Native OSGi
Modular Software Development in a Native World

Alexander Broekhuis
alexander.broekhuis@luminis.eu

Sascha Zelzer
s.zelzer@dkfz-heidelberg.de
Agenda

- Introduction
- Motivation
- History
- Current State
- Native OSGi
- Outlook
Alexander Broekhuis

- Software Developer
  - Started as a Java Engineer
  - Programming C since 2010
- At Luminis since 2008
  - Software House
  - Research and innovation oriented
  - Involved in Open Source (Apache)
- Apache committer since 2010
Sascha Zelzer

- **Software Developer**
  - 15 years of experience with Java, Eclipse, C++
  - Current focus on modular C++ systems
    - For research environments
- **Since 2005 at the German Cancer Research Center (DKFZ)**
  - Large multi-disciplinary research environment
  - Mainly cancer research and medical imaging
  - Long open-source history within the department
OSGi is:

• component based framework
• allows dynamic assembly of components
• Java, so independent of operating system

OSGi technology is the dynamic module system for Java™

OSGi technology is Universal Middleware.

OSGi technology provides a service-oriented, component-based environment for developers and offers standardized ways to manage the software lifecycle. These capabilities greatly increase the value of a wide range of computers and devices that use the Java™ platform.
OSGi
public class Activator implements BundleActivator {
    public void start(BundleContext context) {
        context.registerService(Store.class.getName(), null, new StoreImpl());
    }
    public void stop(BundleContext context) {
    }
}
Benefits

• Making products with many variations
• Improving quality through re-use
• Speed: time to market
"The Native OSGi project is a collaborative effort to write, test, and implement the Java OSGi specifications in C and C++ with a focus on interoperability between C, C++ and Java"
Motivation - Why?

- C and C++ are NOT obsolete
  - C++11 is a big step forward

- Traditional Application Domains
  - Embedded Devices
  - Medical Imaging
  - Sensor Networks

- Lightweight Native Module System
  - Benefits native developers
C/C++ Modularization

• Examples
  - CORBA and CCM: Portable, heavy-weight
  - Service Component Architecture (SCA)

• Problems
  - Find a C/C++ implementation with an appropriate license
  - Scope can be too broad/overwhelming
Benefits

• Mature API
  
  • OSGi is around since 1999

• Core Specifications are small

• Enables hybrid Java/C/C++ solutions
  
  • as an alternative to JNI

• Eases migration
  
  • From Native to Java

• Embedded Performance
Challenges

• Dynamic Module Layer
  • Code Sharing
  • Linking and Versioning

• Different Platforms
  • Posix/Win32/...
• **RFP-89**
  - Proposed to the mailing in 2007
  - Since then remained silent
  - Ongoing (slow) effort to pick up again

• **Focused on**
  - Supporting different languages in OSGi
  - Supporting framework in different languages
  - Languages mentioned:
    - Native (C and C++), .NET (C#), Scripting (Javascript/Actionscript)
History - Universal OSGi

- Completely Different Languages
  - Native, Managed, Scripting
- Limit Scope
  - Focus on C/C++
- Makes it easier to progress
- Keeps Focus on Common Runtime

“Native-OSGi”
Current State

- OSGi-like Implementations
  - CTK Plugin Framework
  - Apache Celix
  - nOSGi
  - Service Oriented Framework (SOF)
Current State

- Small Communities
- No Interoperability
  - Different bundle format
  - API differences
  - Wiring solved differently
    - Module layer
Developed at the German Cancer Research Center (DKFZ)

Largest biomedical research institute in Germany

Founded in 1964, ~2200 employees
CTK Plugin Framework

• Part of “Common Toolkit”
  • Large international initiative (medical imaging)
• C++ API is very close to the OSGi Specification
• Provides Implementations
  • Log
  • Configuration Admin
  • Metatype
  • Event Admin
• Runs on: Windows, Linux, MacOS
CTK Plugin Framework

Powers an “Eclipse RCP”-like C++ platform
Apache Celix

- Development started at Thales Netherlands
- Open Sourced / Donated by Luminis to Apache
- Embedded distributed systems
- Dynamic (Re)Configuration
- Used as middleware in large research project
Apache Celix

- Implemented in C
  - API close to the specification
  - Adapted to Non-Object Oriented use
- Donated to the Apache Incubator
- Provides
  - Log Service
  - Devices Access
  - Shell
  - Remote Service Admin
  - Deployment Admin
nOSGi

- Research project at University of Ulm
  - Steffen Kächele
- Very lightweight and fast implementation (only requires c++ runtime and unzip)
- Runs on POSIX systems
- Features
  - Wiring of shared objects for code-sharing
  - Service registry with filters
  - Supports source bundles (compiled at runtime)
  - Comes with a Shell implementation
Service Oriented Framework (SOF)

• Mature open-source project (BSD)
  • Matthias Grosam

• Shared libraries model bundles

• Runs on Windows and Linux

• Features
  • Service registry, trackers and listeners
  • Provides a command shell
  • Remoting capabilities (using CORBA)
    • Remote services and service listeners
    • Command shell for each process
Specification

- Members
- Goal
- Bundle Format
- Module Layer
- Life Cycle Layer
- Service Layer
- C and C++ Interoperability
Members

- CTK Plugin Framework
- Apache Celix
- nOSGi

Initial/startup meeting took place in Hengelo in May this year
Goal

- Follow OSGi Specification
- Allow Interoperability
  - Bundles
  - Remote Services
- Seamless C and C++ Interoperability
  - E.g. provide a C service, consume via C++ interface
Goal

- Grow Communities
- Combine where possible
- Channel efforts
- Write Open Source reference
  - All feedback is welcome!
Realization

- JAR ➔ ZIP
  - Same Format, Different Layout

- Packages equal Shared Libraries

- Class Loader replaced by Dynamic Linker
Bundle Format

- Like Java Archives (JAR)
- Using ZIP format
- Bundle Manifest
  - .cmf vs .mf
- Headers
- Optionals
- Libraries
- Resources

Layout:
- META-INFO/
  - MANIFEST.CMF
- OSGI-OPT
  - share/
  - include/
  - src/
- lib/
- resources/
Module Layer

• Shared Libraries model Java Packages
  • Allows Code Sharing
  • Multiple Libraries per Bundle
  • Symbols must be exported explicitly
    • Additional visibility control
  • Symbol Searching Handled by Linker

• Meta-data
  • Import/Export Headers
  • Execution Environment for Additional Requirements
“Package” Wiring

- Mechanism from nOSGi
- Library Dependencies Patched at Runtime
  - To match with available exports
- Allows Multiple Versions
  - Of the same package
- Allows Bundle Updates
“Package” Wiring

**Diagram Description**

- **A**
  - x 2.4.0
  - a
- **B**
  - x 2.5.0
  - b
- **X**
  - x-2.4.0
  - x-2.5.0

**Key Elements**

- **linker**: The connection between packages.
- **bundle**: A collection of packages.
- **private package**: A package that is not intended to be shared.
- **export**: A package that is intended to be shared.
- **import**: A package that is used by another package.
- **patched import**: A package that is updated with a newer version.
- **version**: The version of the package.
- **wire**: A connection between packages.
- **uses**: A relationship between packages.
- **bundle name**: The name assigned to a bundle.

**Example Diagrams**

1. **Diagram 1**
   - **A**
     - x 2.4.0
     - a
   - **B**
     - x 2.5.0
     - b
   - **X**
     - x-2.4.0
     - x-2.5.0

2. **Diagram 2**
   - **A**
     - x 2.8.1
     - a
   - **B**
     - x 2.8.1
     - b
   - **X**
     - x-2.8.1
Life Cycle Layer

- Follows Specification
- Resolves Dependencies using Manifest
- Bundle Activator API
  - Start Activator
  - Stop Activator
- Native Specific
  - Create Activator
  - Destroy Activator
Service Layer

• Native API Close to the Specification
  • Especially C++
  • C API is adapted to Non-Object Oriented use

• Requirements for C++
  • Be Type-Safe
    • Avoid exposing void* where possible
  • Do not require a Service base class
  • Allow multiple inheritance of Service interfaces
Service Layer

- Requirements for C
  - Use Struct with Function pointers for Services
  - Components are represented as void*
  - Return value only used for error codes
    - Return values via arguments
C / C++ Interoperability

- Native-OSGi
  - Provides both C and C++ headers
  - Provide thin bi-directional wrapping
- Service Interfaces
  - Should provide C and C++
    - C++ Services implemented using Interfaces
    - C Services implemented using Structs and Function Pointers
C / C++ Interoperability

- **Service Interfaces**
  - Provide bindings for C -> C++ and C++ -> C
  - IDL for Service description and code generation

- **Service Provider**
  - Implement either the C or C++ header

- **Service Consumer**
  - Use either the C or C++ API
C / C++ Interoperability

- C API
- C++ API
- C++ to C Wrapper
- C to C++ Wrapper
- Headers & Utilities
- Native OSGi C Framework Impl

- C Bundle
- C++ Bundle
- C++ Bundle
- C Bundle
greeting_service.h

typedef struct greeting *greeting_t;
typedef struct greeting_service *greeting_service_t;

#ifdef __cplusplus
extern "C"
#endif

struct greeting_service {
    greeting_t instance;
    void (*greeting_sayHello)(greeting_t instance);
};

IGreetingService.h

struct IGreetingService {
    virtual ~IGreetingService();
    virtual void sayHello() const = 0;
};

OSGI_DECLARE_SERVICE_INTERFACE(IGreetingService, IGreetingService_NAME)
greeting_impl.c

```
#include "greeting_service.h"
struct greeting {
  char *name;
};
void greeting_sayHello(greeting_t instance) {
  printf("Greetings from %s\n", instance->name);
}
void register_services() {
  BUNDLE_CONTEXT context = ...
  greeting_service_t greeter = malloc(...);
  greeter->instance = malloc(...);
  greeter->instance->name = "C greeter";
  greeter->greeting_sayHello = greeting_sayHello;
  bundleContext_registerService(context,
      IGreetingService_NAME, greeter, NULL, NULL);
}
```

GreetingImpl.cpp

```
#include <IGreetingService.h>

struct CppGreeter : public IGreetingService {
  std::string name;
  void sayHello() const {
    std::cout << "Greetings from " << name << std::endl;
  }
};

void register_services() {
  osgi::BundleContext* context = ...
  CppGreeter* greeter = new CppGreeter;
  greeter->name = "C++ greeter";
  context->registerService<IGreetingService>(greeter,
      osgi::ServiceProperties());
}
consumer_impl.c

BUNDLE_CONTEXT context = ...
SERVICE_REFERENCE serviceRef = NULL;

bundleContext_getServiceReference(context, IGreetingService_NAME, &serviceRef);

void* serviceHandle = NULL;
bundleContext_getService(context, serviceRef, &serviceHandle)

greeting_service_t service = (greeting_service_t)serviceHandle;
service->greeting_sayHello(service->instance);

ConsumerImpl.cpp

osgi::BundleContext* context = ...

typedef osgi::ServiceReference<IGreetingService> ServiceReferenceReferenceType;

ServiceReferenceReferenceType greetingRef = context->getServiceReference<IGreetingService>();
IGreetingService* greetingService = context->getService(greetingRef);
greetingService->sayHello();
Outlook

- Write Specification
- Test ideas/solutions
- As part of the OSGi Alliance
- Define Reference Implementation
- Look into Compendium Services
  - Remote Service as alternative to JNI
  - Adapt other Services to Native-OSGi
- Community!
Resources

• Native-OSGi: www.nativeosgi.org

• Apache Celix:
  • incubator.apache.org/celix

• CTK Plugin Framework:
  • www.commontk.org/index.php/Documentation/Plugin_Framework

• nOSGi:
  • www.uni-ulm.de/in/vs/proj/nosgi/

• SOF:
  • sof.tiddlyspot.com/