

EQUIP Runtime

Chris Greenhalgh

1st April 2001. v.1.1.

Last updated 11th April 2001.

Introduction

This document describes the core runtime of the EQUIP platform, including its modularisation, type definition language and common runtime facilities. Each section begins with cross-language (Java and C++) facilities, and then adds language-specific observations.

The sections are as follows:

- Modularisation (Bamboo)
- Other external dependencies
- Object definition (IDL)
- Common object capabilities (including memory management)
- Basic types

Modularisation (Bamboo)

EQUIP uses Bamboo (see below) to provide the following elements of the EQUIP build- and run-time:

- module organisation;
- cross-platform build environment for C++;
- C++ runtime including dynamic code loading (optional Java runtime).

This section describes some of the main aspects of Bamboo that are relevant to its use in EQUIP (and to understanding the file organisation of EQUIP).

Bamboo is a cross-platform, cross-language build environment and run-time system by Kent Watsen, started while he was at the Naval Postgraduate School, US (<http://www.watsen.net/Bamboo>). Bamboo was motivated by ideals of long-lived multi-user virtual worlds, but is not VR-specific. Indeed, Bamboo itself has no VR-specific facilities at all, although the related Bambox modules include VR-specific components (<http://www.watsen.net/Bambox>).

Bamboo is written primarily in C++, and uses the Netscape Portable Runtime (NSPR, see next section) from the Netscape Mozilla project to build and run cross-platform.

The EQUIP software release includes a version of Bamboo as 'Equator/Bamboo'. The file 'Equator/Modules/equip/BuildNotes.cmg' includes information about compiling Bamboo, and pre-compiled versions are also available for some platforms from the EQUIP web page.

Modules and Plugins

Bamboo requires that systems and applications be organised as 'modules'. These are the main unit of logical organisation, and are also the typical unit of code distribution and downloading. EQUIP itself is organised as a number of Bamboo modules, as noted in the overview (e.g. equip/eqRuntime, equip/eqNet, ...). Each module corresponds to a single directory and its contents (including certain subdirectories). For example the EQUIP release directory 'Equator/Modules/equip/eqRuntime' is the EQUIP runtime module.

Every module directory contains a Bamboo Makefile that identifies the directory as a module. When built by Bamboo the module directory will also contain a module description file called '.module.txt' (note the leading full stop which will prevent the file being listed by default on UNIX).

Each module may contain data files and/or one or more language-specific 'plugin's. A C++ plugin corresponds to a dynamic loadable library, while a Java plugin corresponds to a JAR archive of Java class files.

The standard subdirectories within a Bamboo module directory are:

- include: for C/C++ include files that must be accessible to other modules that depend on this one.

- `src`: for C/C++ and Java source files.
- `lib`: autogenerated by the build system to hold the generated libraries and jar files for the module.

Plugin Dependencies

When compiling and loading plugins Bamboo enforces build/load dependencies between plugins. These are specified in the plugin Makefile as `PLUGIN_DEPEND`. The exact build-time handling of dependencies is language specific (see below).

C++ Plugins

A C++ plugin corresponds to a dynamic loadable library.

To create a C++ plugin there must be a Bamboo plugin Makefile in the `src` subdirectory of the module. This must specify the `PLUGIN_LANG` (or `PLUGIN_OTHER_LANG`) as `CPP`. The C/C++ files to be included in the dynamic library must be explicitly specified in the plugin Makefile. Files ending `.c` are assumed to be plain C; files ending `.cpp` or `.cc` are assumed to be C++. Look at existing modules for examples.

When compiling a C/C++ plugin Bamboo will create the following directories and files:

- `lib`: directory to hold DLL/shared object and import libraries. Note that the filename for the DLL/shared object and import libraries will be based on the module name.
THEREFORE ALL MODULE NAMES MUST BE DISTINCT EVEN WITH NO PATH COMPONENTS.
- `src/OBJS-*`: a platform-specific directory to hold object files and other generated files.

If a C++ plugin is specified to depend on another module then the following are implied:

- the specified modules/plugins will be checked (and compiled if necessary) before this plugin is compiled.
- the `include` subdirectory of the specified modules will be passed to the C/C++ compiler as a potential source of include files (typically using the `-I` flag).
- the import library for the specified C++ plugins (if any) will be included in the link options for the current plugin.

Bamboo (at least in its current version) will check transitive dependencies (other modules that the module that you depend on depend on in turn) but will NOT add the include or link information. Therefore in many cases it will be necessary to add transitive dependencies explicitly.

Java Plugins

A Java plugin corresponds to a JAR archive of Java class files.

To create a Java plugin there must be a Bamboo plugin Makefile in the `src` subdirectory of the module. This must specify the `PLUGIN_LANG` (or `PLUGIN_OTHER_LANG`) as `JavaLoader`. All `.java` files in the `src` directory or its subdirectories are included in the plugin and need not be specified explicitly.

If the java files are part of a package then the package directory structure should start at the `src` directory. For example the `eqRuntime` Java files are in the directory `equip/eqRuntime/src/equip/runtime`, reflecting the IDL namespace (see later) and Java package name chosen for them (i.e. `equip.runtime`).

When compiling a Java plugin Bamboo will create the following directories and files:

- `lib`: directory to hold the plugin JAR file. Note that the filename for the JAR file will be based on the module name. THEREFORE ALL MODULE NAMES MUST BE DISTINCT EVEN WITH NO PATH COMPONENTS.
- `src/classes`: to hold the class files before the JAR file is created.

If a Java plugin is specified to depend on another module then the following are implied:

- the specified modules/plugins will be checked (and compiled if necessary) before this plugin is compiled.
- the JAR files for any specified Java plugins will be included in the `CLASSPATH` used to compile this plugin.

Bamboo (at least in its current version) will check transitive dependencies (other modules that the module that you depend on depend on in turn) but will NOT add the CLASSPATH information. Therefore in many cases it will be necessary to add transitive dependencies explicitly.

Common commands

Bamboo requires a UNIX-like environment; on Windows this requires a number of supporting commands (e.g. gmake) that are available from the Bamboo and the EQUIP webpages.

Bamboo's compile system operates recursively in the current directory and its subdirectories. The main commands are:

- 'gmake compile'
 - recurse, compiling modules and plugins as they are found.
- 'gmake clean'
 - delete build-time files that are not needed to run the module(s).
- 'gmake clobber'
 - delete all non-source files including those normally needed to run the modules.
- 'bamboo <modulepath> [-<modulearg> ...] ...'
 - load and initialise (run) the specified module(s).

Module names, paths and loading

Each module must have a unique name – the name of its main directory. Modules can be specified to Bamboo for loading (either on the Bamboo command line, or from a currently loaded module) in four different ways:

- Name only. This causes Bamboo to recursively search the BB_DIR directory and its subdirectories, each directory in BB_PATH and their subdirectories and the user's Bamboo cache directory (typically HOME/.bamboo/cache) and its subdirectories searching for a module of the given name. This is slow and should be avoided if at all possible.
- Relative name (including a non-leading slash '/'). This causes Bamboo to check for the module relative to BB_DIR, the elements of BB_PATH and the user's Bamboo cache directory. This is much faster than name only and should normally be used (just specify './...') for a top-level module.
- Absolute path name (with a leading slash, '/'). This causes Bamboo to check the specified local file path.
- HTTP URL (e.g. 'http://host:port/path/name'). This causes Bamboo to attempt to download a platform-specific 'bar' (Bamboo Archive) file from related URL 'http://host:port/path/name.platform.bar' where 'platform' identifies the runtime platform, e.g. 'winnt'. This bar file can be a '.tar.gz' file of the module directory, renamed to '.bar'. This file is downloaded and unpacked in the user's Bamboo cache directory. Bamboo also has an archive utility that uses PGP (Pretty Good Privacy) to make and check cryptographically secure signatures on '.bar' files; this requires that you have PGP installed – see the Bamboo archive documentation for details.
Note that downloading from URL requires that the module be the only and top-level directory in the tar file. This is also not fully implemented in the current version of Bamboo, although it may well work.

Note that if any of the last three forms fail then they fall back to the first, exhaustively searching for the specified module name.

Also note that EQUIP additionally constrains the names to be used for dynamically loaded modules. Specifically the EQUIP runtime assumes that a class 'X.Y.Z' (i.e. module/namespace/package 'X.Y' or '::X::Y') will be found in the Bamboo module 'X_Y'. E.g. all dynamically loadable classes in the 'equip.data' module/package/namespace will be in the module 'equip_data'. EQUIP also uses the EQUIP-specific environment variable 'EQUIP_MODULE_PREFIX' to specify the module's relative path to speed the loading process. Normally this should be set to './' (assuming top-level modules). Note that this functionality is currently (10/4/01) only directly supported from C++ modules.

Other external dependencies

This section outlines the other main pieces of external software upon which EQUIP depends. These are:

- NSPR: C++ cross platform runtime and libraries;

- libIDL: IDL parser library.

Netscape Portable Runtime (NSPR)

NSPR is a set of C dynamic libraries that provide a cross-platform runtime environment for C applications. NSPR builds on Windows, most flavours of UNIX and also on MacOS. The libraries include cross-platform versions of the following:

- threads (over Windows, posix and a couple of others), including locks, etc;
- IP sockets;
- malloc-style memory allocation;
- stdlib-type functions.

NSPR is used by Bamboo, and must be compiled/obtained before Bamboo can be built/run. The EQUIP web site contains precompiled versions for some platforms. NSPR can be obtained from the EQUIP website or from the Mozilla project (<http://www.mozilla.org/>).

NSPR is used for cross platform threading and sockets in EQUIP, and it is suggested that new modules make use of NSPR facilities wherever possible rather than platform-specific alternatives.

libIDL

libIDL is an open source IDL parser library written in C. It is used by the XP-COM project with Mozilla and by the Gnome project CORBA ORB.

libIDL has been extended for EQUIP (based on version 0.6.8) to provide partial support for CORBA valuetypes. This is used to support language-neutral type and class specification for EQUIP.

The EQUIP-modified version of libIDL is available in the EQUIP release under 'Equator/libIDL'. Note that this is not a Bamboo module, and is only used the standalone application 'eqidl' (described later).

libIDL depends on the open source GLib portable C/type library, which must be obtained separately.

Object definition (IDL)

EQUIP uses a subset of CORBA Interface Definition Language (IDL) to specify types and classes in a language-neutral way (Note that there are several things called 'IDL', including Microsoft COM IDL, however they are all different in detail).

CORBA is a public standard for an Object-oriented distributed middleware, and is cared for by the Object Management Group (www.omg.org). The standards are available on the web. The most relevant for EQUIP and the core specification (version 2.4) and the ++ and Java language binding specifications.

The rest of this section provides a brief introduction to the relevant parts of IDL and describes the 'eqidl' application used for code generation for EQUIP.

Introduction to CORBA IDL

CORBA IDL is similar to C++ (and Java to a lesser extent), but is restricted to only support the definition of types and interfaces, rather than implementations.

General notes

IDL declarations are normally put in a file ending '.idl'. In Bamboo/EQUIP these should be in the relevant module's 'include' directory.

IDL understands C++ and Java type comments, i.e.:

- `///
//` begins a comment that lasts for the remainder of that line of input;
- `/*
*/` bracket a comment that may span multiple lines.

In principle, IDL should allow the use of a full C preprocessor, however the current version of 'eqidl' only supports the following directive:

- `#include "filename"`
includes the text of the specified IDL file in place.

Namespaces

IDL has a concept of 'namespaces' that is similar to C++'s, and related to Java's use of packages. For example:

```
namespace Foo {
    typedef int myInt;
    // ...
}
```

opens a namespace called 'Foo' which includes all declarations within the curly brackets. The effect of this is prepend the specified namespace name to all identifiers declared within the given scope. Thus the type name 'myInt' is actually '::Foo::myInt'. Note that, as in C++ the namespace separator is '::' and the root namespace can be explicitly specified by a leading '::'. In Java this becomes 'Foo.myInt', i.e. IDL namespace 'Foo' becomes package 'Foo'.

Unlike C++ and Java IDL has no equivalent to 'using' (C++) or 'import' (Java). In other words, all qualified names must be individually specified in the IDL file whenever they appear outside of the defining scope. For example (continuing from the sample above):

```
namespace Foo {
    // resolved to ::Foo::myint
    typedef myInt myOtherInt;
}
// namespace must be specified
typedef Foo::myInt aThirdInt;
```

Note also that namespaces can be arbitrarily nested.

Basic Types

The basic types defined in IDL (and their C++ and Java equivalents) are described below. Note tht 'eqidl' uses a non-standard mapping of types to C++ which is specific to the use of NSPR:

- boolean: PRBool (C++), boolean (Java)
- char: char (C++), char (Java)
- octet: PRInt8 (C++), byte (Java)
- short: PRInt16 (C++), short (Java)
- unsigned short: PRUint16 (C++), short (Java)
- long: PRInt32 (C++), int (Java)
- unsigned long: PRUint32 (C++), long (Java)
- long long: PRInt64 (C++), long (Java)
- unsigned long: PRUint64 (C++), long (Java)
- float: float (C++), float (Java)
- double: double (C++), double (Java)

IDL also has a wide character type, but this is not yet fully supported in 'eqidl' (in Java it is just a 'char').

Enumerations

Enumerated (integer) values cn be specified in a similar way to C/C++, except that values cannot be specified:

```
enum Blah {
    BLAH_A, BLAH_B, BLAH_C
};
```

In C++ this generates a similar enum, with `Blah` as the type name. In C++ `BLAH_A`, etc. are in the same namespace as `Blah`. The first enumerated value has integer value `0`.

In Java this generates a class `Blah` that include `static final int` values for `BLAH_A`, etc. Consequently they are also within the `Blah` sub-namespace. In eqidl the type `Blah` becomes `int` in most contexts.

Constants

IDL allows constants of most built-in types:

```
const int aConstant = 1;
```

In C++ this maps to a similar `const` declaration.

In Java, if this is not inside a valuetype of interface then this maps to a class called `aConstant` that has a single static final class variable called `value` with the specified value. Otherwise, it maps directly to a static final class variable with the given name and value.

Strings

IDL has two string types: `string` and `wstring` (wide char string).

The Java mapping for both is `java.lang.String`.

The C++ mapping for `string` depends on its context within the IDL declaration:

- as an `in` parameter to an operation it is `const char *`;
- as a return value from an operation it is `char *` (I think);
- as a member variable of a valuetype (object) it is `String_var` (defined in `equip/eqRuntime` - see later).

eqidl does not currently support `out` or `inout` parameters, and the C++ mapping for `wstring` is not fully implemented yet in eqidl.

Type aliases

IDL supports a `typedef` type declaration similar to C/C++. This introduces a new name for an existing simple or compound type (e.g. array or sequence).

In C++ these map into equivalent C++ `typedef` statements.

There is typically no Java mapping for a `typedef` (although in full CORBA it causes the generation of associated Helper and Holder classes).

Fixed size arrays

Fixed size arrays are specified as in C++:

```
// IDL
typedef int threeInts[3];
```

Fixed size arrays can be used in `typedefs` and valuetype member variables. They cannot be used directly as operation argument or return types (although defined array types such as `threeInts`, above, can be).

The C++ mapping of a fixed-size array is an equivalent fixed size array.

The Java mapping of a fixed-size array is a normal (unsized) Java array. It is the application programmer's responsibility to ensure that it is correctly sized. Autogenerated routines (marshalling, equals, matches) are likely to raise exceptions if the array size is wrong.

Variable size arrays (sequences)

IDL specifies variable sized arrays as in the following example:

```
typedef sequence<int> someInts;
```

Like fixed size arrays, sequences can be used in `typedefs` and valuetype member variables. They cannot be used directly as operation argument or return types (although defined array types such as `someInts`, above, can be).

In C++ these map to the template class `Sequence` (defined in `equip/eqRuntime` and described later in this document). For example:

```
// C++
typedef Sequence<PRInt32> someInts;
```

In Java these map to the normal Java array type.

IDL also includes limited size sequences ('<type, size>') but these are not yet supported in eqidl.

Interfaces and methods (Operations)

IDL allows interfaces to be specified. Like Java interfaces these can contain constants and abstract (unimplemented) methods (known as operations), but not class or instance variables, constructors or destructors.

eqidl does not currently support interfaces, however valuetypes (below) can also contain operations, so we will still introduce operations.

Each operation is mapped into an instance method of the same name.

An IDL operation has one return type (which may be 'void') and zero or more parameters. Each parameter must have one of the following modes specified for it:

- 'in' - the value may not be modified by the operation, and the operation may not retain the passed reference (if applicable) beyond the completion of the operation unless it performs appropriate memory management. If it does retain a reference then it must not use this to change the referenced parameter. The calling code retains its reference to the parameter (and responsibility for freeing it, if relevant).
- 'out' - the value is returned from the operation. It is allocated by the operation code, but becomes the responsibility of the calling code once the operation has completed.
- 'inout' - the value is passed into the operation, where it may be changed, and the original or new value will be passed back to the calling code, which is responsible for freeing it.

At the present time eqidl only supports 'in' parameters.

Parameter and return types must be simple, i.e. a built-in type or a typedef'd name.

'in' parameter and return types are mapped as follows:

- basic types: equivalent native type;
- enum: (C++) the enum name is used, (Java) 'int';
- string: (C++) 'const char *', (Java) 'java.lang.String';
- fixed array or sequence: (C++) the typedef name is used, (Java) 'T []';
- valuetype: (C++) 'T *', where 'T' is the valuetype name, (Java) 'T'.

Serialisable objects (valuetypes)

As of version 2.3 CORBA has added support for serialisable objects to IDL. These are also referred to as Object By Value (OBV) because of their semantics when used as interface operation parameters.

For example:

```
valuetype Shape {
    public float size;
    void draw();
};
valuetype Square : Shape {
    public string colour;
};
```

Defines two valuetypes, 'Shape' and 'Square'. Each maps to a class (C++ or Java) of the same name. 'Square' derives from (Java: 'extends') Shape. The 'Shape' class has the single public instance variable 'size'. The 'Square' class inherits this, and adds its own instance variable 'colour'. The class 'Square' defines a single instance method 'draw' that must be implemented in all subclasses.

IDL generates the interface but cannot generate the actual implementation of 'draw' because it is not linked to any one programming language.

IDL requires that the application programmer provide concrete classes that subclass these and provide the actual implementations. By default eqidl generates only the abstract classes ('Shape' and 'Square' in this case). However

it can also generate empty template versions the required implementation classes ('ShapeImpl' and 'SquareImpl' in this case). Note that eqidl will refuse to overwrite existing implementation files unless forced to to avoid destroying your implementation code.

Valuetypes only support single inheritance. Valuetypes with no specified base class are given the built in base class `::equip::runtime::ValueBase` (Java `equip.runtime.ValueBase`), following the CORBA convention.

It is not possible to directly inherit methods implementations from 'ShapeImpl' to 'SquareImpl'. Instead a delegation strategy is suggested: a delegate class (e.g. 'ShapeDelegate') is created while implements operations from 'Shape' but without deriving from it. 'ShapeImpl' is now simple to code: it creates a corresponding instance of class 'ShapeDelegate' and passes all method invocations on to it. This same 'ShapeDelegate' class can be used in 'SquareImpl' to easily pull in standard 'Shape' functionality.

Instance variables may be 'public' or 'private'. 'Private' instance variables are mapped to 'protected' in both C++ and Java since they must be accessible to the implementing subclass.

Operations are mapped to instance methods as described in the section on interfaces (above).

Instance variable types are mapped as follows:

- basic types: equivalent native type;
- enum: (C++) the enum name is used, (Java) 'int';
- string: (C++) 'String_var' (see `eqRuntime`), (Java) 'java.lang.String';
- fixed array or sequence: (C++) the typedef name is used, (Java) 'T []';
- valuetype: (C++) 'T_var *', where 'T' is the valuetype name and 'T_var' is related type defined from the template class `::equip::runtime::ObjectVar<T>` (see `eqRuntime`), (Java) 'T'.

As noted, in C++ the helper classes 'Shape_var' and 'Square_var' would also be generated (described with C++ memory management).

Unimplemented

The following are currently unsupported in eqidl:

- wchar
- wstring
- value box
- limited size sequence
- interfaces
- 'out' and 'inout' parameters
- valuetype factories

EQIDL

The IDL compiler written for EQUIP is called 'eqidl'. It parses IDL files and can output corresponding C++ header and body file and Java files.

eqidl is currently distributed and built in the EQUIP version of libIDL (Equator/libIDL/libIDL-0.6.8); at some point in the future it will be pulled out to a separate directory, independent of libIDL (since it is not subject to the same licensing terms - libIDL is LGPL).

Code generation is controlled by the following command line arguments:

- '-h': output C++ header.
- '-c': output C++ body.
- '-j': output Java files (of course Java has no separate headers).

In each case the default path for the generated files is the same as that of the input IDL filename. Alternatively the can be followed (with no space) by an alternative directory path for generated files.

For C++ the base filename for the generated files is the same as the input IDL filename.

For Java the base filename for the generated files is the name of the corresponding class or type. The Java files are also placed in (dynamically created) subdirectories corresponding to the package(s) in which the classes are specified (corresponding to the IDL namespace(s)).

eqidl can optionally generate implementation template files as controlled by the following command line options:

- '-i': generate implementation files for all selected output options (above). These have 'Impl' appended to the base filename.
- '-f': force generation of implementation even if they already exist (be careful).

eqidl also understands the following command line options:

- '-I<path>': look for include files in the specified directory.
- something about windows linking...

The remaining argument(s) are one or more IDL files to be processed.

Note that eqidl will only generate output for types defined in the directly read IDL file(s) and not for types defined in included files (although these are also read and parsed in order to check types and generate correct output).

At the present time eqidl is only built on Windows, although sorting out the few dependencies should be easy.

Common object capabilities

Much of the EQUIP runtime module is concerned with providing runtime support for the classes generated by eqidl from valuetype definitions (see previous sections).

Module 'equip_runtime' uses namespace '::equip::runtime', package 'equip.runtim' and can be found in the release directory 'Equator/Modules/equip/equip_runtime'.

In relation to eqidl, equip_runtime provides the following (described in this section) (all are in namespace ::equip::runtime):

- Object base class and memory management.
- ValueBase base class.
- Status enum.
- (C++ only) sequence template class Sequence<T>.
- (C++ only) string variable class String_var, to help with memory management (unnecessary in Java due built in java.lang.String class and garbage collection).
- (C++ only) object variable template class ObjectVar<T>, to help with memory management (unnecessary in Java due to garbage collection).
- stream classes ObjectInputStream and ObjectOutputStream for ValueBase serialisation.
- (C++ only) basic stream classes - abstract, binary, ascii and file.

These are described in turn below.

::equip::runtime::Object base class and memory management

The 'Object' class is not visible in IDL. It provides one common method (specified as if in IDL):

- string getMethodName();
returns the name of the module that contains the implementation of this class. This is not currently supported (pending dynamic code loading).

Java

In Java 'equip.runtime.Object' extends 'java.lang.Object' which provides most of the required functionality. E.g. getClass, 'instanceof' and type-safe type casting (type assertions), equals, memory management, locking and synchronization.

C++

In C++ `equip::runtime::Object` provides C++ versions of some facilities that are standard in Java:

- `ClassInfo *getClass();`
`ClassInfo` is specific to EQUIP.
- `static ClassInfo *_class();`
- `const char *getClassName();`
the equivalent of (Java) `getClass().getName()`.
- `void lock();` and `void unlock();`
Obtain (and release) a re-entrant lock on the specified object. This will only block other calls to lock on the same object and will not affect method invocations that do not explicitly call lock.
- `static Object *_downcast(Object *o);`
There is a method like this in every eqidl-generated class that performs a type-safe cast to the specified class (this example performs a no-op). If the object is not a subclass of the class whose `_downcast` method is used then the result is NULL. This effectively combines 'instanceof' and type cast/assertion ('(TYPE)o') in Java. This method is specified in CORBA.

Note that memory management is rather more complicated in C++ than in Java. EQUIP uses reference counting for memory management. The current implementation probably does not cope with reference loops, so be careful. The reference counting API is specified by CORBA:

- `equip::runtime::Object *_add_ref();`
increase the reference count by one.
- `void _remove_ref();`
reduce the reference count by one, deleting the object if it reaches 0.
- `int _refcount_value();`
return the current reference count.

To help to avoid memory leaks in C++ the template class `ObjectVar<class T>` is provided. Each class defined by eqidl (e.g. 'T') also defines a 'variable' class based on this:

```
typedef ObjectVar<T> T_var;
```

This type is used to 'hold' object references used as member variables in valuetypes. When an instance of this class is destroyed (e.g. passes out of lexical scope) it calls `_remove_ref()` on the reference that it holds. Note that a `T_var` is 'greedy' for references. E.g. assigning a simple 'T*' pointer to a `T_var` causes the `T_var` to assume ownership of the reference (removing it on destruction). If you do not own the 'T*' reference (e.g. it was received as a method parameter) then you must explicitly call `_add_ref()` on it before assigning to the `T_var`. Assigning between two `T_var`'s of the same type automatically adjusts reference counts correctly. Getting the reference out of the `T_var` uses one of the `T_var` operators or methods (see `Equator/Modules/equip/equip_runtime/include/eqTypeVar.h`):

- `Object_var var = new Object();`
var takes ownership of the object reference/pointer and calls `Object::_remove_ref()` when it goes out of scope.
- `Object_var var2 = var;`
calls `_add_ref` on the object reference.
- `var->F(...);`
to get access to the pointer to call a method F
- `Object *ptr = val.in();`
to get the pointer (without `add_ref!`) and with the `Object_var` still owning it, e.g. as an 'in' parameter to an operation:
- `... val.inout() ...`
to get a reference to the internal pointer, to allow updating, e.g. as an `Object*&` as an 'inout' parameter.
- `... val.out() ...`
to get a reference to the internal pointer, releasing any value currently referenced, e.g. as an 'out' parameter.

- `return val._retn();`
to return the reference, yeilding ownership (i.e. will then contain null), e.g. in returning value from a fn.

ValueBase base class

This is the base class for all valuetype classes generated by eqidl. It includes a number of standard autogenerated facilities, typically accessed via virtual methods in the Object and ValueBase base classes (ValueBase is derived from `::equip::runtime::Object/ equip.runtime.Object`). From Object these are `getModuleName` (not yet implemented) and (C++ only) `getClass, _class, getClassname`, and static `_downcast`. In addition there are the following:

- (Java) `boolean equals(java.lang.Object o);`
(C++) `PRBool equals(const equip::runtime::Object *o)`
which returns true if the two objects are equal in value, i.e. all primitive types have the same value, strings contain the same text and member objects are also equal (using `equals()`).
- (Java) `boolean matches(java.lang.Object o);`
(C++) `PRBool matches(const equip::runtime::Object *o)`
which returns true if the two objects 'match' (defined below).
- Internal read and write object helper methods, that are used by the `ObjectInputStream` and `ObjectOutputStream` to read and write instances of the given class. These should never be invoked directly by the application programmer.

Matches

Matching is core to the EQUIP data service. The autogenerated matching code requires that:

- the (nominal) class of 'this' is the same as or a subclass of the argument object; and
- the value of each member variable of this is either NULL or 'matches' the same variable in the other.

Note: only strings or valuetype members recognize a NULL value (wildcarded). Therefore you may need to introduce additional valuetypes to support the pattern-matching process (i.e. to make those parts of the object wild-cardable).

The meaning of matches on instance elements depends on the type, as follow:

- on a valuetype this is a call to `ValueBase::matches`;
- on a string it is string equality (perhaps this should become regular expression at some point??);
- on a primitive type it is value equality (same bits);
- on a sequence it is true if `this length()==0`, else requires that lengths are equal, and applies matches to each element in turn.

Status enum

This is a standard status type, inherited from MASSIVE-3, used in a number of places in EQUIP, e.g. serialisation status codes in C++.

It is defined in the IDL file 'Equator/Modules/equip/equip_runtime/includes/eqCoreTypes.idl'. Its current defined values are:

....

(C++ only) sequence template class **Sequence<T>**

In Java just use the Java array types.

This is based on CORBA sequences, and includes the following operations:

- `void length(int l);`
change the size of the sequence to l.

- `int length();`
return the current size of the sequence.
- `T& operator[] (int i);`
return a reference to the *i*th element of the sequence.

A sequence is initialized to zero length, and automatically releases its contents on destruction (using `delete []`).

...

(C++ only) string variable class `String_var`

....

(C++ only) object variable template class `ObjectVar<T>`

See Object notes, above.

`ObjectInputStream` and `ObjectOutputStream`

See `Equator/Modules/equip/equip_runtime/include/eqStreams.h` (C++) and `Equator/Modules/equip/equip_runtime/src/equip/runtime/ObjectInputStream.java` and `Equator/Modules/equip/equip_runtime/src/equip/runtime/ObjectOutputStream.java` (Java).

See also notes on Bamboo module naming and loading for auto-loading of classes on deserialisation in the section on Bamboo (above).

....

(C++ only) basic stream classes

See `Equator/Modules/equip/equip_runtime/include/eqStreams.h`

- abstract, binary, ascii and file.

....

Basic types

See `Equator/Modules/equip/equip_runtime/include/eqBasicTypes.idl`

- `ValueBase` 'forward decl'....
- `Time`....