JunC++ion® – Java™/C++ Integration without Pain

Executive Summary

Whether you own a Java API and your customers demand a C++ version, or whether you own a C++ application and need to integrate with a third-party Java API: the integration of code written in Java and C++ can be expensive and fraught with risk.

You probably immediately think “web services” or “enterprise service bus” when you are confronted with such an integration scenario for the first time. These technologies can yield satisfactory results under some circumstances, but they are not a general solution for the integration problem of making calls across the Java/C++ language boundary.

This whitepaper introduces JunC++ion, a Java/C++ integration technology that was designed from the ground up to solve cross-language integration problems. We contrast different integration approaches and provide you with a comparison matrix that you can use to decide which of the many integration approaches is best for you.

Introduction

There are many use cases that require that you integrate components written in Java with software written in C++. The following list contains some problems that we have encountered time and time again:

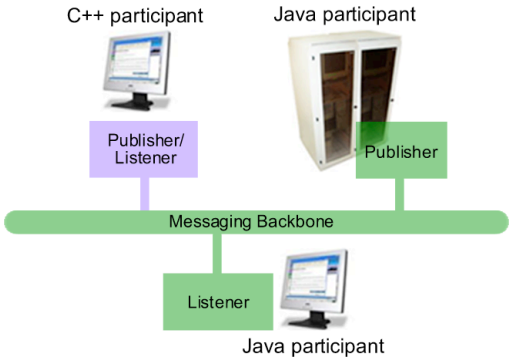
- Publish a C++ version of your Java API.
- Integrate a C++ application with enterprise Java APIs (EJB, JMS, JNDI).
- Use a unique third-party Java library from a C++ application.
- Display existing Java GUIs in a C++ application.

In this paper, we will discuss Java/C++ integration by looking at the second problem category: use of an enterprise Java API, in this case the Java Message Service (JMS). Please understand that by no means is JunC++ion’s integration approach reliant on or limited to JMS; we chose JMS as our example for two reasons:

- It is a real-life use case that is immediately appealing to many people.
- It is a beautiful example of mixing in-process and out-of-process integration technology.

In case you’re not familiar with the Java Message Service: JMS is a specification for performing asynchronous messaging between Java applications. Sun Microsystems maintains the API specification which in turn is implemented by several dozens of independent messaging providers. Every J2EE application server comes bundled with a JMS implementation but you can also use one of many free or commercial stand-alone messaging implementations if all you need is messaging.

JMS is a very popular Java enterprise API and there are many people who have to integrate with it or would like to leverage this technology from their C++ applications. Following the old adage of one picture being worth more than a thousand words, the diagram below describes the integration problem at the architectural level.



Candidates: the Good...

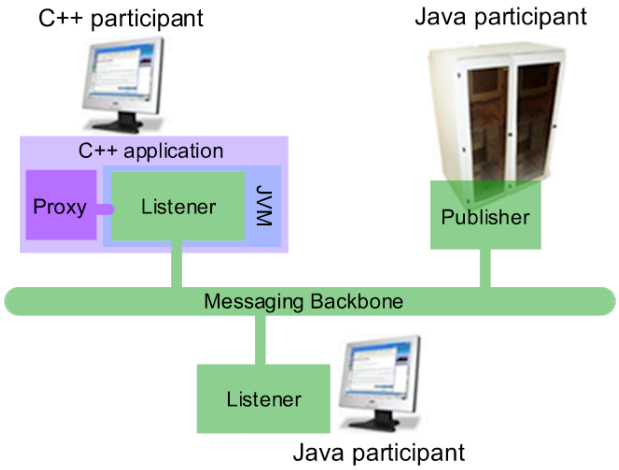
As already mentioned in the executive summary, a modern developer's integration toolbox contains a variety of tools for integrating Java and C++ components. Just to mention the most common ones:

- CORBA
- Web services
- Java Native Interface (JNI)

This last option (JNI) is significantly different from the first two,and, in fact, all other integration approaches. JNI allows you to mix Java and C or C++ within one process. Basically, you can call a JNI C function that results in an action being taken on the Java side within your C or C++ process. Contrast that to all the other integration approaches that have the Java and C++ components running in different processes and have to use some kind of inter-process communications to achieve integration.

Consequently, in-process integration via JNI has some very attractive technical characteristics when compared with all other integration technologies:

- JNI is fast.
  The cost of a JNI call is comparable to the cost of an expensive virtual function call; inter-process calls are usually several orders of magnitude slower. There is also very little data marshaling involved.
- JNI is secure.
  There are no inter-process communications; all calls are within one process, comparable to calling a function that is exported by a shared library. Expensive security measures (both in terms of performance and money) like authentication or encryption are not required.
- JNI is reliable.
  You might cringe as you are reading this statement if you have written JNI code by hand. Your reaction is simply due to JNI's ease-of-use, or better, the lack thereof, not due to any technical shortcomings. When we look at the integration problem, JNI is the only approach that does not introduce additional failure modes into the integrated solution. An error-free JNI-based application will either work (if it is correctly configured) or not (if it is incorrectly configured). The same cannot be said for solutions that rely on inter-process communications or the availability and reachability of server applications.



The diagram above illustrates the architecture when the C++ application uses JNI to bridge the language gap to Java. The C++ application contains a dynamically loaded Java Virtual Machine (JVM) that executes all Java code on behalf of the containing process. This diagram also illustrates a mix of in-process and out-of-process technologies. JNI, an in-process integration technology, is used to bridge the language gap efficiently; JMS, our example use case and by itself an out-of-process integration technology, is used for asynchronous messaging between applications.

We will discuss the significant downsides of using handwritten JNI code later in this whitepaper.

..., the Bad...

How do the other integration technologies stack up? In our opinion, they don't. Whether you're looking at CORBA or at web services, you're looking at technology that would be used outside of its area of core competency. Both technologies were invented to solve the problems associated with distributed systems, possibly distributed across the internet, and that's what they are good at. Bridging a programming lan-

guage gap is a mere side effect of these technologies trying to be useful to as many developers as possible. Using CORBA or web services to solve a “mere” language integration problem is like using a sledge hammer to kill a mosquito: you might succeed after a fashion but you should really assess the collateral damage afterwards...

There are two big drivers behind the use of CORBA or web services for language integration:

1. They might already be used in the project for other purposes.
2. They might be perceived as “sexy” and interesting (at least in the case of web services).

You should not fall into the trap of choosing an inappropriate tool for a job for the sole reason that it looks nice or that it is available! If you do, you’re going to pay a price farther down the track.

..., and the Ugly

The one significant downside of JNI has already been mentioned: it is very hard to use, in fact, it is downright ugly! While it is a beautifully designed API, it really was not meant to be used by the casual software developer. A software tools architect at Sun Microsystems once told us that —in his opinion— JNI should be consumed by a code generator and not by a human being. The C++ snippet below illustrates this point. In this snippet, we attempt to call a Java method that takes a `String`, a `Date`, and a `bool` argument. We assume that the JVM has already been loaded and that the thread has already been attached to the JVM.

```
jclass     clsDate = env->FindClass( "java/util/Date” );
jclass     clsMyType = env->FindClass( "com/mypack/MyType” );
if( clsMyType == NULL )
   exit( 1 ); //handle error in real app
jmethodID  ctor = env->GetMethodID( clsDate, "<init>”, "()V” );
jmethodID  mid = env->GetStaticMethodID( clsMyType, "calculate”,
                 "(Ljava/lang/String;Ljava/util/Date;Z)J” );
if( mid == NULL )
   exit( 1 ); //handle error in real app
jstring    arg1 = env->NewStringUTF( "An input string” );
jobject    arg2 = env->NewObject( clsDate, ctor );
jboolean   arg3 = bArg ? JNI_TRUE : JNI_FALSE;
jlong      result = env->CallStaticLongMethod( clsMyType, mid,
                        arg1, arg2, arg3 );
if( env->ExceptionOccurred() )
   exit( 1 ); //handle error in real app
```

If this were production quality code, this snippet should really contain much more —and much more complex— error handling. It is probably no surprise that production JNI code usually does not contain the required error handling. This is one of the reasons for JNI having a reputation of being unreliable.

Most people who have used JNI successfully have only used it in very limited scenarios, where they implemented a few integration points with relatively simple usage. Using JNI by hand to expose a significant enterprise Java API to C or C++ would be a truly herculean task! Furthermore, JNI is a low-level C API. The creation of easy-to-use C++ proxy types for the Java types of interest is a problem that is not solved by using JNI.

To sum it up: JNI is technically the best solution for integrating Java and C++, but it is too hard to use. What should be done?

The solution

There are several parts to the solution, but they revolve around a few key points:

- The low-level integration must use JNI for performance, security, and reliability reasons.
- JNI must be invisible yet accessible to the developer.
- The developer must have access to all aspects of the underlying Java type through an easy-to-use and “unsurprising” C++ type.
- The developer must be able to easily specify which Java types she wants to use.

These key points lead us to a solution that includes a runtime library that abstracts JNI away as much as possible and to a code

generator that generates C++ proxy types for user-specified Java types.

In the next sections we're going to take a look at some key requirements for the code generator and the runtime library and at some very hard to implement features, like, for example, callbacks.

What you want

It's probably best if we take a Java snippet and demonstrate by example what a corresponding C++ snippet should look like. Staying with our JMS example, we'll start out with piece of boilerplate code that almost every JMS client has to implement: the JNDI lookup of a ConnectionFactory. The following C++ snippet assumes that we're using JBoss as our JMS provider.

```
#include "java_util_pkg.h"
#include "java_lang_pkg.h"
#include "javax_naming_pkg.h"

int main( int argc, char* argv[] )
{
   Hashtable               ht( 5 );
   InitialContext          ictx = null;
   TopicConnectionFactory  tcf = null;

   ht.put(Context::PROVIDER_URL, "jnp://localhost:1099" );
   ht.put(Context::INITIAL_CONTEXT_FACTORY,
          "org.jnp.interfaces.NamingContextFactory" );
   ht.put("java.naming.factory.url.pkgs", "org.jnp.interfaces");

   try
   {
      ictx = InitialContext( ht );
      tcf = (TopicConnectionFactory)ictx.lookup("ConnectionFactory");
   }
   catch( NamingException ne )
   {
      ne.printStacktrace();
   }
}
```

This code could almost be straight out of our JunC++ion JMS example. The only thing that we actually can't do the way it is shown in the above snippet is the boldfaced cast to `TopicConnectionFactory`. This cast will have to be replaced with a special framework method invocation. No matter how hard you try to make one technology pretend to be another: there will always be a few corner cases where the pretense won't work and a little bit of special integration knowledge is required. Leaving this one exception aside, consider how much clearer and more maintainable this C++ fragment is than the preceding JNI fragment. Also consider that this fragment is a fully functional application that on-demand loads a JVM, calls a sequence of Java methods, dereferences a number of Java fields and handles all Java exceptions. It does all that while requiring almost no special integration knowledge and in about the same number of lines as the cryptic, incomplete, and completely unmaintainable JNI snippet that called just one Java method!

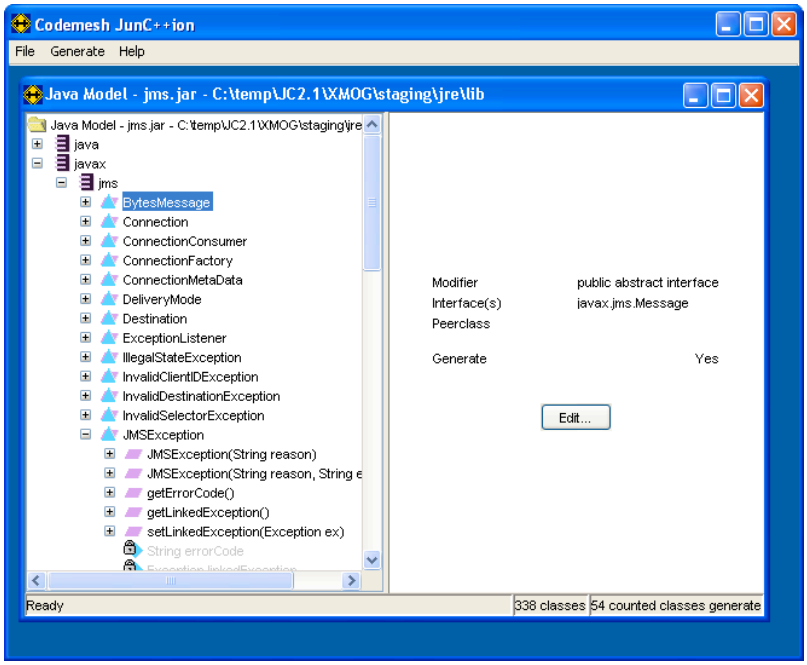
There is no magic involved here, just some very hard work and a lot of experience with C++ compilers, the Java and C++ languages, and the JNI interface that provides the glue between them. Codemesh has by now spent approximately eight man years on developing a solid runtime library and a code generator that takes Java bytecode as input and generates C++ source code. The generated code can be used in your application to call the corresponding Java API. The library and the code generator are of course the components that we identified as the crucial pieces of a good Java/C++ integration technology: a JNI abstraction layer and a way for the developer to specify the Java types that he wishes to use from C++.

The combination of the components creates a technology that allows you to write C++ code that looks and behaves *almost* exactly like the corresponding Java code and requires very little expert knowledge about the integration layer.

Creating the C++ proxy types

You might think that it is relatively easy to wrap a Java type in a C++ type. As an exercise, you

can pick any Java type, manually create a C++ wrapper type and you're right: it's probably not too hard. The relative ease with which you can solve this problem for one Java type might deceive you into thinking that wrapping an interdependent set of Java types in corresponding C++ types is equally easy. Once you get into "set of arbitrary types" issues, the problem becomes vastly more complex, particularly if you want to be able to keep the generated C++ code portable and the set of types that you're generating small. In general, one Java type will depend on many other Java types. There are implemented interfaces and ancestor classes, but there are also types that are used as method arguments or fields. In order to call a method you will need access to the argument and return types used by the method. This causes a combinatorial explosion that may result in huge sets of C++ types being generated. To give you an example: the `java.lang.Object` type directly or indirectly references between 250 and 300 other Java types (depending on the version of Java that you are using). A code generator needs to be smart about type relationships and it needs to be able to determine the developer's intent to reduce the number of generated types.



Then there are issues related to platforms and compilers. Different C++ compilers define different preprocessor macros. What might be a legal identifier on one platform and one compiler might result in compilation errors on another platform or another compiler due to a clash with a predefined macro. The developer needs to be able to override the naming of the C++ elements.

We came to the conclusion that the only way to tackle all these problems is via a code generator that includes a GUI. The code generator GUI allows the developer to set transformation options and visualize and specify the set of Java types which she is interested in. Once the type set is defined, the C++ code can be generated and the generation instructions can be saved. The saved generation instructions can then also be used by a command line version of the code generator during automated builds.

The screenshot below shows a code generator session into which the JMS API has been imported by drag-and-dropping a `jms.jar` file. One major reason for a graphical code generator being a necessity becomes clear by looking at this picture: you can easily browse the types that are to be generated by expanding packages. You can even expand types and see (or change) which fields and methods are marked as generating. If you were to generate at this moment, you would end up with a C++ version of the entire JMS API but you would not yet be able to use it successfully from C++. In order to use JMS, you normally have to first use JNDI (the Java Naming and Directory Interface) to discover the JMS implementation types but the JNDI API would not have been generated. No JNDI type is referenced by the JMS API, so the code generator is not aware that you need JNDI. The code generator GUI allows you to check which types are marked as generating and to add additionally required types to the model. You can persist the model at any point and come back to it at a later point to add or remove further Java types. The persisted model also acts as input to the command line version of the code generator. This part of de-

velopment is usually iterative in nature: you import, generate, start coding, find a missing Java type, reopen the saved model, add the missing type, regenerate, etc. After a few regeneration cycles you will typically have all the types you need to be productive on the C++ side. Some integrators will never or very rarely have to revisit this part of development because the Java APIs they are using change infrequently.

How the C++ code works

Now that you're done with code generation and have a beautiful C++ version of the JMS API and assorted other Java types available, you can really start working on writing application code. You have already seen an application fragment of a JMS client application, but now we will scrutinize the code a little more closely.

Earlier, we made the statement that the C++ application fragment was actually a fully functional application that would load a JVM into the process and start executing Java code. You might have asked yourself how a C++ application would be able to a) find a JVM and b) find the Java classes that are referenced by the C++ code. We glossed over these points because we wanted you to focus on the usability of the proxy API and not on the runtime infrastructure. Let's take a look at the runtime infrastructure now.

The first thing that we're doing in `main()` is the instantiation of a Java `Hashtable` object. When and how did the JVM get loaded that allows this operation to succeed? The answer is that the JVM is on-demand loaded when the C++ constructor for the `Hashtable` proxy type discovers that no JVM has been loaded yet. Because no JVM had been explicitly configured, the runtime library checks for installed JVMs and picks one of them. While this will work on many hosts, it is clearly a haphazard way of

```
xmog_jvm_loader & loader =
        xmog_jvm_loader::get_jvm_loader();

loader.setJvmPath(
        "../jre/bin/server/jvm.dll");
```

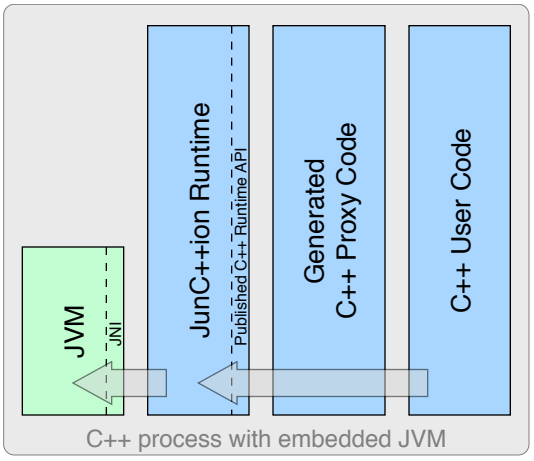
writing and deploying software. To solve this problem, the runtime publishes a complete configuration API that gives you control over all aspects of JVM initialization and configuration. You could, for example, write the above code to use a JVM that is bundled with your application.

In addition to the `setJvmPath()` method, there are methods for specifying the classpath, memory limits, system properties, etc. You have full programmatic control over all aspects of JVM configuration.

```
xmog_jvm_loader::setConfigFile(
                        "myapp.exe.config" );
```

All of these options can also be configured via a configuration file. Simply add the above line as the first line in your application and the `myapp.exe.config` file will be used as the source for JVM configuration information.

There are actually many more configuration options available, but the crucial point has hopefully been made without having to go into more detail: a good integration product needs a powerful and flexible configuration API that allows users to integrate it into their own application's configuration mechanism.



The diagram above illustrates the components that participate at runtime to create the illusion of a pure C++ application, while in reality a significant part of the application is running inside an embedded Java Virtual Machine.

Callbacks and Semantic Usability

All of these configuration options have one purpose: to start up a correctly configured JVM in the C++ process via the JNI Invocation Interface. Once the JVM has been launched, every interaction with it occurs through a JNI call. We said earlier that JNI is the best technology for this type of integration, but even JNI does not offer everything that we might be looking for. One particularly significant shortcoming is in the area of callbacks. A general purpose Java/C++ integration product has to allow a C++ developer to perform most of his work in C++ without having to resort to Java. Many Java APIs rely on callbacks or asynchronous notification mechanisms; just take Swing `ActionListeners` or JMS' `MessageListeners` as examples. A Java developer simply implements the callback interface, registers an instance of the implementation type with the event source and starts receiving event notifications or messages. We would like a C++ developer to be able to do the same thing with similar ease.

This requirement goes beyond the mere creation of proxy types: while it is easy to create a C++ proxy type for the `MessageListener` interface, it is very hard to create a C++ proxy type for the `MessageListener` interface that can be implemented in C++ and then registered with a Java queue or topic as a valid `MessageReceiver`!

This requirement is a perfect example of what we call the *semantic usability* of a proxy type. A proxy type should not only look like its original, it should have the same usability. In fact, the look is far less important than the usability. If you study JunC++ion-generated C++ proxy types in detail, you will find a lot of framework methods whose purpose is hard to understand. These methods are there to enable certain semantic use cases and you'll never use them explicitly; instead, the C++ compiler will invoke them when you use a proxy type in a certain way. These methods are the "silent enablers" of many features.

But let's return to our callback use case: callbacks are usually one of the hardest requirements for any integration technology; JNI is no exception. While JNI allows C programmers to call a Java method and Java programmers to call a C method, there is no easy way in JNI to say the following: take this array of C function pointers as the native implementations of a Java interface and give me an object of a type that implements the interface using these function pointers. To give a C++ developer the same powers that a Java developer has by default requires an elaborate code generation scheme: we not only have to generate C++ proxy types but also Java types which implement the callback interfaces. With such a scheme in place, a C++ developer can simply extend the special callback proxy type and override the interface proxy methods.

A checklist for technology decision making

The following table provides you with a checklist for your choice of integration technology. We contrast web services (WS), CORBA, and JunC++ion according to a variety of criteria.

We intentionally do not take hand-written JNI code into account here. The only project in which handwritten JNI should be used is a tiny point-integration problem. Such a problem is neither the focus of our JunC++ion product nor of this paper.

Some criteria don't lend themselves to a simplistic answer: whether a particular technology is suitable or not may depend on the combination of several criteria and not just on one independent factor.

Legend

✔	strong match
✔	possible match
✘	unlikely match
✘	not a match
✘	total mismatch

Criterion		Explanation	WS	CORBA	JunC++ion
API Granularity	Coarse	A component API like a CORBA or a service specification, designed for client/server usage.	✔	✔	✔
	Fine	A set of types, usually designed for local in-process usage and not for client/server usage.	✘	✘	✔
API Size	Large	Many or complex types.	✘	✘	✔
	Small	Few and simple types.	✔	✔	✔
API Origin	Pre-existing	The API with which we need to integrate is already written in Java or C++. It might even be a third-party API that we don't own.	✘/✔	✘/✔	✔
	Newly designed	We have full control over the API and can start with whatever seems most appropriate (WSDL, IDL, etc.)	✔	✔	✔
API Variability	Static	The API changes never or infrequently.	✔/✔	✔/✔	✔
	Dynamic	The API changes frequently or infrequently but potentially dramatically.	✘	✘	✔

Criterion		Explanation	WS	CORBA	JunC++ion
API Usage	Public	The integration API is intended for public consumption and should be as easy to use as possible.	✘	✘/✔	✔
	Internal	The integration API is used internally and does not have to be very easy to use.	✔	✔	✔
Performance	High	Language boundary crossings must be fast.	✘/✔	✔	✔
	Low	Performance is not a critical requirement.	✔	✔	✔
Deployment	xcopy	The integrated application must be copied to or movable on a system without further configuration.	✘	✘	✔
	Configuration required	The integrated application may require configuration and/or helper infrastructure.	✘	✘	✔
C++ Quality	Highly usable	The integration API needs to be of the highest quality, for example because it is published as an integration API.	✘	✘	✔
	Basic	The overall integration code usability is not so important.	✔	✔	✔
Process Character	Pure	No JVM allowed, the process must only contain pure native C/C++ code.	✔	✔	✔/✘
	Mixed	The process may allow the loading of a JVM.	✘/✔	✘/✔	✔

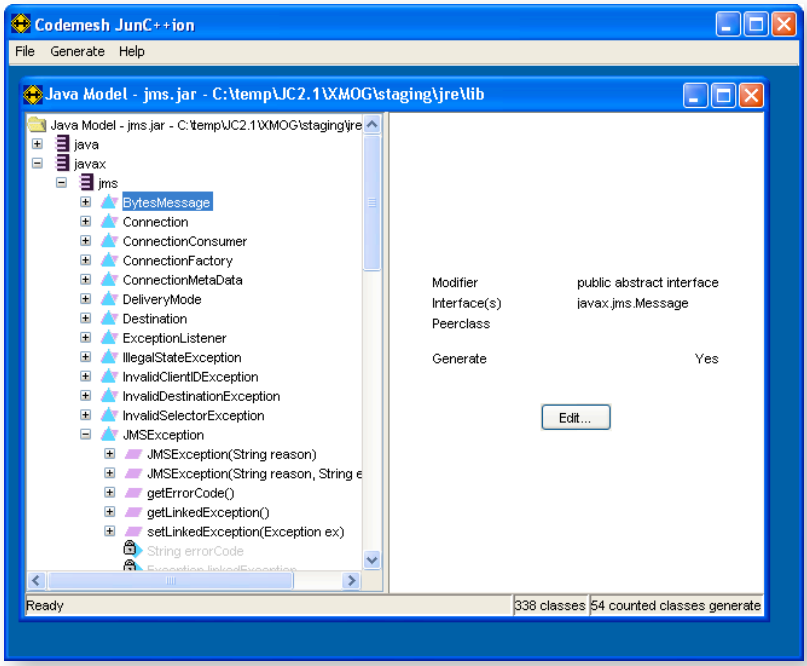
JunC++ion® – Summary and Data

Basic Features

- Generate C++ proxy type for any Java type from GUI or command line (automated build)
- Takes any jar or class file as input
- Smart code generator understands relationships between Java types
- Manually override package, type, field, or method generation
- Generate native method implementations
- Generate Visual Studio project files or makefiles
- C++ proxy types work on every supported platform, independent of generation platform
- C++ proxy types work both in C++ calling Java and Java calling C++ scenarios
- Configure Java runtime environment in code, configuration file, or registry
- Use fields as fields, methods as methods, etc.
- C++ proxy instances manage Java object life-cycle exception- and thread-safely
- Use C++ proxy types on native threads
- Support multiple native string encodings on a global or on a per-thread basis

Advanced Features

- Break up type sets into multiple modules (shared libraries)
- Implement Java interfaces in C++ (callbacks)
- Generate API documentation
- Use bundled ANT tasks for code generation and C++ compilation
- Create self-configuring shared integration libraries



The code generator GUI with a model containing `jms.jar`

Key Benefits

- Practice continuous integration by generating type-safe integration code as part of your nightly build.
- Turn 95% of integration problems into compile-time rather than run-time problems.
- Highest performing integration technology between Java and C++.
- Designed for OEM use.

Licensing & Pricing

Flexible and probably surprisingly inexpensive for a product that does not have a published price list ☺

Select Customers

Avaya, Deutsche Post AG, Fujitsu Germany, GE Healthcare (IDX), Gemstone Systems, Gigaspaces, Lockheed Martin, Luciad, McCabe & Associates, Metabit, Müller GmbH, Neartek, Northrop Grumman Corp., Panacya, Sagem, SAIC, Streambase, Thales

Operating Systems

AIX, HP-UX, Linux, MacOS-X, Solaris, Windows

Processor Architectures

IA-32, Sparc, PA-RISC, PowerPC

C++ Compilers

aCC, Sun CC, g++, MSVC++, xlC

Java Runtime Environments

IBM, JRockit , Sun

Up-to-date information

Availability of platform and compiler ports is updated regularly. Please visit our website for more detailed and up-to-date information on supported platform- and compiler-versions.

Contact Information

Codemesh, Inc.

POBox 620
Carlisle, MA 01741

T +1 978 369 8583
F +1 978 369 9088
E info@codemesh.com
W www.codemesh.com