AMQ RFC004
The Module System

version 0.1

iMatix Corporation <amq@imatix.com>

Revised: 2005/03/03

Contents

1 Cover

1.1 State of this Document

This document is a request for comments. Distribution of this document is currently limited to iMatix and JPMorgan internal use.

This document describes a work in progress. This document is a formal standard. This document is ready for review.

1.2 Copyright Notice

1.3 Authors

This document was written by Pieter Hintjens <ph@imatix.com>.

1.4 Abstract

In order to reduce the complexity of the OpenAMQ server and thus reduce the cost and effort of building, testing, maintaining, and improving it, we have chosen a design that is almost entirely based on the concept of plug-in "modules". While the server architecture is well-defined, with a specific core flow of data from client interface to destination through various layers, the actual implementation of each layer is defined as an abstract interface with an arbitrary implementation. In this document we define the general API that allows modules to register with the OpenAMQ server kernel, and we sketch the specific APIs for each class of module.

2 Introduction

2.1 Problem Statement

We need a generic way of linking arbitrary packages of C code with the OpenAMQ server, along with standard semantics to allow such packages to find and talk to each other. At the same time, the run-time cost of this abstraction must be as low as possible, ideally no more than the normal cost of invoking functions. At present we can assume that the modules are linked at build-time; the issue of loading dynamic modules can be handled at a later stage.

2.2 Argumentation

We have a number of specific problems to solve:

1. Identifying the different modules in some way.
2. Telling the linker what modules to include in our executable.
3. Finding the correct module to use for a specific task.
4. Passing information to and from the module.
5. Abstracting the module API for all modules of a specific type.
6. Providing standardised ways of testing and qualifying modules.

Let us work through each of these problems...

We may create servers with many dozens of modules. To ensure that all modules are clearly and unambiguously defined, we need naming standards. While we can speak of "modules" in general, it is also clear that each module belongs to a specific class, which circumscribes its functionality if not its specific implementation. We can decide to use the class name as part of the module name.

To use the linker to embed modules in our server binary we must call one of the functions in the module source file. It makes sense to chose a standard type of function for this, and we call this the "module main". How do we list all the module mains we want to call? At some point we have to allow the developer to modify code to include new modules. It is not a good idea to modify core code, or even any code which forms part of the formal server source package. I.E. we cannot say "this header file defines the modules to link, edit it to add your own modules", because as soon as the developer updates to a new source release, there will be a change conflict. The best solution (the one we used in similar cases in the past) is to list the required modules in the main source file. This file can be copied from templates, adapted and saved under a new name. The advantage of this approach is that each distinct main file will also correspond to a unique binary, which is exactly what we want.

To do module lookup at runtime ("what module do I have to call to process this piece of data"), we need to do two things. First, we need a standard way for modules to register themselves. Secondly, we need a standard way to use this information to choose the appropriate modules when needed. It would be possibly to short-cut this so that we can refer to modules by name, but this does not allow the flexibilty we need. A module must be totally anonymous (from the point of view of the code that uses it), and only known by

the way it has registered itself to handle different kinds of work. We will call these the "bindings" that a module makes.

Passing information to and from modules happens via normal C function calls, through what we call the "module methods". Methods are just a name for formalised function calls that follow the standard naming style used by iCL. There are, generally, two kinds of work. There is work that takes a short finite time, and there is work that takes an undetermined time, often because it depends on events coming from outside the system. We do not want module methods to 'block' while waiting for work to be done. Rather, we want such methods to return at once, and do the waiting in some way that does not block the rest of the server.

We have discussed module classes: these are the distinct types of module that we need to make our server work. Each class has a specific set of data (its properties) and functions (its methods). From the point of view of the server and code that uses modules, how these classes are implemented is not significant, so long as they conform. From the point of view of the developer it is desirable to use a framework that provides the "conformity" automatically. We have such a framework, iCL.

Lastly, it vitally important that every module can be tested in a solid manner. Since we know the module's API we should be able to define a full framework for testing each module, quite independently from its operation within the server. This has several advantages: it is faster for the developer, since problems can be found immediately rather than after a complex test involving servers, clients, and test data. It is better for the quality of the product, since modules can be "certified" by their success in passing a series of tests.

2.3 Basic Proposal

We propose a simple framework for developers in which they can write and test server modules of different kinds. This framework consists of a set of naming conventions, a toolkit (a set of iCL classes), and standard models for building modules.

3 Design Proposal

3.1 Definitions and References

See AMQ_RFC008 for terminology, and Appendix 1 for a definition of the iCL language.

3.2 Objectives

We aim to make the process of writing, testing, and using server modules as cheap and safe as possible. We assume that modules will be statically linked into the server binary.

3.3 Architecture

Our solution is based on these design elements:

1. A standard naming and typing convention for modules.
2. A standard API for registering modules.
3. A standard API for locating a module.
4. A set of standard APIs for talking to modules.
5. A standard toolkit for abstracting modules APIs.
6. A standard toolkit for testing and validating modules.

3.4 Proof and Demonstration

The architecture and design described in this document is the basis for the current implementation of the OpenAMQ server.

3.5 Detailed Proposal

3.5.1 Naming and Typing

We define modules as having:

- A class identifier
- A public name
- A version number

The class identifier is one of the standard module classes that OpenAMQ supports. The module classes are defined in AMQ_RFC010 and are encoded in amq_module.icl.

The module name is a text string that is unique within the class. The version number is a text string that documents the version number of the module. At present this is not used for any kind of checking or selection, it may be used for error solving.

The source code for an iCL class definition that defines a module class is always in a file called amq_cccc.icl where 'cccc' is the name of the class. For example:

```
amq_driver.icl
amq_terminal.icl
```

The source code for a module will be in a single file (which may include other files if necessary for its internal organisation), which will be named amq_cccc_mmmm.xxx where 'cccc' is the class name, 'mmmm' is the module name, and 'xxx' is the extension of the source file. For example:

```
amq_terminal_http.smt
amq_terminal_amqp.smt
amq_driver_filequeue.icl
amq_parser_amqxml.icl
amq_dispatcher_fast.icl
```

3.5.2 Module Registration

Telling the linker what modules to use means using some public identifier defined by that module. We standardise this as follows: all modules will have a function called "modulename_main" where "modulename" is the prefix common to all functions in that module.

Here is a main program that registers two modules and then runs the server core (thus launching the OpenAMQ server):

```
/*  Example OpenAMQ server main program  */
int amq_terminal_abc_main (void);
int amq_driver_xyz_main   (void);
int
amq_user_modules (void)
{
    if (amq_terminal_abc_main ())
        return (1);
    if (amq_driver_xyz_main ())
        return (1);
    return (0);
}
int
main (int argc, char *argv [])
{
    return (amq_core (argc, argv));
}
```

The amq_core() module starts the server and when it is ready to register user-specified modules, it calls the

amq_user_modules() function, which must be present in the main program (or included in the binary through some other route).

The _main function is defined in each module. This is a typical example:

```
amq_driver_t
    *this_driver;
this_driver = amq_driver_new (MY_NAME, MY_VERSION);
if (this_driver) {
    this_module->constructor   = &create_channel_thread;
    this_module->new_event     = new_event;
    this_module->post_event    = post_event;
    this_module->request_event = request_event;
    this_module->destroy_event = destroy_event;

    /*  Bind to the paths that we manage                                */
    /*  These would come from the server configuration                  */
    amq_driver_bind (this_driver, ...);
    return (0);
}
```

Note that the _main function returns 0 to signal success. If any module fails when registering, the server will halt.

3.5.3 Locating a Module

We do not look for modules by name, since this would presuppose a static configuration that we want to avoid. Rather, each module exports the list of services that it provides, and we provide a centralised way of finding a module associated with a service.

This is done simply by using a hash table (an array indexed by string keys) where we store the bindings each module makes. The bind method registers a new binding between a module and a key value. The lookup method finds the module for a key, if any. This is a general solution that handles all the module classes we have.

For example:

- Drivers bind on URL paths. Terminals perform lookups on paths to find the drivers they need to talk with to handle work on those paths.
- Parsers bind on MIME types. Dispatchers perform lookups on message MIME types to find the parser responsible for handling each message.

Obviously we do not need to perform a lookup each time we need a module; the _lookup method returns a static module reference that can be stored and reused.

3.5.4 Standard Module APIs

Module APIs depend on the module class implementation. We do not define a single standard for method names, argments, or implementation except that methods are synchronous and return immediately. Any

asynchronous work that a method does must be queued and handled asynchronously, e.g. by a thread in an SMT agent.

Our standard way of building an asynchronous API between two modules is:

- The calling module supplies its own reference when invoking an asynchronous method.
- The called module queues the method arguments and returns immediately.
- When the called module has finished processing the method it invokes the called module (doing a so-called "callback") with one of a set of standard methods.
- Success or failure is signalled by invoking different methods, rather than by providing a return code. When invoking an error method, the called module (now doing the callback) provides a reason code.

The implementation of each module API will be the subject of a separate AMQ RFC document. At present time, these AMQ RFCs cover module APIs:

AMQ_RFC005 The terminal-driver API.

3.5.5 Module API Abstraction

We use iCL to abstract the module APIs. Each module class is implemented by a corresponding iCL class. These classes are listed and explained in AMQ_RFC010.

To create a new module of a particular class, the developer may use iCL as a framework, but this is not essential.

There are two ways to abstract the module API for any given module class. The first way is to derive an iCL class and write the module methods in the iCL framework. This gives a new class which will be instantiated as a single object.

An alternative is to use method references in the module object. The developer creates a module object of the required class and initialises the method references to point to the correct function for each method.

We will provide examples of both model or standardise on a specific model when we have implemented this.

3.5.6 Testing and Validating Modules

The module class will provide a test handler that performs full validation of the module's functionality. We expect these test handlers to be quite large, and will be developed and extended over time.

3.6 Alternatives

We do not propose any alternatives at this moment.

3.7 Security Considerations

This proposal does not have any specific security considerations.

4 Appendices

4.1 Appendix 1 - The iCL Language

iCL is the iMatix Class Language. iCL is a fat programming language used to build and maintain classes, which are packages of code and data. The goal of iCL is to provide a standard framework in which to write modular library functions and classes. iCL is loosely based on object-oriented concepts.

4.1.1 Summary of Language

This summary shows the hierarchy of elements you can use, with the required and optional attributes for each element. The XML entity and attribute names are case-sensitive and we use only lower-case names.

```
<class name version target [comment] [copyright] [animate] [abstract] [role]>
   <doc [domain]/>
   <inherit class>
      <option name value/>
   </inherit>
   <import class/>
   <assert role/>
   <option .../>
   <invoke [script]/>
   <public [name]>
      <doc .../>
   </public>
   <private [name]>
      <doc .../>
   </private>
   <context [overlay]>
      <doc .../>
   </context>
   <method name [return] [template] [inherit] [export]>
      <doc .../>
      <argument name [type] [default]/>
      <declare name [type] [default]/>
      <animate/>
      <generate/>
      <option .../>
      <local>
         <doc .../>
      </local>
      <header/>
      <footer/>
   </method>
   <template name [return] [inherit] [export]>
      <doc .../>
      <argument .../>
      <declare .../>
```

```
      <animate .../>
      <generate .../>
      <local .../>
      <header .../>
      <footer .../>
   </template>
   <todo [owner]/>
   <generate .../>
</class>
```

4.1.2 Detailed Specifications

All child entities are optional and can occur zero or more times without any specific limits unless otherwise specified. The same tag may occur at different levels with different meanings, and in such cases will be detailed more than once here.

The 'class' Item

The class tag defines the class. One iCL file defines exactly one class.

```
<class
    name = "..."
    version = "..."
    target = "stdc | perl | ruby | java"
  [ comment = "..." ]
  [ copyright = "..."  ("Copyright(c) iMatix Corporation") ]
  [ animate = "0 | 1"  ("0") ]
  [ abstract = "0 | 1"  ("0") ]
  [ role = "..." ]
    >
    <doc>
    <inherit>
    <import>
    <assert>
    <option>
    <invoke>
    <public>
    <private>
    <context>
    <method>
    <template>
    <todo>
    <generate>
</class>
```

The class item can have these attributes:

name Specifies the name of the class. This name will be used to prefix all function names and will also be used as the filename for generated code. The name attribute is required.

comment An optional one-line comment that describes the class. The comment attribute is optional.

target Specifies the name of the target environment; the target is implemented by a GSL script that generates code for a specific language environment. The target can be inherited from a parent class. The target attribute is required. It can take one of the following values:

Value	Meaning
stdc	Standard ANSI C + iMatix runtime
perl	Perl 5.x
ruby	Ruby
java	Java 1.2

version Specifies the version of the class. This text can take any format but we recommend this standard format: '2.4b1' which is major version 2, minor version 4, release b, update 1. This string is stamped into the project sources. The version attribute is required.

copyright This specifies the copyright string for the class. This string is stamped into the project sources. The copyright can be inherited from a parent class. The copyright attribute is optional. Its default value is "Copyright(c) iMatix Corporation".

animate If set, the generated code contains animation that can be switched on and off at runtime. This option can be overridden by a command-line switch (e.g. "-animate:0"). The animate option can be inherited from a parent class. The animate attribute is optional. Its default value is "0". It can take one of the following values:

Value	Meaning
0	do not animate
1	generate animation code

abstract If set, no code is generated. The abstract attribute is optional. Its default value is "0". It can take one of the following values:

Value	Meaning
0	normal class
1	abstract base class

role Defines the role for a base class. A class can assert using the assert tag, that at code generation time (possibly in derived class) certain roles are present. The role attribute is optional.

The 'doc' Item

Documentation for the current element: this is included in the generated source code in a suitable form. Documentation should be in iMatix gurudoc format.

```
<doc
  [ domain = "..." ]
    />
```

The doc item has this single attribute:

domain Allows documentation of different types to be included in the iCL definitions. The domain attribute is optional.

The 'inherit' Item

A class can be based on one or more parent classes, defined using one or more inherit elements. Each inherit element refers to an iCL file, a single class. All elements in the parent class can be inherited into the class; the precise rules for inheritence depend on the element and are explained later. Multiple inheritence is permitted, so a class can itself inherit definitions from a parent class. All these iCL files must be present and available during code generation.

```
<inherit
    class = "..."
    >
    <option>
</inherit>
```

The inherit item has this single attribute:

class The name of the class inherited. The parser will look for an iCL file with the specified name and the extension ".icl". The class attribute is required.

The 'option' Item

Passes an option to the method template or inherited class. Options can be used in the template code generation logic, or in method handlers.

```
<option
    name = "..."
    value = "..."
    />
```

The option item can have these attributes:

name The name of the option. The name attribute is required.

value The value for the option. The value attribute is required.

The 'import' Item

Specifies other classes that this class refers to. Note if you want the generated code to be correct you must define an import item for each class that you refer to in your class context or methods. By default, the import tag is inherited unless you specify inherit = "0".

```
<import
    class = "..."
    />
```

The import item has this single attribute:

class The name of the class imported. The class attribute is required.

The 'assert' Item

Asserts that a specified class role is present at code generation.

```
<assert
    role = "..."
    />
```

The assert item has this single attribute:

role The name of the class role being asserted. If no class is present (inherited or current) with this role, the code generation process aborts. The role attribute is required.

The 'invoke' Item

Invoke gsl code to operate on the class tree. Invokes the gsl code contained in the invoke item body, if any, followed by the gsl code specified by the script attribute, if present. The gsl code can access the class entity and manipulate it in any way desired. It may also generate other files; the script is invoked before any other code generation starts. The invoke tag can contain arbitrary XML definitions for use by the gsl script.

```
<invoke
  [ script = "..." ]
    />
```

The invoke item has this single attribute:

script The name of the GSL script, without any extension (.gsl is enforced). The script attribute is optional.

The 'public' Item

Public definitions, exported for use by callers of the class. In C, these definitions are copied into the class header file. Public definitions are inherited from the parent classes unless you specify inherit = "0". If the definitions are named, each named block is indepedently inherited.

```
<public
  [ name = "header | include | types | functions | footer"  ("types") ]
    >
    <doc>
</public>
```

The public item has this single attribute:

name The name of the public block, which really means the place in the generated code that this public block shoud be inserted. The name attribute is optional. Its default value is "types". It can take one of the following values:

Value	Meaning
header	issued at start of file
include	issued after class imports
types	for type definitions
functions	for functions prototypes
footer	issued at end of file

The 'private' Item

Private definitions, used by the class itself. The private definitions can include static variables and local functions. Private definitions are inherited from the parent classes unless you specify inherit = "0". If the definitions are named, each named block is indepedently inherited.

```
<private
  [ name = "header | body | footer"  ("body") ]
    >
    <doc>
</private>
```

The private item has this single attribute:

name The name of the private block, can be "header" to hint the code generator to place this at the top of the generated file. The name attribute is optional. Its default value is "body". It can take one of the following values:

Value	Meaning
header	issued at start of source
body	issued in middle of source
footer	issued at end of source

The 'context' Item

Defines a context block; one or more variables which will be held in all class instances.

```
<context
  [ overlay = "..." ]
    >
    <doc>
</context>
```

The context item has this single attribute:

overlay If specified, the context block of the specified class is used as the basis for the context block; this can be used for containers that need to be able to manipulate child objects with the same context structure as themselves. Use with care, experts only! The overlay attribute is optional.

The 'method' Item

Methods provide functionality for the component class. Methods can operate on specific objects, on the whole class of objects, or on other arbitrary data. Methods are inherited from parent classes unless the inherit attribute is set to "0". In the new and destroy methods, the class is addressed using the name "self".

```
<method
    name = "..."
  [ return = "..." ]
  [ template = "..." ]
  [ inherit = "0 | 1"  ("1") ]
  [ export = "before | after | none"  ("before") ]
    >
    <doc>
    <argument>
    <declare>
    <animate>
    <generate>
    <option>
    <local>
    <header>
    <footer>
</method>
```

The method item can have these attributes:

name The name of the method, used in the API. The name attribute is required.

return The name of the returned value. This must be one of the items declared in the method body using 'declare'. The return attribute is optional.

template If specified, defines a method that acts as template for this method. The template defines arguments and return values but no code. Templates are a quick way of defining many methods that have a similar API. Note that the template is extended by any arguments or declarations provided in the method itself. The template attribute is optional.

inherit If inherit is defined as 0 then the method will not be inherited from parent classes. If inherit is 1, or not specified, the method is inherited from the parent classe and extended. The inherit attribute is optional. Its default value is "1". It can take one of the following values:

Value	Meaning
0	may be inherited
1	is not inherited

export Defines the inheritence model for the method. There are three models: 'after' means child code and definitions are added after the inherited code and definitions; 'before' means child code and definitions are added before inherited ones; 'none' means the method is not inherited. The export attribute is optional. Its default value is "before". It can take one of the following values:

Value	Meaning
before	new code comes last
after	old code comes last
none	no inheritence

The 'argument' Item

Defines one argument passed to the method. The body of this entity is used to document the argument.

```
<argument
    name = "..."
  [ type = "..." ]
  [ default = "..." ]
    />
```

The argument item can have these attributes:

name The name of the argument. The name attribute is required.

type The type of the argument, which is a native type name. The type attribute is optional. Its default value is "".

default The argument default value, used for integer arguments with value zero, and string and reference arguments with value null. The default attribute is optional.

The 'declare' Item

Defines data declarations for the code that follows. All local variables used in the code body must be placed in declare tags so that the final code can be correctly generated.

```
<declare
    name = "..."
  [ type = "..." ]
  [ default = "..." ]
    />
```

The declare item can have these attributes:

name The name of the variable. For non-atomic variables like arrays, this can contain the full variable declaration. Note that only atomic variables can be passed as arguments. The name attribute is required.

type The type of the argument, which is a native type name. To use a reference to the the current class (a pointer in C), use the value "$\(selftype) *". The type attribute is optional. Its default value is "".

default The default value for the variable. To define a string value, you must enclose it in " symbols. The default attribute is optional.

The 'animate' Item

Provides a message or comment that will be shown when animating the component. The animate tag can be mixed with code.

```
<animate>
```

The 'generate' Item

Contains GSL code (in script form) that is executed before code generation. This GSL code can directly manipulate the class XML tree as needed. For an example, see the icl_base.icl class.

```
<generate>
```

The 'local' Item

Variable definitions used by the method itself. The local definitions are a simpler way of defining blocks of variables than using declare items. Note you must use a declare for the return value.

```
<local>
    <doc>
</local>
```

The 'header' Item

Defines a block of method code that should come before all bodies from the current and parent classes. Do not use for variable declarations, use 'local'.

```
<header>
```

The 'footer' Item

Defines a block of method code that should come after all bodies from the current and parent classes.

```
<footer>
```

The 'template' Item

Defines a method template.

```
<template
    name = "..."
  [ return = "..." ]
  [ inherit = "..."  ("1") ]
  [ export = "..."  ("before") ]
    >
    <doc>
    <argument>
    <declare>
    <animate>
    <generate>
    <local>
    <header>
    <footer>
</template>
```

The template item can have these attributes:

name The name of the template, referred to by methods. The name attribute is required.

return Provides a default 'return' attribute to methods based on this template. The return attribute is optional.

inherit Provides a default 'inherit' attribute to methods based on this template. The inherit attribute is optional. Its default value is "1".

export Provides a default 'export' attribute to methods based on this template. The export attribute is optional. Its default value is "before".

The 'todo' Item

Defines a change request, bug or other issue that needs changing in the iCL class. Todo items are formalised so that they can be extracted and processed mechanically.

```
<todo
  [ owner = "..." ]
    />
```

The todo item has this single attribute:

owner The developer who registered the issue and will deal with it, specified as an email address. The owner attribute is optional.

5 Comments on this Document

5.1 Date, name

No comments at present.