Proposal for a suspend/resume extension to the OSGi specification

Ronan Dunklau, Stéphane Frénot

Proposal for a suspend/resume extension to the OSGi specification

Ronan Dunklau — Stéphane Frénot

N° 7060

June 2009

Thème COM

Rapport technique

ISSN 0249-0803 ISRN INRIA/RT--7060--FR+ENG

Proposal for a suspend/resume extension to the OSGi specification

Ronan Dunklau, Stéphane Frénot

Thème COM — Systèmes communicants
Projet Amazones

Rapport technique n° 7060 — June 2009 — 18 pages

Abstract: The OSGi platform is a service-oriented architecture allowing the deployment of software components (called bundles). This report proposes an architecture allowing a bundle activity to be suspended and later resumed in the OSGi platform. This architecture consists primarily in an extension to the OSGi specification. In order to ease the development of bundles according to this extension to the specification, a centralized thread management is proposed. An implementation of these extensions is presented. It is based on Apache Felix, a particular implementation of the OSGi specification.

Key-words: OSGi suspend resume extension

Proposition d'une extension de la norme OSGi pour la suspension et la reprise d'activité

Résumé : La plate-forme OSGi est une architecture orientée services permettant le déploiement de composant logiciels (appelés bundles). Ce rapport propose une architecture permettant la suspension / reprise d'activité de bundles au sein de la plate-forme OSGi. Cette architecture se compose en premier lieu d'une extension à la spécification OSGi. Pour faciliter le développement de bundle respectant cette extension à la spécification, une gestion centralisée des threads est proposée. Enfin, le rapport détaille l'implémentation effective de ces extensions au sein d'Apache Felix, une implémentation particulière de la spécification OSGi.

Mots-clés : OSGi suspension reprise extension

1 Preliminary work

This work proposes an extension to the OSGi framework to enable a suspend mechanism. The main point is to enable OSGi bundle to freeze their state in order to resume their behavior later. We firstly present the OSGi framework related to the bundle life-cycle then we go much deeper into our motivations.

1.1 Overview of the OSGi specification

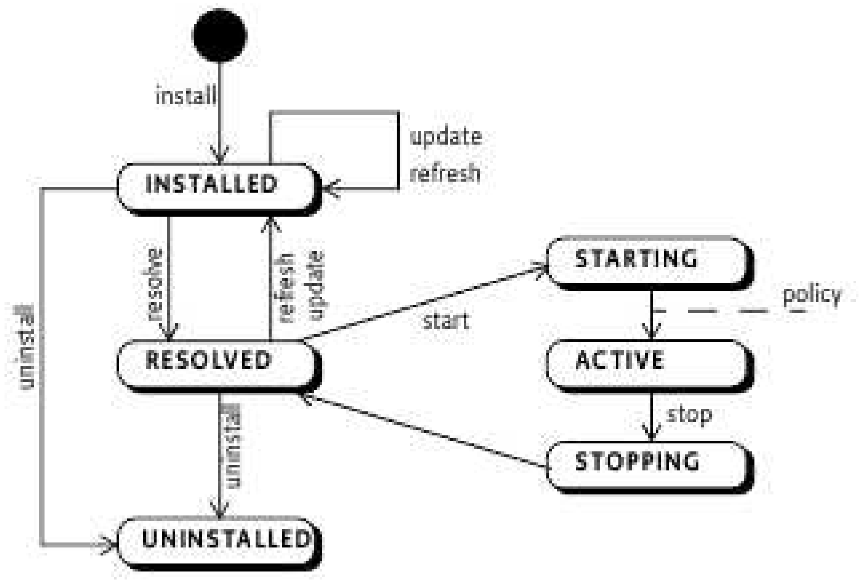

Figure 1: Bundle Life Cycle (from OSGi specification) [1]

The OSGi (Open Service Gateway initiave) specification proposes a SOA (Service Oriented Architecture) based on the Java programming language. It is used by a broad range of projects, from complex applications (Eclipse IDE, Glassfish application server ...) to applications deployment on devices such as mobile phones or domestic service gateways.

This architecture is centered around the concept of bundle: a bundle is a component which can be installed, started, stoped or undeployed on the OSGi platform.

The specification proposes a remote bundle deployment model, and the bundle life-cycle management with a dedicated API.

Bundles have a specific life-cycle, which is managed by the framework (as showed on figure 2). This life-cycle is composed of the following states (see figure 1):

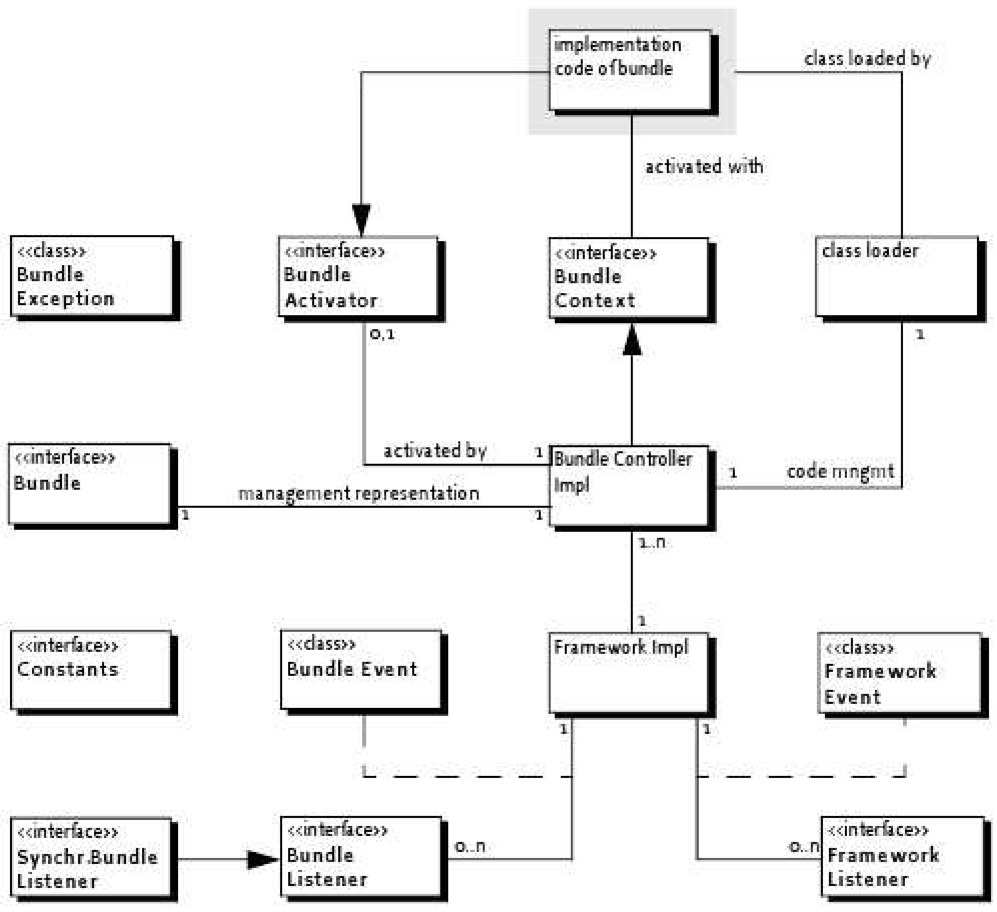
implementation code of bundle
class loaded by
activated with
<<class>> Bundle Exception
<<interface>> Bundle Activator
<<interface>> Bundle Context
class loader
0,1
1
activated by
1
Bundle Controller Impl
1
code mngmt
<<interface>> Bundle
management representation
1
1
1..n
1
<<interface>> Constants
<<class>> Bundle Event
Framework Impl
<<class>> Framework Event
1
1
<<interface>> Synchr.Bundle Listener
<<interface>> Bundle Listener
0..n
0..n
<<interface>> Framework Listener

Figure 2: Class diagram org.osgi.framework Life Cycle Layer (from OSGi specification) [1]

- INSTALLED: the bundle is installed, but not yet ready to use. That only means it is available on the platform.
- RESOLVED: the installed bundle has been *resolved*, that means it's dependencies have been resolved. In this state, classes the bundle exports are available for use by other bundles.
- STARTING: the bundle's activator start method has been called, but has not yet returned.
- ACTIVE: the bundle's activator start method has returned.

- STOPPING: the bundle's activator stop method has returned.

Each bundle can provide a reference to an *Activator*, which is the entry point for the bundle. The *BundleActivator* interface offers two methods: start() and stop(), which are hooks to the bundle lifecycle.

1.2 Motivation

In the original OSGi specification, it appears that malicious bundles, or buggy ones, can be deployed on the platform and that the isolation behavior proposed by the specification lacks a certain amount of features such as thread stopping, state record...

Previous work include attempts at automatically detect malicious bundles and stop them. [5]. However, the OSGi specification does not permit to suspend a bundle's activity and resume it later. For example, if a bundle consumes all the available CPU, it could be useful to suspend it while other time-constrained bundles are executing. Currently, the only solution is to stop the bundle, and restart it later, which implies some problems.

1.3 Various approaches to the problem

Low-level hard constraints The I-JVM[5] virtual machine is a Java virtual machine based on the VmKit project [3]. The VmKit project aims at sharing the development effort on Java and CLI virtual machines, in order to ease the development of virtual machines experimentations. This project is itself based on the LLVM project(Low Level Virtual Machine)[6]. The I-JVM proposal relies on bundle isolation to provide low-level security constraint and resource accounting, and is designed for OSGi. The bundle isolation concept, applied at OSGi, consists at isolating the bundles execution in order prevent a malicous bundle from interfering with others. The OSGi specification imposes the use of a ClassLoader based isolation, which is not sufficient. Indeed, these implementation does not allow ressources accounting and management, neither static variables integrity. [8, 9]. Another considered approach to implement such an isolation in OSGi is the Isolate concept. An Isolated is an execution entity, and each Thread posseses a reference towards its Isolate, which provides it a totally private and isolated execution environment. However, this solution is seen as inapproppriate for OSGi, since bundles are not task but components, which have strong interaction. Inter-isolates call are generally expensive, and prevent any use of isolates as a support for bundle isolation[4]. I-JVM proposes to solve this inter-bundles calls problem. It relies on thread migration, allowing each thread to be executed in the called method's corresponding isolate. An isolate is associated to each bundle, so that it has it's private environment. I-JVM also offers a resource accounting feature: the resources used by each bundle are accounted, thus allowing to detect hazardous behaviours. This study's interest in I-JVM is this bundle isolation mechanism: controlling the isolate should allow to suspend an isolate, copying its whole state, waiting to resume it later. Unfortunately, after several experimentation, the I-JVM project is not mature enough to be used right now. However, one can expect great developments from this virtual machine, and the solution proposed in this study acknowledges it's specificity. Therefore, section 4.2 will explain how the following proposal could be integrated with I-JVM.

Using the Java Threading API The suspend/resume process can be achieved through the Java Threading API.

This API allows concurrent programming in the Java language, and is based upon the concept of *thread*. A thread executes independently of other call sequences, and is allowed to share system ressources or objects created in the same program. The java.lang.Thread object is the representation of such a thread, and offers an API to execute code in a thread.

To execute some code, a Thread must be provided with a Runnable object, which contains the actual code needed to be run within the thread. The java.lang.Runnable interface defines only one method, run(), in which the code is encapsulated. This Runnable object is provided during the Thread construction.

When the Thread start() method is invoked, the run method is initiated as an independent thread.

Originally, the Thread API offered a *stop*, a *suspend* and a *resume* method, which respectivily paused the Thread execution and resumed it later. Those methods have been deprecated because it raised more problems than were solved [7]. Specifically, the result of their execution was impredictable, since the program could be suspended or stopped in an inconsistent state. Developers were then given back the responsibility of achieving their own suspend and resume behavior, forcing them to understand the problems behind these mechanism.

One common solution to implement a suspend behaviour using the Java Thread API consists in using the wait() and notify() mechanism.

This high-level approach to suspend an executing thread and is the one that is being used in this work.

2 Proposal for a suspend/resume architecture

We propose to extend the OSGi specification to provide a suspend/resume mechanism. Unlike the approach presented in section 1.3, this one supposes a trust relationship between the platform administrator and the bundle developer, since it relies on contract programming. That means it is the bundle developer responsibility (and his effort) to ensure his bundle has the proper behaviour when it needs to be suspended.

2.1 Extending the OSGi specification

We propose to add new states in the OSGi bundle life-cycle, and corresponding hooks for the bundle developer who would like to implement these.

The life-cycle now contains a new state : *SUSPENDED*, as shown in figure 3

As in the original OSGi specification, the bundle implementation is responsible for achieving the proper behaviour, and the BundleActivator must implement hooks which will allow

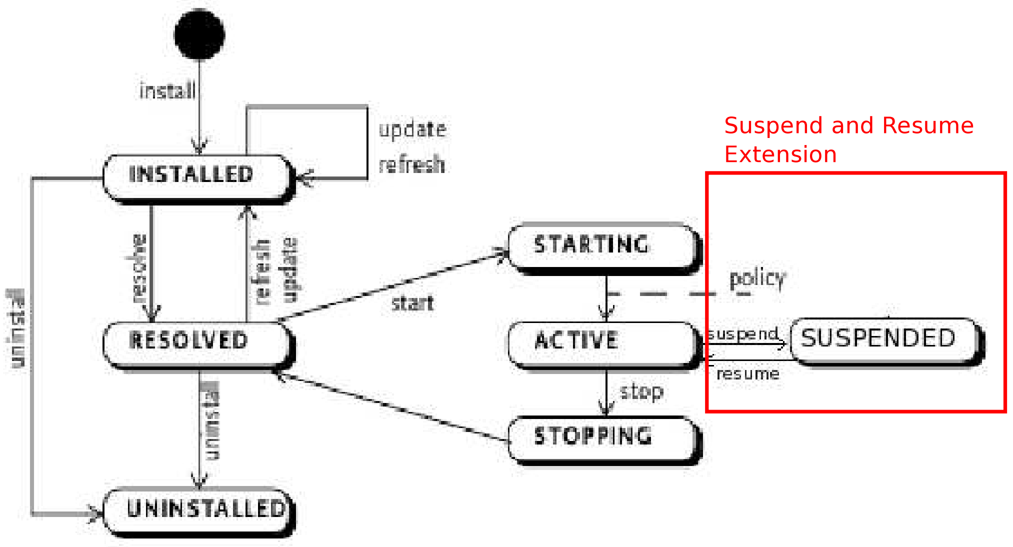

Figure 3: Bundle life-cycle including a susended state

it to react to the states transition from *ACTIVE* to *SUSPENDED* and from *SUSPENDE* to *ACTIVE*. Presumably, one common behavior to implement in reaction to those events will be to suspend the bundle's threads and unregister services during the suspension.

These modifications impacts two classes of the core OSGi specification: the Bundle and the BundleActivator. As shown in figure 4, the Bundle class provides to more methods, *suspend* and *resume*, which will initiate the respective transitions between the *ACTIVE* and *SUSPENDED* states. The BundleActivator now provides two more hooks to react to those transitions: the *suspend* and *resume* methods.

2.2 Problems raised with this extension

When developing for the OSGi platform, one common practice consists in performing the bundle initialization in the BundleActivator start method, which is the entry point for a bundle contributing executable code that needs to be run when the bundle is launched. Any other executed code should be run in a separate Thread launched from the *start()* method, as recommended in [2].

When a developer has to use threads on the OSGi platform, the proposed extension implies that it has to manage the suspension of all the started threads by himself. This can cause difficulties to the average developer, since he is responsible for the bookkeeping of every started thread, and for implementing a suspension mechanism for each one of these.

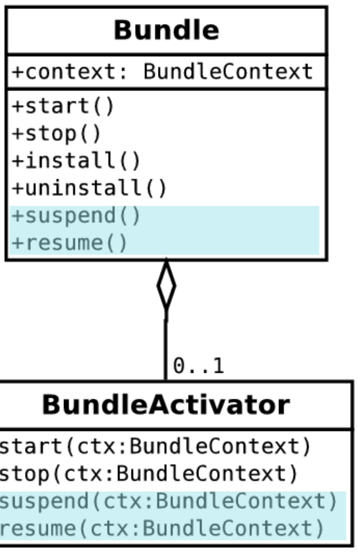

Figure 4: Extension of the OSGi specification (in color)

Therefore, we additionnaly proposes an architecture to which most of this thread management work can be delegated.

This architecture must meet the following goals:

- Allow to suspend all threads attached to a bundle.
- Centralize thread management.
- Take most of the thread suspension mechanism out of the bundle developer responsibility.

2.3 Thread Suspension mechanism used

As described in section 1.3, an efficient mechanism to achieve thread suspension in Java consists in using the wait/notify mechanism.

The *wait* method is part of the java.lang.Object class. When an object *wait* method is invoked, it causes the executing thread to be placed on a wait set. Its execution is paused. When the *notify* method is invoked, a random thread is waken up among the set of waiting threads associated with this object.

This mechanism is often used as a way to achieve a suspend behavior: the code runs in a waiting loop while its awakening condition is not met. Once the condition has been met, the notify method is called and the execution can resume.

To generalize this approach, we proposes to extend the *Thread / Runnable* couple to offer them to the developer with the *SecureThread / SuspendableRunnable* couple.

The goal of this *SuspendableThread* is to be easily suspended during it's execution. The suspension itself is achieved with an asynchronous message, and has been inspired by the "best practices" for implementing thread cancellation in Java ([7]).

Unlike it's parent class, the *SecureThread* runs code from a *SuspendableRunnable* instead of a standard *java.lang.Runnable*.

The *SuspendableRunnable* abstract class implements the Runnable interface to provide this mechanism. It offers a flag, *shouldSuspend*, indicating that the code should suspend itself as soon as possible. Along with this flag, a *suspendIfNeeded* method is available, which performs the actual suspension. This method is very simple, it waits until the *shouldSuspend* flag has been set back to false.

Along with this *SuspendableRunnable* comes an extension to the *java.lang.Thread* class, which allows management of such a runnable.

It offers the methods to control suspension and resumation of the *SuspendableRunnable*.

The detailed process, described on figure 5, is the following:

1. The client calls the *secureSuspend()* method of the thread it wishes to suspend.

2. The thread,in turn, sets the *souldSuspend* flag on its Runnable.

3. When the runned code calls *suspendIfNeeded* (which should be done each time the Thread is in a state in which it could safely be suspended), it checks the *shouldSuspend* flag. Since it has been set, the Thread waits (using *java.lang.Object* standard method) until the flag has been set to false.

4. When the clients calls the *secureResume()* method of the *SecureThread* class, it sets the *shouldSuspend* flag to false, and notifies the waiting *SuspendableRunnable*, which will in turn resume its execution, returning from the *shouldSuspend()* method.

With this mechanism, the thread suspension mechanism is mostly taken out of the bundle developer hands, who only has to ensure that the Threads he uses respect the following constraints:

- they are instances of the SecureThread class
- the code runned by the Thread is encapsulated in a SuspendableRunnable
- the code runned by the Thread is regularly marked out by *suspendIfNeeded()* method calls, each time it reaches a point in his execution at which it is acceptable to be suspended.

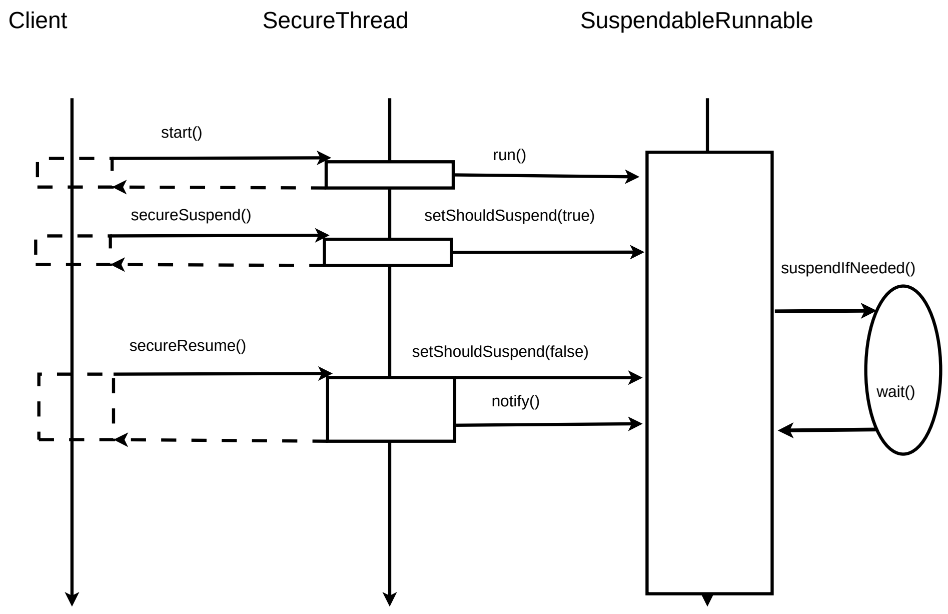

Figure 5: Sequence Diagram for the thread suspension

2.4 Centralization of the Thread management

In order to perform a global thread suspension for a given bundle, our architecture proposes to centralize the SecureThread creation and storage in a singleton object, the *ThreadFactory*. This ThreadFactory holds the responsibility for creating the *SecureThread*s when the bundles need it, and to keep an account of those threads on a per-bundle basis, thus allowing to retrieve (and suspend) all SecureThread created by a bundle.

Each bundle has to use this *ThreadFactory* to obtain new SecureThreads, passing it it's BundleContext to allow the factory to account the created *SecureThread* to this particular bundle.

Internally, the threads are stored in *ThreadGroups*. In the java Thread API, each Thread belongs to a ThreadGroup. ThreadGroups are usally used as a basis to security policies restricting access to thread operations. *ThreadGroup*s are stored as a tree, since each ThreadGroup is nested in the *ThreadGroup* to which the executing thread belongs. The ability to use *ThreadGroup*s to support security policy make it a good choice to store threads on a per-bundle basis: it gives the ability to restrict access on a *ThreadGroup* to members of this ThreadGroup only.

In the ThreadFactory, each Bundle is associated with a ThreadGroup. Then, each newly created SecureThread will be affected to this Thread. The system bundle, which is responsible (among other things) for managing the bundle life-cycle, will start each Bundle in a

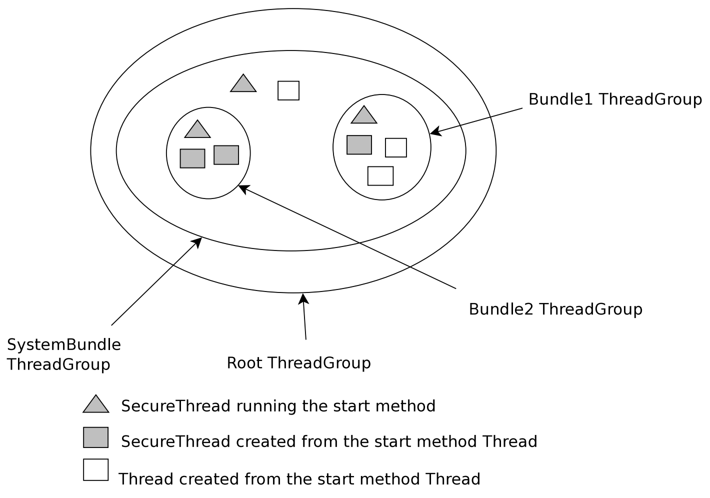
Bundle1 ThreadGroup
Bundle2 ThreadGroup
SystemBundle ThreadGroup
Root ThreadGroup
SecureThread running the start method
SecureThread created from the start method Thread
Thread created from the start method Thread

Figure 6: Association between bundles and ThreadGroups

dedicated SecureThread. The ThreadFactory will thus create a new ThreadGroup for this bundle as soon as it has been started. Figure 6 shows how the Threads and ThreadGroups are associated to bundles[1].

The whole process for running code within a Secure Thread is the following, as summarized on figure 7 :

1. The client code constructs an instance of its own SuspendableRunnable implementation.
2. The client code asks the ThreadFactory to get a SecureThread, providing it with the SuspendableRunnable instance and its BundleContext.
3. The ThreadFactory constructs the SecureThread with the given SuspendableRunnable, as a member of the ThreadGroup associated to the bundle.
4. The client code starts the thread

When the framework needs to suspend a bundle, it asks the ThreadFactory for the list of all SecureThreads associated with that bundle, and can then suspend them one by one.

[1]It is worth noting that if a bundle creates a standard *java.lang.Thread* independently from the ThreadFactory, this Thread will actually belong to the bundle's ThreadGroup

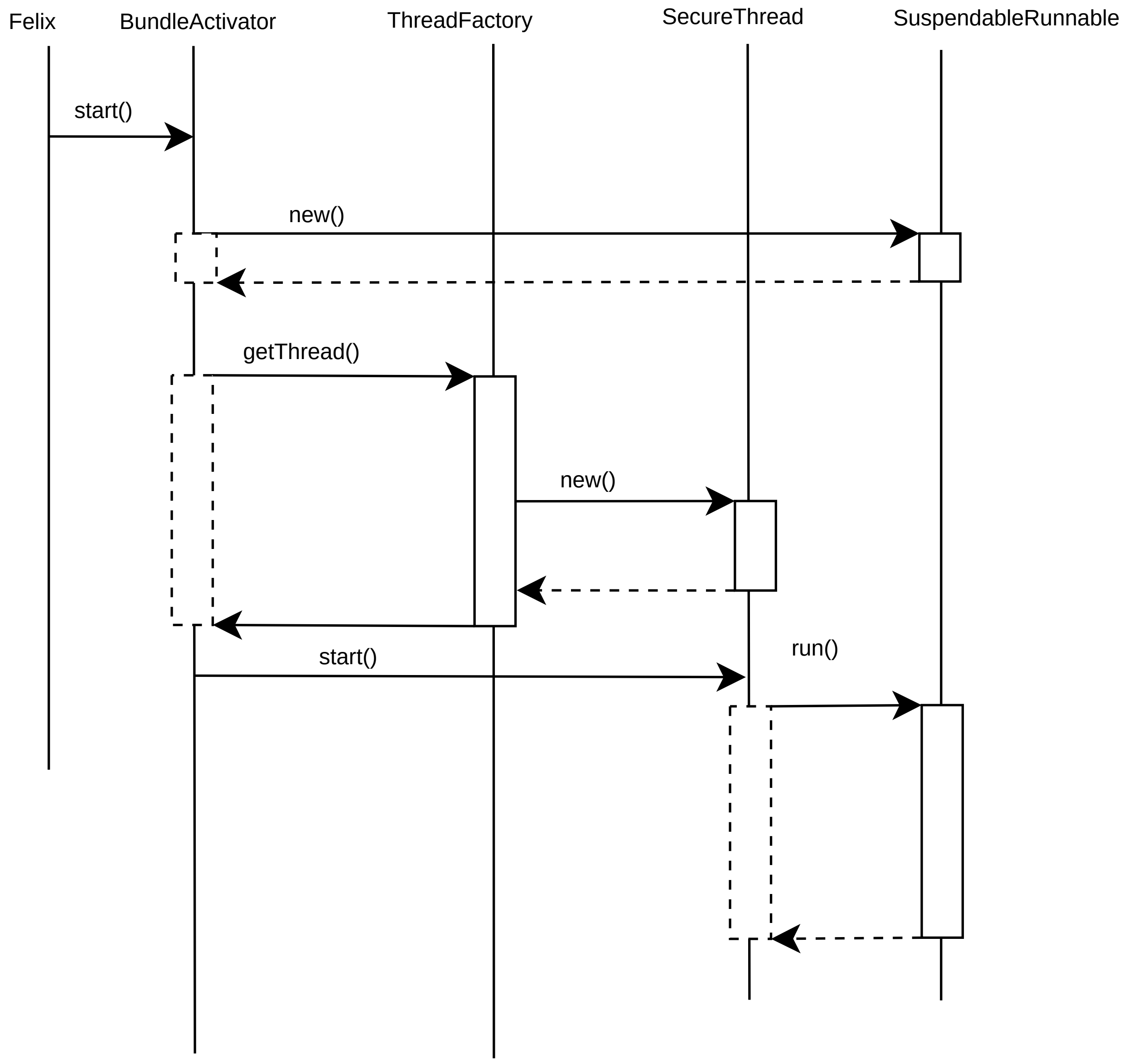

Figure 7: Sequence diagram showing the process of aquiring a suspendable Thread

3 Implementation

3.1 Based on Apache Felix OSGi implementation

Those mechanisms have been implemented, providing a few modifications to the Apache Felix OSGi implementation. It should be easy to implement itin other OSGi implementation, since it does not rely on specificities of the Felix framework.

The Apache Felix framework implements the *Bundle* interface in the *BundleImpl* class, which is subclassed by the *Felix* class. The Felix class implements the *Framework* interface. The role of these different classes is explained on figure 2

The Apache Felix framework has been slightly altered to meet our requirements.

- The BundleImpl class now implements BundleSuspendable instead of Bundle, thus implementing the suspend/resume.
- The Felix class (Framework implementation) has been altered to allow activator's start and stop method to be run in a separate Thread. The reason for such a change is explained in section 2.4.

The *suspend* and *resume* methods implementation in BundleImpl are straightforward: it verifies that the bundle is active, and then call the suspend hook on the Activator, if the Activator is a *BundleSuspendableActivator*.

The whole extension to OSGi Bundle and BundleActivator specification resides in the fr.inria.amazones.sars package, which is then directly imported into the felix core.

3.2 Thread Manager architecture

This implementation comes with a Thread Manager implementation, which implements the principles described in section 2.4.

This ThreadFactory implementation offers several public methods:

public static ThreadFactory getFactory() Returns the singleton object

public synchronized SecureThread getThread(BundleContext caller, SuspendableRunnable target)
: Construct a new SecureThread using the client-defined SuspendableRunnable implementation, and accounts it to the BundleContext associated for the bundle. The threads associated with the BundleContext will be organized in a ThreadGroup.

public synchronized SecureThread getThread(BundleContext caller, SuspendableRunnable target,
: Roughly the same as the previous method, except that a parent BundleContext is provided. This method is used within the felix framework to ensure the threads responsible for starting the bundles are attached to the SystemBundle ThreadGroup[2]

[2]In the particular case of a bundle starting other bundles, this prevents the ThreadGroup associated with the started bundle to be created as a child from the starting bundle, which would lead to an inconsistency in the way the starting bundle should manage the suspension of its "child".

public synchronized SecureThread getAssociatedThreads(BundleContext caller)
: Returns all SecureThread associated with a bundle.

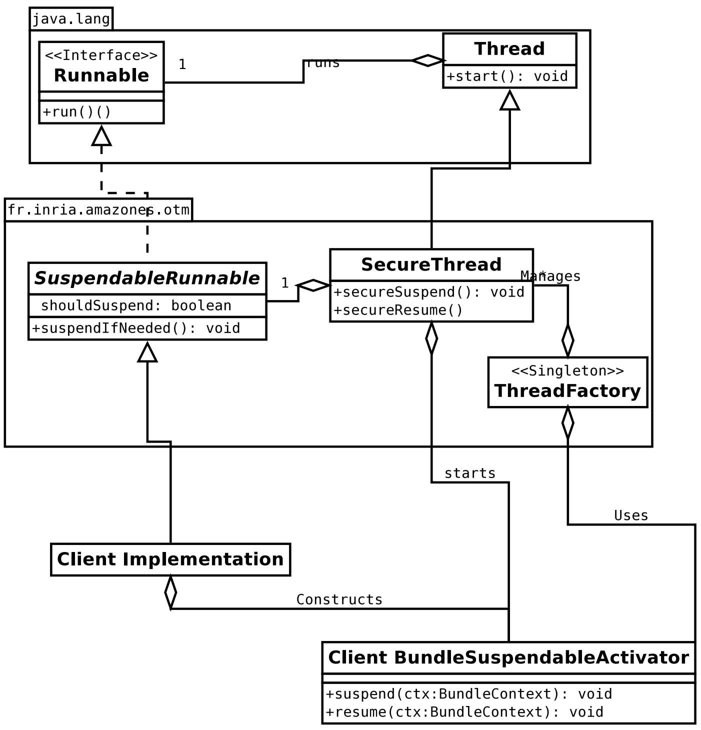

Figure 8: Classes Diagram for the ThreadFactory implementation

The whole ThreadManagement API resides in the fr.inria.amazones.otm package (OSGi Thread Manager), which is then directly imported into the felix core.

3.3 Implementing a suspendable bundle within this architecture

With this implementation, it is very easy for the bundle developer to ensure its thread will actually be suspended when the bunde will be requested to suspend.

The only thing to do is to implement BundleSuspendableActivator instead of BundleActivator, and to use *ThreadFactory.getThread()* instead of creating new *java.lang.Threads*.

A simple example is presented in A.1.

Other common implementations will unregister the published services, or suspend other bundles which would have been started by the suspended one.

3.4 The suspend and resume shell commands implementation

The Apache Felix OSGi implementation contains a simple shell allowing an administrator to install, start, stop or uninstall bundles. This section will describe how a suspend and a resume commands have been implemented within this architecture.

In Apache Felix, the Shell is a bundle (installed by default), which contains several commands. Those commands are registered as Services.

Our implementation is a bundle containing the suspend and resume shell commands implementation. This bundle offers an activator which register those commands as a service. This service is then used by the shell bundle to execute the desired commands as they are typed in the command line interface.

The SuspendCommand and ResumeCommand classes implements the Command interface. This interface defines, among other things, an execute() method. In both cases, this method expects a bundle ID argument. The command fetches the corresponding bundle from the BundleContext, and executes the suspend (respectively resume) method of this bundle.

4 Further work

4.1 Improving the specification

The current implementation already offers a usable suspend/resume architecture. However, those specifications coud be greatly improved by:

- Managing bundles dependencies. In the current proposal, bundle dependencies management is left to bundle developer responsibility. This could be improved by automatically suspending bundles depending on suspended bundles.
- Turning the *SuspendableRunnable* abstract class to an interface. Or even better, dropping the need for such a class and allow the developer to use the standard *java.lang.Runnable* interface.
- Managing the Bundle stop operation the same way as the suspend.

4.2 Future integration with I-JVM

As mentioned in [5], an OSGi implementation must meet a few requirements to be managed by I-JVM, most of them being already met by Felix. Our implementation enforces the stated condition that *The start and stop method from the BundleActivator should be started in separate Thread*.

This implementation could benefit from I-JVM. Moreover, I-JVM allocates isolates per bundles, and runs the threads in thes isolates. Our proposal should indirectly map to the I-JVM thread management, by the mean of ThreadGroup.

A Appendices

A.1 Suspendable Bundle Example

```
package evaluation;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.Bundle;
import org.osgi.framework.BundleContext;
import fr.inria.amazones.sars.BundleSuspendableActivator;

import fr.inria.amazones.otm.SuspendableRunnable;
import fr.inria.amazones.otm.SecureThread;
import fr.inria.amazones.otm.ThreadFactory;

public class EvaluationActivator implements BundleSuspendableActivator{
  private int counter = 0;
  boolean hasToStop = false;
  private BundleContext ctx;
  public  void start(BundleContext ctx){
    this.ctx = ctx;
    createCounter("MyCounter␣");
  }
  public void stop(BundleContext ctx){
    hasToStop = true;
  }

  void createCounter(String name){
    Thread th = ThreadFactory.getFactory().getThread(ctx, new CounterRunnable(name));
    th.start();
  }

  public void suspend(BundleContext arg0){
    //Nothing specific to handle, since the thread is managed
  }

  public void resume(BundleContext arg0){
    //Nothing specific to handle, since the thread is managed
  }

  private class CounterRunnable extends  SuspendableRunnable {

    String name = "Counter";

    CounterRunnable(String name){
      this.name = name;
    }

    CounterRunnable(){

    }

    public void run()
    {
      while(true){
        if(hasToStop){
          return;
        }
        suspendIfNeeded();
        synchronized(this){
          System.out.println(name + " : "  + counter++);
          try {
            Thread.sleep(1000);
          } catch (InterruptedException e) {
            e.printStackTrace();
          }
        }
      }
    }

  }
}
```

References

[1] O.S.G. Alliance. OSGi Service Platform Core Specification.

[2] O.S.G. Alliance. About the OSGi Service Platform, technical whitepaper, revision 4.1. *OSGi Alliance, 20 pp*, 2005.

[3] N. Geoffray, G. Thomas, C. Clément, and B. Folliot. A lazy developer approach: Building a jvm with third party software. In *International Conference on Principles and Practice of Programming In Java (PPPJ 2008)* , Modena, Italy, September 2008.

[4] Nicolas Geoffray, Gaël Thomas, Bertil Folliot, and Charles Clément. Towards a new isolation abstraction for osgi. In *IIES '08: Proceedings of the 1st workshop on Isolation and integration in embedded systems*, pages 41–45, New York, NY, USA, 2008. ACM.

[5] Nicolas Geoffray, Gaël Thomas, Gilles Muller, Pierre Parrend, Stéphane Frénot, and Bertil Folliot. I-JVM: a Java Virtual Machine for Component Isolation in OSGi. Research Report RR-6801, INRIA, 2009.

[6] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[7] Doug Lea. *Concurrent Programming in Java. Second Edition: Design Principles and Patterns.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[8] Pierre Parrend and Stéphane Frénot. Classification of component vulnerabilities in java service oriented programming (sop) platforms. In *CBSE '08: Proceedings of the 11th International Symposium on Component-Based Software Engineering*, pages 80–96, Berlin, Heidelberg, 2008. Springer-Verlag.

[9] Pierre Parrend and Stphane Frnot. Java components vulnerabilities - an experimental classification targeted at the osgi platform. Research Report 6231, INRIA, 06 2007.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l’Europe - 38334 Montbonnot Saint-Ismier (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-0803