Universal Declarative Services
A component framework for C++ and a model-driven solution to polyglot components

Simon Chemouil

1Global Vision Systems

OSGi Community Event, 2012
Who am I?

- Free Software enthusiast, functional programmer, interested in model-driven development, modularity, re-usability, maintainability...
- “How to never have to write that piece of software again?”
- Technical Architect at Global Vision Systems, Toulouse
  - We create data visualization software for large industries (2D, 3D, tables, etc).
  - We do both Java and C++
    - Java using OSGi (Felix, iPojio) for server and client side (Eclipse RCP and Android)
    - C++ to keep a fast 3D engine-agnostic manipulation layer
Outline

1. Motivation
   - The case for modularity
   - Service Components and C++

2. Implementing Declarative Services for C++
   - Challenges in C++
   - Defining components

3. Going Universal
   - Simpler Component Definitions
   - What’s next?

4. Summary
   - Conclusion
Did you miss “Native-OSGi, Modular Software Development in a Native world” by Alexander Broekhuis and Sascha Zelzer?

“Universal OSGi”

- an RFP and a blog post by Peter Kriens in 2007.

- Apply OSGi techniques to native (C/C++) development.
An OSGi Architect Walks in a C++ Bar...

- Our C++ project going through re-architecture / refactoring
  - Custom-made plugin system, limited.
    - Plugins can’t use each other

- C/C++ OSGi ports:
  - Celix (Alexander), CppMicroServices (Sascha), SOF, CTK, nOSGi, ...  

- Yet no service component framework such as Declarative Services
Why a Service Component Framework?

- Services: powerful primitives, unpractical programming:
  - Lack of structure
  - Dynamic

- Service Components:
  - Structure the services
  - A nice semantic model to reason about dynamism

Why is there no such framework for an OSGi C++ port:

1. Technical challenges
2. It was just not done yet!
Why Declarative Services?

- Many existing component frameworks for C++
  - Just as there were some for Java before DS
  - No modularity

- Simple and matured component model
  - A few years of personal experience

- Works with POJOs (or POCOs!)
  - Components do not depend on the framework
  - Makes them testable and easily reusable!
Does it make sense for C++?

- We use C++ for performance:
  - Performance hit with component frameworks?

- We need extensibility and customization...
  - Do we need modularity?

- Why use different architecture between Java and C++?
  - What if we could talk about components, whatever the implementation language?
Getting started

- Native OSGi implementation:
  - CppMicroServices is light, works with any shared library
  - Inspired by PojoSR
  - For C++!

- Bundles are called Modules:
  - Native shared libraries: not archives.

- Moving to NativeOSGi later?
  - Easy! Components are framework-agnostic!
Ingredients in Declarative Services

- Components are described in XML files in the bundle
  - Service callbacks
    - Defined in description
    - Service arrival/departure
    - Policy, Cardinality, Target
  - Component callbacks
    - Activation, modification, deactivation
XML descriptions...

- We have native shared libraries, not JARs!

- Instead, we register a ComponentDescriptor instance to the ComponentManager service.

- The ComponentManager service is provided by the core Ds4Cpp framework.
Component Descriptions in C++

Definition

```cpp
class ComponentDescriptor {
public:
    const string componentId;
    const string implSharedObject;
    const vector<string> *providedServices;
    vector<ComponentReference> *references;
    const bool immediate;
    const bool autoEnable;
};
```
Component Descriptions in C++

Let’s see how it looks like in practice...
Extensible Hello World!

Simon Chemouil  Universal Declarative Services
Definition

std::vector<ComponentReference>* references = ...;
references->push_back(
    ComponentReference("greetdemo::GreetProvider", "greetdemo::GreetProvider",
    std::string(), ComponentReference::DYNAMIC,
    ComponentReference::MULTIPLE,
    ComponentReference::OPTIONAL_REF));

std::vector<std::string> services;
services.push_back("greetdemo::GreetManager");

ComponentDescriptor* componentDesc =
    new ds4cpp::ComponentDescriptor(
        "greetdemo::GreetManagerImpl","",
        services, references, true, true);
class GreetManagerImpl : public GreetManager {
public:
    GreetManagerImpl();  // Constructor
    virtual ~GreetManagerImpl();  // Destructor
    void activate();
    void addGreetProvider(GreetProvider *);  // Add a greet provider
    void removeGreetProvider(GreetProvider *);  // Remove a greet provider

    // Service methods.
    string getDefaultTarget(const string &);  // Get default target
    string getGreeting(const string &, const string &);  // Get greeting
    const list<string> getAllGreetings(const string &);  // Get all greetings
    const list<string> getAvailableLanguages();  // Get available languages
};
Service Callbacks in C++ (1/2)

- No reflection/introspection!
  - There are some limited libraries (e.g. Reflex).

- C++ name mangling
  - We can’t guess the binary name

- Instead, we force a convention in the C++ component:
  - set<RefName>, unset<RefName> for single dependencies
  - add<RefName>, remove<RefName> for multiple dependencies
Service Callbacks in C++ (2/2)

- Our tools: dlopen/dlsym/dlclose (and Windows equivalents)
- Against us: C++ name mangling

Solutions?
- Reflection libraries: too limited, not dynamic.
- C++11 attributes (≈ Java annotations): still largely unsupported
- Qt meta-objects? Not bad!
  - Yet need a custom preprocessor
  - Still not modular (work in progress)

How about C wrappers?
- Difficult to write manually, but provide universal ABI!
C Wrapper

Definition

```
extern "C" {
    DS_ABI_EXPORT GreetManagerImplWrapper* __greetdemo__GreetManagerImpl__create() {
        return new GreetManagerImplWrapper;
    }

    DS_ABI_EXPORT void __greetdemo__GreetManagerImpl__activate(
        GreetManagerImplWrapper* object) {
        object->activate();
    }

    DS_ABI_EXPORT void __greetdemo__GreetManagerImpl__add_greetdemo__GreetProvider(
        GreetManagerImplWrapper* object, ::us::Base*service) {
        greetdemo::GreetProvider* lservice = dynamic_cast<greetdemo::GreetProvider*>(service);
        object->addGreetProvider(lservice);
    }

    DS_ABI_EXPORT void __greetdemo__GreetManagerImpl__remove_greetdemo__GreetProvider(
        GreetManagerImplWrapper* object, ::us::Base*service) {
        greetdemo::GreetProvider* lservice = dynamic_cast<greetdemo::GreetProvider*>(service);
        object->removeGreetProvider(lservice);
    } // extern "C"
```
It works! But...

- The proof-of-concept is successful:
  - We have components wired dynamically
    - OSGi’s patterns work nicely (yet a bit more tricky)
  - We support the use cases we need
    - Dynamic arrival, thread-safety planned but unfinished, service removal to do.
- But no one would write that wrapper code!
- Are there solutions to help us here?
Demo Time!
C++ is too complex!

- Automatically generating the wrappers?
  - Where to get the description?
    - Macros?
    - Parsing C++?!

- Xtext
  - Java tooling to define DSLs and compilers
  - Generates an Eclipse editor!
  - Uses EMF as backend, so more can be built upon.
Definition

```java
component greetdemo.GreetManagerImpl {
    provides {
        greetdemo.GreetManager
    }
    references {
        dynamic service greetdemo.GreetProvider [0..n]
    }
}
```
Defining multiple components

```module DSDemo {
  name = "DS Greet Demo"
  version = 1.0.0
  includes {
    greetdemo.EnglishGreetProvider
    greetdemo.FrenchGreetProvider
    greetdemo.GreetManagerImpl
    greetdemo.ConsoleGreeter
  }
}
Generating C++ code
Demo Time!
Improving our ADL

- The ADL we designed maps to Declarative Services’ capabilities
  - What about richer component models?

- Creating new target language back-ends
  - Java/DS, Java/iPojo
  - Making it extensible
Bridging Java and C++

- Using Remote OSGi Services:
  - An idea discussed in NativeOSGi.
  - We expose C++ services where we can generate a Java interface
    - Use JNI/JNA bindings or network.

- Basically, we bridge the two service registries!
  - It does sound like CORBA
Creating UI Tools

- Textual DSLs are nice...
  - But component diagrams are nicer!

- Up-to-date component metadata is extremely useful:
  - Forces high-level design (round-trips between architects and developers)
  - Allows code generation/rapid project bootstrapping
  - Static analysis (code conformance)
  - Can be used for SDK documentation.

- Long term project!
We have been using Ds4Cpp for a few months!

If you want to try it... you can! We just open sourced it (Apache Software License 2.0)

https://github.com/Global-Vision-Systems/Ds4Cpp
(the demo is packaged with it)!

We need testers and eyes to fix my bad C++ :-)

And remove the limitations (talk to me!)
Universal Components Tooling

- The tooling has still some rough edges
  - Enough for our current needs...
  - Some clean-up required.

- We have a roadmap for the improvements.
Questions?

Twitter: @simach

simon.chemouil@global-vision-systems.com