity (e.g., stale references), excessive resource consumption, among others reasons that may halt or significantly degrade an application. Importing or wrapping native libraries (e.g., a device driver) also increases the risk of an application crash, since they are unmanaged code.

[4] Java Management Extensions - http://java.sun.com/jmx

3. MAPPING LAYERS TO ASPECTS

Layers [3] are a widely used architectural pattern for grouping different levels of abstraction in a system. In layered architectures, it is a good practice to design a *flat interface* that offers all services from a given layer. If we consider a purist layer design, a layer of a system should only communicate with its adjacent layers, via such flat interfaces. A relaxed layered system, also mentioned in [3], is less restrictive in the sense that a layer may directly use all layers below it, which is the case in the OSGi platform where the bundles layer freely accesses the other three layers, as illustrated in Figure 1. But in practice such access in OSGi is not done through a single interface per layer. Actually, there is no such flat interface for explicitly representing layers in OSGi's API. The functionality of each layer is scattered over different interfaces which may accumulate roles from other layers.

To illustrate this, we analyze how the bundle layer accesses the other layers. The BundleContext interface has responsibilities in the service and lifecycle layers. The OSGi API centralizes operations in the BundleContext, where we find different layers entangled and several non-related responsibilities. The BundleContext is an interface that works as a sort of Façade that exposes varied framework functionality to a bundle. Through its BundleContext instance, a bundle directly accesses operations of the service layer and part of the lifecycle layer. A bundle is represented by an instance of the Bundle interface, which provides lifecycle transitions (not all of them) and gives access to other two layers: service and module layers. An important principle described in [3] says that layers should be separated from each other, having no component spread over more than one layer. However, in OSGi a bundle has different points of access to each layer, and each point of access does not work in a per-layer basis since they are entangled with code from different layers.

We use aspects to create a flat interface vision for each layer, making explicit a sort of central weaving point of access to the services of a given layer. These layering abstractions are fundamental for adding crosscutting concerns in a more structured way, providing a clear architectural vision of the layers affected by a crosscutting concern that reuses such abstractions. Another advantage of this approach is that the pointcuts that define the layers can be reused. For example, if two different aspects need to intercept lifecycle transitions the pointcut definitions need not be repeated. If a developer needs to think in terms of OSGi layers for applying aspects (e.g., service layer monitoring), the task becomes easier by using our approach. The principles documented here serve as a contribution to others needing this form of abstraction for adding crosscutting concerns to application frameworks, like the OSGi framework, in the same structured way as we did.

In OSGi, our approach focuses on code that lies in-between the interaction of the bundle layer (the components deployed at runtime) with the lower layers. The aspects would use the OSGi framework as the point of interception. Code that concerns the internals of bundles implementation does not interest us. Therefore, pointcuts are defined using join points of the OSGi API. For that reason we weave only the framework and not the OSGi bundles.

We have left the security layer out of our scope since it is an optional layer according OSGi's specification. Besides clearly crosscutting all layers, as illustrated in Figure 1, the join points related to security are easily identifiable in the OSGi specification, which details all methods and corresponding interfaces that need to perform security verifications in each of the layers. In addition, existing work [38] already has contributions handling security as aspects in OSGi.

The next subsections detail the layer aspects, followed by a discussion on their reusability. We kept the pointcuts of the layer aspects as simple as possible, defining *kinded* pointcuts (in our case, execution and call) using join points in methods and constructors. Refinements of such pointcuts such as the combination with *non-kinded* pointcuts (e.g., control flow) were left to the specific aspects definitions that reuse the layer aspects.

3.1 Lifecycle Aspect

The methods and transitions that concern bundle lifecycle are scattered across four interfaces (Bundle, BundleContext, BundleActivator, PackageAdmin) that already have roles other than lifecycle management. Figure 2 shows the states and their respective transitions concerning a bundle's lifecycle in OSGi. The install state transition is actually fired in the BundleContext (BC in the figure) interface. The resolve transition is defined in the PackageAdmin (PA) service interface, while the update and uninstall can be found in the Bundle (B) interface. The refresh transition is part of the package admin, which is not part of the core API but rather declared in the PackageAdmin (PA). The start and stop transitions are both located in the Bundle and BundleActivator (BA) interfaces. In case of a Bundle having a BundleActivator, those calls are delegated to the activator. In the LifeCycle aspect we have rather called it as activation and deactivation, respectively.

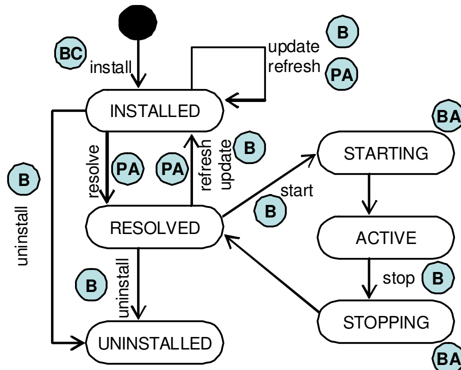

Figure 2. OSGi bundle lifecycle state transitions scattered over several interfaces: BundleContext (BC), Bundle (B), BundleActivator (BA), PackageAdmin (PA).

Figure 3 shows the LifeCycle aspect containing the corresponding pointcuts. Most of the pointcuts have used join point definitions that concerned interfaces whose implementations are provided by any OSGi framework. Only the *activate* and *deactivate* pointcuts, respectively, have been defined using *call* join points. This happens due to the fact that an OSGi framework implementation itself does not provide implementors of the BundleActivator interface. BundleActivators are rather provided by bundles that will be loaded by the framework. Calls to start and stop lifecycle transitions are done toward the framework, which performs its work and then delegates the calls to the start and stop methods of the BundleActivator from the corresponding bundle. Because we weave only the framework, not applying aspects to a bundle's code, we cannot apply *execution* join points in such transition. Instead, we use a *call* join point on the OSGi framework side.

```
public aspect LifeCycle {

 pointcut install():
     execution(Bundle
BundleContext+.installBundle(String,..));

  pointcut stop():
     execution(void Bundle+.stop(..));

  pointcut start():
     execution(void Bundle+.start(..));

  pointcut uninstall():
     execution(void Bundle+.uninstall());

  pointcut update():
     execution(void Bundle+.update(..));

  pointcut resolve():
     execution(boolean
     PackageAdmin+.resolveBundles(Bundle[]));

  pointcut refresh():
     execution(void
    PackageAdmin+.refreshPackages(Bundle[]));

  pointcut activate():
     call(void
      BundleActivator+.start(BundleContext));

  pointcut deactivate():
     call(void
       BundleActivator+.stop(BundleContext));
}
```

Figure 3. Aspect defining bundle lifecycle pointcuts.

3.2 Service Layer Aspect

According to its specification, the service model in OSGi is based on a publish, find and bind model. All of these operations are centered around the service registry, which actually does not have a standard class or interface representing it in the API. The methods that give access to the service registry can be found scattered in different interfaces. In addition, implementations of a service registry may be completely different from one OSGi implementation to another. We reified the service registry as the aspect that represents the OSGi service layer, since we are mostly interested in the methods that concern the three operations of the OSGi's service model. The pointcuts that group the join points giving access to the service layer were grouped in the ServiceRegistry aspect, which is detailed in Figure 4.

Most of the pointcuts were defined using *execution* join points. However, similar to the join points used in the activate and deactivate pointcuts of the lifecycle layer, the join points concerning the ServiceFactory were declared as *call* join points since a ServiceFactory is an interface whose implementations are provided by bundles that are dynamically deployed instead of being provided by OSGi implementations. As a practical example

for using the service layer aspect, we could implement a service interception mechanism more powerful than the standard service hooks provided by the OSGi framework, which are very limited.

```
public aspect ServiceRegistry {

  pointcut registration():
     execution(ServiceRegistration
      BundleContext+.registerService(..));

  pointcut unregistration():
     execution(void
          ServiceRegistration+.unregister());

  pointcut retrieval():
     execution(Object
      BundleContext+.getService(
               ServiceReference ))
     || call(Object
          ServiceFactory+.getService(Bundle,
                ServiceRegistration));

  pointcut release():
     execution(boolean
BundleContext+.ungetService(ServiceReference))
     || call(void
     ServiceFactory+.ungetService(Bundle,
                ServiceRegistration,
                Object));

  pointcut referenceQuery():
     execution(ServiceReference[]
BundleContext+.getAllServiceReferences(..))
     || execution(ServiceReference
 BundleContext+.getServiceReference*(..));

  pointcut bundleServices():
     execution(ServiceReference[]
      Bundle+.getRegisteredServices());

  pointcut usageQuery():
     execution(ServiceReference[]
            Bundle+.getServicesInUse());

  pointcut addListener():
     execution(void
      BundleContext+.addServiceListener(
            ServiceListener));

  pointcut removeListener():
     execution(void
        BundleContext+.removeServiceListener(
             ServiceListener));
}
```

Figure 4. Aspect that abstracts the service layer.

3.3 Module Layer Aspect

Although scattered in different interfaces that accumulate roles from different layers, the functionality of both service and lifecycle layers can be well identified in the OSGi API. However, we cannot say the same concerning the module layer. All the classloading and package visibility requirements are well defined in the OSGi core specification, but they are not explicit in the API. Also, most of the runtime behavior concerns implementation specific code, which may differ from one implementation to another. For example, the classloading mechanism of the Module Loader [18], used in both Oscar[5] and Felix OSGi implementations, differs from those of Equinox and Knopflerfish, but they must all comply with the OSGi specification.

One of the few methods of the module layer that are explicit in the API can be found in the Bundle.loadClass method. However, typical code does not necessarily use that method explicitly. It rather relies on Java's transparent classloading mechanism (e.g., automatically performed when instantiating a class for the first time). We have only defined three classloading related pointcuts, as detailed in Figure 5. Given that a bundle is the unit of modularization in OSGi, we also have included a pointcut that uses a join point for bundle construction.

```
public aspect ModuleLayer {

 pointcut bundleInstantiation():
              execution(Bundle+.new(..));

 pointcut classLoaderInstantiation():
     execution(ClassLoader+.new(..));

 pointcut getResource():
     execution(* Bundle+.getResource*(String));

  pointcut loadClass():
     execution(Class
          Bundle+.loadClass(String))
     || execution(Class
          ClassLoader+.loadClass(String));
}
```

Figure 5. Module layer abstraction.

The OSGi Package admin service stores metadata concerning packages and their bundle dependencies, which are related to the module layer. The module layer aspect is useful, for example, for tracking bundle creation or as an alternative mechanism for intercepting class loading for performing custom bytecode manipulations on classes known only at runtime (the typical case in a dynamic platform such as OSGi). Other less intrusive usages could be fine grained tracing of the classloading process (an alternative to the general command line `–verbose:class` option); tracking the creation of new classloaders provided to bundles; and so forth.

3.4 Reusable pointcuts

Hanenberg et al. propose the *separate pointcut* [20] aspect-oriented refactoring for avoiding redundant anonymous pointcut declarations. Indeed, separate pointcut declarations are a good practice for reusability. The typical solution proposed in [19] is to inherit from an abstract aspect and to provide the advice code referring to the inherited pointcuts. However, we have chosen to use the design principle of favoring composition instead of inheritance, taken from object-oriented design [15]. This choice was mainly due to inheritance limitations in AspectJ. Instead of creating an abstract aspect to be extended so it can be reused, we rather defined the pointcuts in reusable library aspects that map the points of interest of each of the corresponding target OSGi layers (i.e. Lifecycle, Service and Module layers), reusing them in the advices of our aspects, as illustrated in Figure 6.

[5] Oscar OSGi framework. http://oscar.ow2.org

If we analyze the semantics of an *is-a* relationship – which legitimates inheritance – between one concrete aspect and the library aspect that represents a layer, we do not have a 1 to 1 cardinality, which would justify single inheritance in most of the cases. We rather have a concrete aspect that may crosscut multiple layers. As some concrete aspects may crosscut layers and layers have been abstracted as aspects, a concrete aspect may need to use code – in this case, pointcuts– inherited from different layers. In an illustrative example we can consider that a given concrete aspect (e.g., service monitoring) may affect two layers, (e.g., module and service layers) which are represented as aspects as well. In cases like this we could see the single inheritance provided by AspectJ as a limitation, since we can only inherit from one aspect at a time. If AspectJ provided multiple inheritance it could be solved in a straightforward manner. However, by using composition we could easily workaround this issue, thus making possible to create aspects reusing pointcuts from different origins (i.e., the layer aspects).

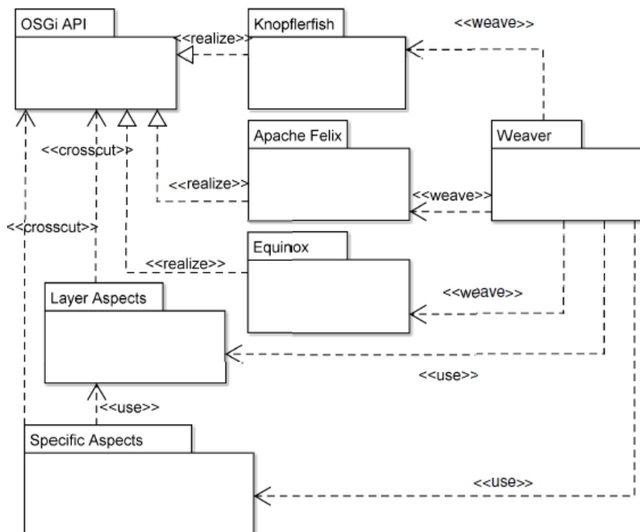

Figure 6. Diagram illustrating the weaver inputs and how the aspects are applied to different OSGi implementations.

As shown in Figure 6, the aspects target the OSGi API without direct dependencies on any of the OSGi implementations (Knopflerfish, Apache Felix, Equinox). The specific aspects are the concrete aspects where we would implement the advices to the selected layer pointcuts, thus reusing their definition. The weaver would take any OSGi implementation as an input together with the layer aspects (used as library aspects) and the specific aspects (which reference the layer aspects), weaving them both into the OSGi implementation. The resulting woven OSGi implementations contain the code that handles the specific aspects that have been developed targeting the layers and the OSGi API.

4. DEPENDABILITY ASPECTS

The layer aspects described so far do not provide advices. This section is a showcase for illustrating the reuse of such abstractions in the creation of specific aspects that are concerned with dependability and monitoring. In our precedent work [14] we patch the OSGi framework in order to increase dependability by reducing the risk of problems previously mentioned. This is done by transparently providing infrastructure that would (a) deploy and execute untrustworthy third-party code in a fault contained environment, and (b) enable the automatic recovery of applications in case of faults or failures. We use a sandbox approach for introducing fault contained boundaries that prevent a component hosted in the sandbox from crashing or interfering the execution of the main application. If sandbox hosted code mishandles OSGi dynamicity we are able to purge the sandbox from memory by performing a full sandbox reset without needing to stop the main application. The recovery approach is based on a combination of techniques taken from autonomic computing [23] and recovery-oriented programming. The former strategy is the basis for introducing a self-healing approach (automatic detection diagnosis and repair of problems) to the sandbox, where components may present unstable behavior. The latter is realized by means of microreboots [4], which propose the individual reboot of fine-grained components. Microreboots achieve similar benefits to an application restart with less cost and without losing application availability. The repair strategy performs microreboots in two levels: components and sandbox (as a last resort).

The code in our precedent solution was manually introduced as a patch on the implementation of Apache Felix v.1.4.0. We refactored these cross-cutting concerns into fine grained aspects, reusing our layer aspects abstraction. Figure 7 illustrates the reuse of the layer aspects in the creation of specific aspects concerned with dependability and monitoring. The layers avoided redundant pointcut definitions and allowed to explicitly identify which layers were being affected by an aspect. For example, the *use* stereotype clearly shows which aspects crosscut which layers. All of the instances of the dependability aspects did not need to have any particular association with classes, objects or control flow. Therefore, they have been implemented with the default `issingleton()` association.

We implemented two groups of cross-cutting concerns: isolation and monitoring. The isolation mechanisms help enforcing dependability towards the problems we try to address, while monitoring gives application information that we can analyze for either automatic or manual decisions on autonomic management helping on the detection and prevention of faulty behaviour.

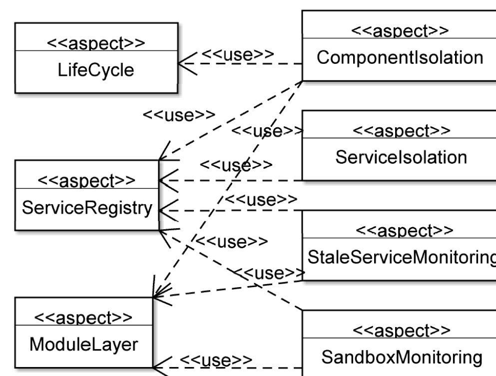

Figure 7. Layers (left side) abstracted as aspects whose pointcuts are reused by the dependability aspects (right side).

4.1 Isolation

By isolation we mean establishing boundaries that isolate one object from another, eliminating direct binding when it may bring risks such as using bindings to/from objects of a third-party component which is not known in advance, or which has not been exhaustively tested with the current set of components in a running application. We implemented two isolation levels: services and components. The first one is performed within the same application boundary (e.g., process, domain) and does not provide fault containment. It introduces proxies for isolating service consumer from service provider, ensuring that references to the servant object are not retained after it is unregistered. The second level concerns a sandboxing approach that hosts components in a fault contained boundary that can be fully restarted without interfering with the execution of the main application. Both of the isolation levels are resolved dynamically at runtime upon service retrieval or component start up. The decision is taken based on the analysis of a policy file that is defined using a domain-specific language that we have created for indicating services to be isolated (e.g., `implements Foo.BarService`) as well as components (e.g., `providedby ComponentVendorA`). The next subsections detail how the isolation strategies have been modularized as aspects. Due to space limitations, code snippets have been simplified for illustrative purposes.

4.1.1 Component Isolation Aspect

The sandboxing approach for isolating components has two techniques for isolation: domain based and operating-system based (i.e. process-based) which we compare against each other in our precedent work [14]. Both of these approaches provide fault contained entities –Java Isolates[6] and JVM process, respectively – with strong isolation boundaries, therefore implying in IPC mechanisms for communicating across such boundaries. The architecture of the sandboxing solution, detailed in previous work [12], takes the form of two OSGi platforms: a main OSGi platform where the trusted components execute, and a sandboxed OSGi where untrusted components are deployed and run, based on an isolation policy.

The component isolation aspect crosscuts the different lifecycle transitions, and also the service registry for allowing services running in one isolated platform to be used in the other, across the isolation boundary. All of these adaptations are encapsulated in aspects that target the OSGi platform. The dynamically deployed components are not changed; therefore, from the components perspective, our approach provides seamless component isolation and communication. The communication between the two isolated OSGi framework instances (main platform and sandbox platform) is done through a class that acts as a transparent proxy to the isolated platform, allowing for instance one isolated OSGi platform (e.g. the main platform) to retrieve services located in the other isolated OSGi platform (e.g. the sandbox) via IPC. For example, when a component is installed in the main (trusted) platform the ComponentIsolation aspect installs it in the sandbox. At component startup, the corresponding advice verifies the policy and if necessary starts the component in the sandbox, as shown in the first advice described in Figure 8.

We also had to avoid reentrant calls on the advices of some pointcuts. For instance, the implementations of Bundle.start() typically call Bundle.start(int), caught by the same pointcut. We simply added a cflowbelow construct, as described in that advice. Local queries to the service registry that bring no match are re-routed to the isolated platform. If a match is then found, the aspect would return an IsolatedServiceReference. Retrieval of service objects using such references generate a proxy that transparently handles the communication between the two platforms, as depicted in the second advice of Figure 8. Every component isolation patch we made in the Felix implementation could be easily migrated to the ComponentIsolation aspect, except for one. The notification of service events from one platform to the other was implemented directly in the EventDispatcher class, which is specific to Felix. In this case we had to adapt a dispatcher that was registered as a ServiceListener in OSGi and was responsible for filtering and propagation of service events to the other platform. Listener registration is done on the initialization of the isolation library, done via a ModuleLayer.bundleInstantiation pointcut.

```
public aspect ComponentIsolation {
...
void around(Bundle b): LifeCycle.start()
     && !cflowbelow(LifeCycle.start())&& this(b){
 if (!PlatformProxy.isSandbox() &&
     PolicyChecker.checkIsolation(b)){
     PlatformProxy.start(b.getBundleId());
  } else {
    proceed();
  }
 }

 Object around(ServiceReference ref):
       ServiceRegistry.retrieval() && args(ref) {
   Object service = null;
   if (ref instanceof IsolatedServiceReference) {
    Bundle b = ref.getBundle();
    service = getIsolatedProxyService(b, ref);
   } else {
    service = proceed(ref);
  }
  return service;
 }
...
}
```

Figure 8. Advices reusing pointcuts of different layer aspects.

4.1.2 Service Isolation Aspect

This aspect is responsible for replacing service objects by service proxies (when the isolation policy applies) that delegate the calls to the wrapped service object. By using this approach we can enforce that an untrustworthy service consumer does not directly reference service objects. It avoids cases of stale references to services that have already been unregistered (e.g. its bundle has been stopped). With this strategy, we can guarantee that service consumers will not prevent the garbage collection of unregistered services. In addition, the proxies are programmed to throw exceptions in cases of method calls when the underlying service object is unregistered. Figure 9 depicts the process of wrapping the service object in a proxy upon service retrieval.

[6] Java Isolation API. http://jcp.org/en/jsr/detail?id=121

```
public aspect ServiceIsolation {
...
 Object around(ServiceReference ref):
      ServiceRegistry.retrieval() && args(ref){
   Object s = proceed(ref);
   if (!PlatformProxy.isSandbox()
       && PolicyChecker.checkIsolation(s)) {
    s = ProxyServiceStore.getProxy(s,ref);
   }
     return s;
   }
...
}
```

Figure 9. Main advice of the ServiceIsolation aspect.

4.2 Monitoring

The other category of aspects that we have developed concerns monitoring. Although they do not enhance dependability by themselves, they help gathering information used for detecting as well as predicting faulty behaviour.

4.2.1 Stale Services Monitoring Aspect

We had previously used Aspect-oriented Programming (AOP) for monitoring in the ServiceCoroner tool [11] but in a less structured manner. That solution has been refactored and integrated to the OSGi dependability enhancements described here. We track service instances using Java weak references, in order to know if unregistered services are still referenced by other objects (when a weak reference returns `null` it means that the object it pointed to has been garbage collected). Classes or interfaces, both represented by java.lang.Class instances, that are no longer reachable can be reclaimed (when no instances of such classes are being referenced) and like so their classloader [16], as long as the classloader is no longer reachable and all of its loaded classes are no longer being used. We also track classloader instances with weak references, identifying which classloader objects still hang in memory after OSGi bundle updates. Our experiments show that when the objects and types of a given bundle are no longer referenced, the bundle classloader is garbage collected. The effectiveness of our approach using weak references has been detailed in the experiments presented in [11]. Figure 10 shows a simplified example of the aspect that forwards the service instance tracking to the ServiceCoroner API.

```
public aspect ServiceMonitoring {
...
 Object around(ServiceReference ref):
   ServiceRegistry.retrieval() && args(ref) {
     Object result = proceed();
     ServiceCoroner coroner =
           ServiceCoroner.getInstance();
     coroner.trackService(ref, result);
     return result;
   }
...
}
```

Figure 10. Aspect for monitoring services garbage collection.

If both component isolation and service monitoring aspects are used together, we must explicitly define the order of precedence so we can be sure that the service monitoring will track always the actual servant object instead of tracking a proxy to a service.

4.2.2 Autonomic Management Aspect

The self-healing capability of the sandbox is achieved via autonomic management which is actually provided by an external application that provides a control loop. It collects information from the sandbox via monitoring probes, analyzes the data and takes appropriate action (e.g., stopping a bundle, rebooting the sandbox) through effector probes implemented as Java MBean. The insertion of such probes is done by the sandbox monitoring aspect on the creation of the first bundle through the module layer, as depicted in the simplified example of Figure 11.

The service layer is also used by this aspect, but in quite a similar way to the approach from section 4.1.2, based on proxies. The proxy enables, for instance, calculating service usage. A particular difference on this aspect is that it also weaves our own classes in order to monitor the interactions with the isolated platform via their proxies. The probe information also depends on our ServiceCoroner API (fed by the service monitoring previously describe), in order to take action against stale services. The fault prediction mechanisms are available for a set of patterns of errors: CPU hogging, stale service, excessive memory allocation; excessive thread instantiation; excessive invocation of services (Denial of Service); stale reference retention. The detection and handling of such faults was provided as customizable scripts that are loaded and executed by the sandbox autonomic manager.

```
public aspect SandboxMonitoring {
...
 void around(Bundle bundle) :
ModuleLayer.bundleInstantiation()&& this(bundle){
 if (bundle.getBundleId() == 0) {
  ObjectName name = new
     ObjectName("fr.imag.adele:type=Touchpoint");
  Touchpoint mbean=new Touchpoint();
  mbean.setSystemBundle(bundle);
ManagementFactory.getPlatformMBeanServer().regist
erMBean(mbean, name);
 }
}
...
}
```

Figure 11. Creation of the sandbox monitoring probe aspect.

5. VALIDATION

Since we have not found similar solutions for isolation and monitoring in OSGi applications we could not compare our implementation against other approaches. This section explains the validation steps —weaving both layer and dependability aspects into different OSGi implementations; and validating the effectiveness of the dependability aspects in a test application— followed by limitations and benefits of our approach.

5.1 Weaving different OSGi implementations

We have successfully woven and tested different versions of three OSGi implementations (Apache Felix, Equinox, Knopflerfish) that are widely used in software industry. The weaving of layers and aspects happened with no problems, and the dependability aspects correctly worked, as detailed further. As part of our evaluation, we extracted some metrics (Table 1) concerning the layer abstraction through aspects, for each tested implementation. We verified how many join point shadows have been found in the classes affected by each of the layer aspects from section 3, so we

could have a perspective of the scattering phenomena in the analyzed OSGi implementations. Although the number of affected classes may seem small, we want to illustrate that there is no single point of access for layers. We also show that the layer concepts are lost in the API, since the classes that contain the join point shadows have other responsibilities than exposing layer services. Likewise, we find classes whose responsibilities overlap different layers. We collect such scattered concepts, and expose as an entity that contains the entry points to a given layer. Another observation that can be made is that Felix and Knopflerfish join point shadows remain stable across different versions, while Equinox shows a significant increase from one version to another.

Concerning woven OSGi frameworks execution, two adjustments had to be done. First, to avoid issues with type visibility in OSGi, we embedded the AspectJ runtime classes in each one of the woven OSGi implementations. The second issue concerned the Equinox OSGi framework jar file which stores in its manifest an SHA1-Digest for each class present in the jar. After the weaving process, the woven classes had their hashes no longer valid and we had security verification errors at OSGi startup. The workaround was to remove such information from the framework bundle manifest file so it could be started up. However, the fact of having the OSGi framework bundle without SHA1 hashes does not influence in the verification process of any other loaded bundles that contains SHA1 hash information. It only means that the framework will not perform that verification against itself at startup, but other bundles will be verified. To illustrate that, the other two implementations (Felix and Knopflerfish) do not provide SHA1 hashes in their manifests but they are able to verify digitally signed jars that are loaded by the framework.

Table 1. Layer scattering over OSGi API: total join point shadows (JPS), affected classes (C) and packages (P).

	Lifecycle			Service			Module		
	JPS	C	P	JPS	C	P	JPS	C	P
Felix 1.4	22	5	2	15	4	1	10	4	2
Felix 2.0.4	22	5	2	14	3	1	7	3	1
Felix 3.0.3	22	5	2	14	3	1	8	3	1
Knopflerfish 2.3.1	17	4	1	15	6	1	7	3	1
Knopflerfish 3.0	18	5	2	18	7	2	12	5	2
Equinox 3.4	18	4	1	16	5	1	17	9	5
Equinox 3.6.1	38	9	4	20	9	4	33	16	9

5.2 Effectiveness of the Dependability Aspects

We have validated the aspects in a simulation of an RFID and sensor-based application, also used in [14]. The application consisted of an OSGi application that simulates the collection of RFID and sensor data with a total of 14 bundles. Sensors and RFID reader simulator components were hosted in the sandbox. One the motivating scenarios concerns applications that collect RFID and sensor data. The application illustrates a scenario where we typically use native drivers wrapped in Java components to access physical devices. Devices may be plugged and detected at runtime, as are their respective drivers. The interaction between the application components that consume data provided by the untrustworthy code is done through OSGi's service layer. In case of an illegal operation or a severe fault in the native code, the whole application is compromised. In this use case the application must also run non-stop and be able to recover in case of such severe faults, therefore we employ different dependability aspects that are woven in the OSGi framework using our proposed approach. Since it is a dynamic component-based application, we could have three different applications (i.e. configurations) running: an RFID-only application, a sensor-only application, and a hybrid application where both types of devices provide data.

Different test cases were executed for verifying: policy-based isolation of components and services; lifecycle operations on the isolated component; calls on services from isolated platform; stale service monitoring; sandbox monitoring. For testing the last two we had to deploy components that caused errors during their execution. Testing systems with faults injected in the interface level (e.g., invalid parameters) instead of faults injected in the component level (e.g. emulation of internal component errors) produces different behavior [30], which does not representing actual application usage. For that reason we have chosen to use component fault injection that could reflect possible faults, providing us with a realistic scenario. Therefore *fault deployment* would be a more appropriate term since the faulty components are deployed and started at runtime.

The autonomic manager works as an external application that is not directly affected by the aspects. However, it collects data from the sandbox through the sandbox monitoring aspect. The autonomic manager, as expected, performed microreboots of the sandbox in different handled cases: when it was non-responsive (i.e. CPU hang); when it was crashed; when the monitored data collected indicated sandbox errors such as excessive thread instantiation, memory or CPU usage. In cases where the source bundle could be identified (stale service usage and excessive invocation of services), the autonomic manager performed microreboots on the offending bundle. Such automatic decisions are taken based on the event history of monitored data stored for each cycle of the autonomic manager's control loop.

5.3 Limitations

Despite the single inheritance issue on AspectJ, we have not found any impeditive limitations concerning the usage of aspects. Performance overheads of the aspectized version in comparison to the patched by-hand version were not checked. However, we believe that the impact is minimal, based on studies [21] indicating that a woven application that captures a given crosscutting concern with AspectJ has performance comparable to the same implementation made by-hand.

The limitations refer to the technical solution introduced by the dependability and monitoring aspects implementation. These limitations are of a fundamental nature and also concerning our current solution. For instance, trying to provide transparent IPC is still a challenging subject in systems engineering. For that reason we provide a restricted flavor of transparent communication where we do not deal with object serialization, so we can avoid the design decision of objects passed by-copy or by-ref. For such reasons, we restrict the usage of our approach to applications that comply with a set of assumptions [14]. In order to be sandboxed, a component needs to have services that are stateless (to avoid state loss in case of reboots) and that provide methods with signatures limited to Strings, primitive types and arrays of these two. In contrast, if a sandboxed component has services with

signatures using objects it will not work with our approach, having runtime exceptions. Another drawback of our solution involves sharing of security permissions between the trusted platform and the sandbox, especially in the OS-based approach where OSGi platforms are separate processes. As the two platforms virtually constitute the same application, we consider the same level of security for both of them. However since potentially untrustworthy code can be executed, we should consider a way of restricting permissions to the sandbox. Other limitation of fundamental nature concerns OSGi technology itself, which does not provide resource consumption monitoring of individual bundles. At bundle level, we could only monitor precisely the service layer (e.g. service invocations) by means of our aspects for autonomic management.

5.4 Benefits

From a general perspective, we find that the strategies we have employed are useful to whoever needs to apply crosscutting concerns to the OSGi framework in a similar way. With the abstraction of OSGi layers as aspects we could identify concepts that were not clear in the OSGi API. It improved the understanding of the API and gives a *better architectural perspective* of which layers are being affected by a given crosscutting concern. We also gained *reuse* of the pointcut definitions from the layer aspects that were referenced by specific aspects (e.g. dependability, monitoring) that crosscut such layers, avoiding redundancy of such definitions. For different reasons, we believe that the aspectized solution of the dependability concerns was better than the version patched by-hand. The refactoring of our dependability crosscutting concerns into aspects helped improving the *modularity* of our solution. This explicit separation of concerns enhanced *maintainability* of the dependability concerns, facilitating their evolution with no need to manually change the target OSGi implementations' source code**.** One may see this solution as invasive due to the changes performed in the OSGi implementations during the weaving process; however we gain *portability* across different OSGi frameworks. Also, separating our crosscutting concerns into distinct aspects brings *flexibility*, by having the possibility of combining different aspects, providing a sort of *à la carte* choices. For instance, the usage of the service layer monitoring without the isolation features. In that case, only the desired aspect and the layer aspects are needed to weave the target OSGi implementation.

6. RELATED WORK

6.1 Separation of Concerns and AOP in OSGi

Handling service registration/unregistration consists of repetitive and error prone code. By using separation of concerns, different efforts try to tackle issues originated from OSGi's dynamism. The Service Binder, presented in [5], keeps an engine for automatically handling service dependencies at runtime. Later, that research effort was enhanced and integrated to the OSGi specification as the Declarative Services standard. iPOJO [8] is a component model targeting the OSGi platform and, in relation to Service Binder, takes a step further for managing the dynamicity and other non-functional requirements in OSGi applications. iPOJO employs strategies such as method interception and bytecode manipulation which are both found in several AOP frameworks. By means of *handlers*, iPOJO tries to provide a clean separation of concerns keeping non-functional code (e.g., service provisioning, dependency management) outside components.

To the best of our knowledge there were no other approaches in literature that introduce aspects directly to the OSGi framework. Our work focuses on transparent enhancements in OSGi frameworks (affecting different OSGi layers) in order to enhance dependability without needing to change the code of existing applications. Other efforts rather try to introduce aspects as part of OSGi bundles dynamically deployed, but that do not necessarily crosscut all OSGi layers. For example, Phung and Sands [35] use aspects for implementing different security policies such as actions suppression, insertion, truncation and replacement. They use AspectJ to weave the aspects into the bundle at download/installation time. The weaving is performed by a trusted control center who is asked for a given bundle.

While we have focused on *introducing* crosscutting concerns into OSGi by means of aspects, Singh and Kiczales [38] have focused on *refactoring* existing crosscutting concerns in the Equinox OSGi implementation. That work was part of a refactoring of the Eclipse IDE, which is based on OSGi. Among their significant findings, they have treated security as a crosscutting concern. Typical security checks in OSGi handle conditional permission check concerning the rights of a bundle to access a given service or to load a given class, for example. They moved to an aspect all calls to the Security Manager (found in several methods that required permission). Lifecycle was handled in a way different than ours. In their case, the state transitions were identified as a finite state machine. The code that handled state transitions was moved to the corresponding aspect, and plugin state changes performed outside the aspect were declared as an error. Frei and Alonso [10] adapted an OSGi framework implementation in order to register services for using AOP through an AOPContext object instead of a BundleContext object. The AOP approach is based on proxies that would allow intercepting calls before and after method execution. The AOP library used needed some adaptation for dealing with multiple classloaders (one per bundle in OSGi).

Lippert's work concentrates on enabling the usage of AOP in OSGi bundles, providing load-time weaving. His initial work [27] on that context focused on enabling the usage of Aspect-J in the Eclipse runtime, before OSGi was adopted for the Eclipse platform. Right after OSGi was used as the Eclipse runtime, a new version of that work, called AJEER (Aspect-J Enabled Eclipse Runtime) [28], was adapted to that environment. It has evolved into the Equinox Aspects[7] project [29], where aspects can be deployed either with the bundles that would be woven, or as separate bundles. Following the OSGi dynamics, deploying aspects as separate bundles would fire the update of affected bundles, which would have their classes reloaded and woven at load-time. It is also possible to uninstall the aspect bundles, which results in an update of the affected bundles so the classes without woven code can be reloaded. Although Equinox Aspects provides a powerful mechanism, it relies on features specific to the Equinox OSGi implementation that are not part of the OSGi specification. Thus, this approach is not portable to other OSGi implementations, contrasting to the portability we propose. Keuler and Kornev [24] also use Equinox's class loader hook mechanism for manipulating the class loading performed in bundles. They

[7] http://eclipse.org/equinox/incubator/aspects

replace the base class loader of all bundles by an intermediary class loader that allows the aspects to be loaded. Irmert et al [22] combine JBoss AOP with the classloading hook mechanism from Equinox for building a mechanism that deploys aspects as OSGi bundles. Like other approaches, the portability of both solutions is compromised as they depend on a proprietary mechanism.

6.2 Autonomic Computing and AOP

Broadly speaking, several efforts such as [36] and [40], to cite a few, have used AOP to address dynamic adaptation. By narrowing down the vision to autonomic computing we can still find works that take advantage of AOP for introducing autonomic managers and monitoring capabilities into systems. In [7], Engel and Freisleben see the autonomic computing principles as crosscutting concerns. Their toolkit deploys aspects into the operating system kernel, allowing the self-adaptation mechanisms to be based on system operation and resource usage. They illustrate their approach for achieving, at the kernel level, degrees of self-optimization, self-configuration, self-healing and self-protection.

Chan and Chieu [6] mention the monitoring function of an application as a crosscutting concern. In their work they describe an approach for building autonomic managers in legacy systems by using AOP techniques for weaving them. Alonso et al. [1] present what they have called AOP-monitoring framework. In their approach they use AOP for injecting into the system some monitoring code, implemented as probes which will capture resources consumed by components. A monitor manager collects information sent by probes, and depending on the analysis of that data it takes appropriate action based on the policies to follow. Grenwood and Blair [17] give an example usage of AOP for building an autonomic cache. They present aspects that monitor requests on a server application and in case of a response time threshold being reached the monitoring aspect dynamically weaves the application for introducing new aspects (e.g., a caching aspect for enhancing response time).

6.3 Generally Related Approaches

We have not found approaches targeting dependability by using aspects for isolating components and services in dynamic applications. Other efforts deal with dependability through AOP, but with different strategies and focused on different objectives. Error handling [9] [26] is one the most addressed dependability concerns using AOP. Other approaches try to handle more general dependability mechanisms, like Rouvoy et al [37] who use AOP for combining dependability concerns with self-adaptive applications. They present predefined dependability mechanisms – like a transactional content processor, or a replicated content repository – that are kept separate from the core application, but that are combined with it when necessary (e.g., based on Quality of Service dimensions). We can find other slightly similar approaches with overlapping interests, for example, Soares et al [39] handle distribution as aspects, which relates to our efforts for transparent communication, but they rather rely on adding remote interfaces via the AspectJ `declare parents` construct.

7. CONCLUSIONS

Executing code not previously known in advance which is deployed at runtime may introduce potential risks to applications. Like so, the dynamic loading and unloading of native libraries in platforms like Java also introduces risks. We try to address such issues in the OSGi Service Platform, which targets the construction of dynamic modular Java applications. We propose an approach for introducing mechanisms for enhancing dependability in that platform. These techniques have been implemented in previous work in a patch targeting the Apache Felix 1.4.0 OSGi implementation, but the code introduced was scattered over different classes. It made the solution hard to be ported and maintained, since we would need to copy and paste source code in the target OSGi implementation versions.

We succeeded in separating crosscutting dependability concerns and refactoring them into aspects. During that process, we improved the modular representation of software layers of the OSGi platform, by abstracting them as an aspect library that provided reusable pointcuts. By using AOP we created better abstractions and achieved better expressiveness being able to represent concepts that were lost when the specification was translated into an API. We believe that such layer representation as aspects can help other developers that try to apply aspects to the OSGi API and may need such abstractions.

The reusable layer representation also helped to better identify which layers were being crosscut by the dependability aspects, whose original crosscutting concerns were broken down into more granular aspects. The whole aspectization has lead to a cleaner solution and also portable across different OSGi implementations. We have successfully woven our aspects into different versions of three OSGi implementations: Apache Felix, Equinox and Knopflerfish. After running the test application in the different woven OSGi frameworks, the runtime behavior of the isolation policies worked as in the preexisting hard-coded solution. The aspect approach brought several advantages to the OSGi dependability enhancements we propose. We have gained easy *portability* across different OSGi frameworks; better *modularity* and *maintainability* of code, and *flexibility* through the possible combination of different fine grained aspects.

We plan to evaluate the migration of certain aspects to be woven at runtime, and compare it to the weaving of OSGi framework implementations, since the portability of our solution across different OSGi implementations is one of our main objectives.

8. ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their invaluable comments and suggestions during both submission rounds. Thanks also to Lionel Seinturier and Walter Rudametkin for their important reviews and opinions. Part of this work has been carried out in the scope of the ASPIRE project (http://www.fp7-aspire.eu), co-funded by the European Commission in the scope of the FP7 programme under contract 215417. The authors acknowledge help and contributions from all partners of the project.

9. REFERENCES

[1] Alonso, J., Torres, J. Grith, R., Kaiser, G. and Silva, L.. Towards self-adaptable monitoring framework for self-healing. In Proc. of the 3rd CoreGrid Workshop on Middleware, June 2008.

[2] Alves, V., Matos, P., Cole, L., Vasconcelos, A., Borba, P., and Ramalho, G. 2007. Extracting and evolving code in product lines with aspect-oriented programming. In

Transactions on Aspect-Oriented Software Development IV, Lecture Notes In C. S., vol. 4640. Springer-Verlag, Berlin, Heidelberg, pp. 117-142.

[3] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M. Pattern-Oriented Software Architecture: A System of Patterns. Wiley, 1996.

[4] Candea, G., Kawamoto, S., Fujiki, Y., Friedman, G., and Fox, A. 2004. Microreboot — A technique for cheap recovery. In Proc. of the 6th Conference on Symposium on Operating Systems Design & Implementation – Vol. 6 (San Francisco, CA, Dec. 2004). Operating Systems Design and Implementation. USENIX Association, Berkeley, CA, 3-3.

[5] Cervantes, H. and Hall, R. S. 2004. Autonomous Adaptation to Dynamic Availability Using a Service-Oriented Component Model. In Proc. of the 26th International Conference on Software Engineering (May 23 - 28, 2004). IEEE Computer Society, Washington, DC, 614-623.

[6] Chan, H. and Chieu, T. C. 2003. An approach to monitor application states for self-managing (autonomic) systems. In Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (Anaheim, CA, USA, October 26 - 30, 2003). OOPSLA '03. ACM, New York, NY, 312-313.

[7] Engel, M. and Freisleben, B. 2005. Supporting autonomic computing functionality via dynamic operating system kernel aspects. In Proc. of the 4th international Conference on Aspect-Oriented Software Development (AOSD). Chicago, Illinois, March 14 - 18, 2005. ACM, New York, NY, 51-62.

[8] Escoffier, C., Hall, R.S. and Lalanda, P. 2007. iPOJO: an Extensible Service-Oriented Component Framework. In Proc. of the IEEE International Conference on Services Computing (July 09 - 13, 2007). IEEE Computer Society, Washington, DC, 474-481.

[9] Filho, F. C., Garcia, A., and Rubira, C. M. 2007. Error handling as an aspect. In Proc. of the 2nd Workshop on Best Practices in Applying Aspect-Oriented Software Development. vol. 211. ACM, New York, NY, 1

[10] Frei, A. and Alonso, G. 2005. A Dynamic Lightweight Platform for Ad-Hoc Infrastructures. In Proc. of the Third IEEE international Conference on Pervasive Computing and Communications (March 08 - 12, 2005). PERCOM. IEEE Computer Society, Washington, DC, 373-382.

[11] Gama, K. and Donsez, D. 2008. Service Coroner: A Diagnostic Tool for Locating OSGi Stale References. In Proc. of the 34th Euromicro Conference Software Engineering and Advanced Applications (SEAA 2008). IEEE Computer Society, Washington, DC, 108-115.

[12] Gama, K. and Donsez, D. 2008. A Practical Approach for Finding Stale References in a Dynamic Service Platform. In Proc. of the 11th Intl. Symposium on Component-Based Software Engineering (CBSE 2008). Lecture Notes In C. S., vol. 5282. Springer-Verlag, Berlin, Heidelberg, 246-261

[13] Gama, K. and Donsez, D. 2009. Towards Dynamic Component Isolation in a Service Oriented Platform. In Proceedings of the 12th Intl. Symposium on Component-Based Software Eng. (CBSE 2009). Lecture Notes In C. S., vol. 5582. Springer-Verlag, Berlin, Heidelberg, 104-120

[14] Gama, K. and Donsez, D. 2010. A Self-healing Component Sandbox for Untrustworthy Third-party Code Execution. In Proc. of the 13th Intl. Symposium on Component-Based Software Engineering (CBSE 2010). Lecture Notes In C.S., vol. 6092. Springer-Verlag, Berlin, Heidelberg.

[15] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995 Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Longman Publishing Co., Inc.

[16] Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification 3rd Edition. pp. 330--331, Addison-Wesley, 2005

[17] Greenwood, P. and Blair, L. Using Dynamic AOP to Implement an Autonomic System. In: Dynamic Aspects Workshop. 2004. Lancaster, UK

[18] Hall, R.S. A Policy-Driven Class Loader to Support Deployment in Extensible Frameworks. In Proc. of the International Working Conference on Component Deployment, pp 81--96. Springer, May 2004.

[19] Hanenberg, S. and Unland, R. Using and reusing aspects in AspectJ. In Workshop on Advanced Separation of Concerns in Object-Oriented Systems, OOPSLA '2001, Oct. 2001.

[20] Hanenberg, S., C. Oberschulte and R. Unland, Refactoring of aspect-oriented software. In Proc. of Net.ObjectDays Conference (NODe'03), 2003

[21] Hilsdale, E. and Hugunin, J. 2004. Advice weaving in AspectJ. In Proc. of the 3rd international Conference on Aspect-Oriented Software Development. AOSD '04. ACM, New York, NY, 26-35

[22] Irmert, F., Lauterwald, F., Bott, M., Fischer, T., and Meyer-Wegener, K. Integration of dynamic AOP into the OSGi service platform. In Proc. of the 2nd Workshop on Middleware-Application Interaction, vol. 306. ACM, 2008, New York, NY, 25-30.

[23] Kephart, J. O. and Chess, D. M. 2003. The Vision of Autonomic Computing. Computer 36, 1 (Jan. 2003), 41-50

[24] Keuler, T. and Kornev, Y. 2008. A light-weight load-time weaving approach for OSGi. In Proc. of the 2008 Workshop on Next Generation Aspect-oriented Middleware (Brussels, Belgium, 2008). NAOMI '08. ACM, New York, NY, 6-10.

[25] Kon, F., Roman, M., Liu, P., Mao, J., Yamane, T., Magalhaes, L. and Campbell, R. 2000. "Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB," In Proc. of the Middleware 2000 Conference, ACM/IFIP.

[26] Lippert, M. and Lopes, C. V. 2000. A study on exception detection and handling using aspect-oriented programming. In Proc. of the 22nd international Conference on Software Engineering. ICSE '00. ACM, New York, NY, 418-427.

[27] Lippert, M. 2003. An AspectJ-enabled eclipse core runtime platform. In Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Anaheim, CA, USA, 2003. ACM, New York, NY, 322-323.

[28] Lippert, M. 2004. AJEER: an AspectJ-enabled Eclipse runtime. In Companion To the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems,

Languages, and Applications (Vancouver, BC, CANADA, 2004). OOPSLA '04. ACM, New York, NY, 23-24.

[29] Lippert, M. 2008. Aspect weaving for OSGi. In Companion To the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications (Nashville, TN, USA, October 19 - 23, 2008). OOPSLA Companion '08. ACM, New York, NY, 717-718.

[30] Moraes, R., Barbosa, R., Duraes, J., Mendes, N., Martins, E., and Madeira, H. 2006. Injection of faults at component interfaces and inside the component code: are they equivalent?. In: 6th European Dependable Computing Conference. IEEE Computer Society, Washington, DC 53-64

[31] OSGi Alliance. http://www.osgi.org

[32] OSGi Service Platform Release 4 Version 4.2 Core Specification. http://www.osgi.org/Download/Release4V42

[33] Papazoglou, M. P. 2003. Service -Oriented Computing: Concepts, Characteristics and Directions. In Proc. of the Fourth international Conference on Web information Systems Engineering (December 10 - 12, 2003). WISE. IEEE Computer Society, Washington, DC.

[34] Plásil, F., Bálek, D., Janecek, R. 1998. SOFA/DCUP: Architecture for Component Trading and Dynamic Updating. In Proceeding of International Conference on Configurable Distributed Systems. pp. 43

[35] Phung, P.H. and Sands, D. 2008. Security Policy Enforcement in the OSGi Framework Using Aspect-Oriented Programming. 32nd Annual IEEE Intl Computer Software and Applications (COMPSAC 2008). pp.1076-1082

[36] Redmond, B. and Cahill, V. 2002. Supporting Unanticipated Dynamic Adaptation of Application Behavior. In Proc. of the 16th European Conference on Object-Oriented Programming (2002). Lecture Notes In Computer Science, vol. 2374. Springer-Verlag, London, 205-230

[37] Rouvoy, R., Eliassen, F., and Beauvois, M. 2009. Dynamic planning and weaving of dependability concerns for self-adaptive ubiquitous services. In Proc. of the 2009 ACM Symposium on Applied Computing (Honolulu, Hawaii). SAC '09. ACM, New York, NY, 1021-1028.

[38] Singh, A. and Kiczales, G. 2007. The scalability of AspectJ. In Proc. of the 2007 Conference of the Center For Advanced Studies on Collaborative Research (Richmond Hill, Ontario, Canada, October 22 - 25, 2007). CASCON '07. ACM, New York, NY, 203-214.

[39] Soares, S., Laureano, E., and Borba, P. 2002. Implementing distribution and persistence aspects with AspectJ. In Proc. of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (Seattle, Washington, USA, November 04 - 08, 2002). OOPSLA '02. ACM, New York, NY, 174-190

[40] Yang, Z., Cheng, B. H., Stirewalt, R. E., Sowell, J., Sadjadi, S. M., and McKinley, P. K. 2002. An aspect-oriented approach to dynamic adaptation. In Proc. of the First Workshop on Self-Healing Systems (Charleston, South Carolina, November 18 - 19, 2002). D. Garlan, J. Kramer, and A. Wolf, Eds. WOSS '02. ACM, New York, NY, 85-92