

equip_data_Test: an Introduction to the EQUIP data service

Chris Greenhalgh

11th April 2001.

Introduction

This document describes the sample EQUIP module 'equip_data_Test' and is intended to act as a main introduction to EQUIP and in particular the data sharing service. Supporting technical details of IDL and the runtime system (e.g. bamboo and equip_runtime) can be found in the equip_runtime documentation.

This document:

- Describes the files that make up a module;
- Describes how to compile a module;
- Traces through a sample data service client line-by-line, explaining how to use the data service (at least for simple applications); and
- Explains how to run the application as well as the EQUIP trader and data server.

Module files

The equip_data_Test module can be found in Equator/Modules/equip/equip_data_Test. It defines a new type to be shared in the data service and a Java test program that uses this type to publish and subscribe to mouse clicks in a small window. There is also simpler a C++ test element to the C++ plugin.

It comprises the following files and directories:

- **Makefile**
The module makefile, that identifies the module name ('equip_data_Test'), relative path ('.') and module version number. Can be copied and changed for any module.
- **include/**
The directory to hold the IDL and C++ header files – that must be accessible to other modules that have build-time dependencies on this one.
- **include/equip_data_Test_types.idl**
The module's IDL type definitions.
- **include/equip_data_Test_types.h**
the auto-generated C++ header declaring the C++ versions of the IDL types.
- **include/equip_data_Test_typesImpl.h**
the initially auto-generated C++ header declaring implementation classes for the IDL types (may be extended by the module programmer after generation – the previous autogenerated file should never be changed by hand).
- **src/**
The directory to hold all source files (C++ and Java).
- **src/Makefile**
The plugin makefile. See further notes below.
- **src/equip_data_Test_types.cc**
the auto-generated C++ body file defining the C++ versions of the IDL types.
- **src/equip_data_Test_typesImpl.cc**
the initially auto-generated C++ body file defining the implementation classes for the IDL types (may be extended by the module programmer after generation, e.g. to implement valuetype operations – the previous autogenerated file should never be changed by hand).
- **src/plugin.cc**
The C++ test application. Defines entry points 'int initPlugin(void)' and 'int exitPlugin(void)' which are called on module loading and unloading (the names of the entry points to call are specified in the

plugin makefile ('PLUGIN_INIT' and 'PLUGIN_EXIT') which should be 'NONE' if no explicit load/unload operation is required).

- src/DataTest.java
The Java test application.
- src/equip/data/Test/Position.java
src/equip/data/Test/PositionImpl.java
src/equip/data/Test/MyType.java
src/equip/data/Test/MyTypeImpl.java
The Java versions of the valuetypes defined in the IDL file. The '...Impl.java' can be modified by the module programmer to implement specific functionality (e.g. implement valuetype operations). Note that these files are placed in directories to reflect the Java package (and IDL module) to which they belong.
- lib/
Directory generated on module compilation.
- lib/equip_data_Test.jar
The module Java code.
- lib/winnt/
Platform-specific directory generated on module compilation to hold the DLLs/libs files.
- lib/winnt/equip_data_Test.dll
The loadable C++ module (Windows). Generated on module compilation.
- lib/winnt/equip_data_Test.lib
The linking library used with compiling other modules that depend on this one. Generated on module compilation.
- src/classes/
Directory generated on compilation; holds the Java .class files before creating the JAR file.
- src/OBJS-winnt/
Platform specific directory generated on compilation, holds the C++ object files and other intermediate files (e.g. lex/yacc generated files).

Making the module

As noted, the module makefile specifies the module name and relative path. For a top-level module the relative path is just '.'.

The plugin makefile (src/Makefile) identifies this as a C++ plugin ('PLUGIN_LANG = CPP'), with Java as a second language ('PLUGIN_OTHER_LANG = JavaLoader'). This means that Bamboo will build both the C++ DLL and the Java JAR file, but will only load the DLL if the module is specified to the bamboo command. Currently we are using the JAR and/or separate class files directly from Java, rather than using bamboo to run Java-only applications.

The plugin makefile specifies the C++ files to be compiled and included in the DLL: equip_data_Test_types.cc, equip_data_Test_typesImpl.cc and plugin.cc.

The plugin makefile specifies the other modules that this module depends on (to build and/or run). These are: equip_runtime (core runtime), equip_net (common networking) and equip_data (the data service).

As noted above, the plugin makefile identifies the module initialization and unloading entry points (in the C++ plugin.cc): PLUGIN_INIT = initPlugin, PLUGIN_EXIT = exitPlugin. These should be 'NONE' for a module with no initialization routines. Note that some initialization is performed automatically on loading/unloaded by static C++ objects (in particular, creation of class factories for IDL-specified classes).

On Windows functions must be explicitly exported from a DLL. This is supported by the module-specific compiler definition: 'COMPILE_DEFS = -DLIBEQ_DATA_TEST=1' which causes compiler definitions autogenerated by eqidl (and/or added by the module programmer) to export IDL classes and entry points from this module, so that they can be accessed from other modules.

The plugin makefile gives build rules for the auto-generated IDL files. The build rule for `equip_data_Test_types.cc` will also generate `equip_data_Test_types.h` and the Java fully auto-generated classes (so delete `src/equip_data_Test_types.cc` and `remake` to regenerate these files, e.g. if `eqidl` is updated).

The rules to make the implementation files are commented out; these are normally only used when the module is first created to generate initial versions of the implementation files. These must then be kept up to date with any changes in the IDL by the module programmer altering them by hand. There is no automated method to merge code additions by the module programmer with changes in the IDL (although CVS/RCS merge operations could be used to facilitate this).

The module can be compiled from the `Equator/Modules/equip/equip_data_Test` directory:

- `'gmake compile'`

Intermediate (build) files can be cleared out leaving a still executable module using:

- `'gmake clean'`

Everything except source can be cleared out (the module can no longer be executed) using:

- `'gmake clobber'`

IDL

The IDL file, `include/equip_data_Test.idl` is as follows (with additional notes in italics):

C++/Java-style comment

```
// sample - equip_data_Test
```

Include other IDL files from which we make use of types. The data service (ItemData) in this case.

```
#include "eqDataTypes.idl"
```

Specify the module name, which determines directly the C++ namespace and Java package name. In this case 'equip_data_Test', which MUST be the same as the module name for class auto-loading to work. Note that the `equip::data` scope is also used in `eqDataTypes.idl`, above. Here we can re-open it and add new classes and sub-packages.

```
module equip {
  module data {
    module Test {
```

A simple position type. We had to make it a separate value type so that we could do matching against `MyType` treating the actual position as a wild card. Position implicitly extends `equip::runtime::ValueBase`.

```
    valuetype Position {
```

Two instance variables. All instance variables must be 'public' or 'private'. If you forget this you will get a vague 'syntax error' from `eqidl`. Note that the type names follow C++ rather than Java, e.g. this is a 32 bit signed integer. There is no 'int' type (because it is ambiguous in C++).

```
        public long x, y;
```

End of Position valuetype.

```
    };
```

The main data type that we want to share in the data service. Extends `equip::data::ItemData`, the base class for all data service data items.

```
        valuetype MyType : :equip::data::ItemData {
            // nb position is a valuetype so that we can wildcard
```

The instance data – a single Position object. It is perfectly acceptable for this to be null (as used to represent a wild-carded match).

```
            public Position pos;
```

```
        };
```

Close the module scopes.

```
    };
};
```

End of File.

When processed by eqidl this produces the autogenerated C++ and Java files list at the start of the document. The mappings from IDL to C++ and Java are described in the equip runtime documentation.

In this application the IDL-defined types are just used to pass data about and have no specific behaviour (methods or operations), so we do not need to add anything to the default implementation files.

Java Application

The Java application is src/DataTest.java. In this section we go through it more or less line-by-line, to explain how it works, and in this way to illustrate the normal nature and operation of the EQUIP data service.

The application begins by importing the equip modules/packages that will be used:

```
import equip.net.*;
import equip.runtime.*;
import equip.data.*;
```

It also imports the Java packages and class that it will use (these may not all be used – I pasted the initial version from the data browser application):

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
```

Not surprisingly it defines the class 'DataTest':

```
public class DataTest {
```

The main class method just checks the arguments and creates an instance of DataTest to do the work:

```
public static void main (String [] args) {
    if (args.length!=3 ||
        args[0].length()!=1 ||
        (args[0].charAt(0)!='m' &&
         args[0].charAt(0)!='s' &&
         args[0].charAt(0)!='a')) {
        System.err.println("Usage: DataTest m|s|a <name> <server-
url>");
        System.err.println(" - m = master");
        System.err.println(" - s = slave");
        System.err.println(" - a = any");
        System.exit(-1);
    }
    System.out.println("DataTest...");
    new DataTest(args[0].charAt(0), args[1], args[2]);
}
```

This instance of DataTest has some key instance variables. These are a reference to the data service proxy that it will use to share data:

```
private DataProxy dataservice;
```

A reference to a GUID (Global Unique Identifier) factory (equip.net.GUIDFactory) that it will use to generate globally unique IDs for data items:

```
private GUIDFactory idFactory;
```

A reference to the generated ID that it is using to publish its own information. It needs to keep track of this in order to be able to update the information that it is publishing:

```
private GUID publishId;
```

Into the constructor itself, to initialize the application:

```
private DataTest(char kind, String name, String serverUrl) {
```

Create an instance of the utility class 'equip.net.ServerURL' to parse an EQUIP URL (which has the form 'equip://host:port/name', comparable to a Java RMI URL 'rmi://host:port/name'). This parses the serverURL argument and returns a (subclass of) 'equip.net.Moniker'. A moniker (inspired by Microsoft COM) is a serialisable object that can be 'resolved' in order to get the network address of an object or service. There are currently two main moniker types: 'equip.net.SimpleTCPMoniker' which is essentially an IP address and port number, and 'equip.net.TraderMoniker' which identifies an entry in an EQUIP name trader (see also notes on running the application, below).

```
    // resolve server URL
    Moniker serviceMoniker = null;
    System.out.println("Resolve server "+serverUrl);
    ServerURL url = new ServerURL(serverUrl);
    serviceMoniker = url.getMoniker();
```

If this comes back as null then the ServerURL object could not parse the URL.

```
        if (serviceMoniker==null) {
            System.err.println("ERROR: could not understand url:
"+serverUrl);
            System.exit(-1);
        }
```

Just as a check we get back the ServerURL's view of the url, which is actually based on the internal moniker form; this can catch some errors or misinterpretations of the url format.

```
        String checkUrl = url.getURL();
        if (checkUrl==null)
            System.err.println("Could not get back url from ServerURL");
        else
            System.err.println("ServerURL -> "+checkUrl);
```

Make a new GUID factory, so we can generate unique IDs. Note that, as with all classes defined in IDL you must ALWAYS instantiate the ...Impl subclass rather than the defined class itself. Otherwise you will not get any of the functionality of the class, and other bad things may happen. At some point I'll probably stop instantiation of the non-Impl class at all.

```
        // GUID factory
        idFactory = new GUIDFactoryImpl();
```

Create a new DataProxyImpl object, which will give us local access to the data service, and also acts locally as its own little data service.

```
        // create data proxy
        dataservice = new DataProxyImpl();
```

Set the moniker of the server to which we will connect when active.

```
        dataservice.serviceMoniker = serviceMoniker;
```

Set a default agent id. Basically, this ID will be the default 'owner' of any items that we generate. When I get round to implementing ownership of data items this will be important for ownership-based consistency.

```
        dataservice.setDefaultAgent(idFactory.getUnique());
```

If we have a valid moniker for the data server then...

```
        if (serviceMoniker!=null) {
            System.out.println("Activate...");
```

...we can activate our client. The two null arguments allow you to register a callback to be invoked if the connection to the server subsequently fails (e.g. because the server terminates or because the network connection

is lost). Activate blocks until a connection is established, and returns true on success. We could carry on running even if it failed, but we would only be sharing data with ourself.

```
    if (!dataservice.activate(null, null)) {
        System.err.println("ERROR: could not activate");
        System.exit(-1);
    }
    System.out.println("OK");
}
```

Obtaining information from the data service is almost universally based on callbacks. Creating a session (equip.data.DataSession) establishes a context within which an application can make queries of the data service and obtain callback-style notifications of results. The arguments to createSession are the callback object (an inner class, RedrawEventHandler in this case, and an object to be passed to the callback (closure or user data). This particular session is going to be used in this application to perform non-blocking checks of relevant items in the local data service when redrawing the window. The closure object is not required.

```
// redraw session
redrawSession = dataservice.createSession
    (new RedrawEventHandler(), null);
```

We create a simple (Swing) GUI...

```
createGUI();
```

If this application is started with first argument ('kind') of 'm' then this will be a 'master' application, and will publish information about mouse clicks within its window.

```
if (kind=='m') {
    System.out.println("Publish our information...");
}
```

This information will be published by creating and updating a data item in the data service that represents the last position where the user clicked in this application's window. Every data item has to have a unique ID (a GUID); we keep it in an instance variable so that we can find and update the item later on.

```
// the item that we publish
publishId = idFactory.getUnique();
```

We create an instance of the implementation class we want to add to the data base:

```
equip.data.Test.MyType item1 =
    new equip.data.Test.MyTypeImpl();
```

We set its id:

```
item1.id = publishId;
```

We initialize the rest of the instance data:

```
item1.pos = new equip.data.Test.PositionImpl();
item1.pos.x = item1.pos.y = 0;
```

We give it a unique name that will allow other applications to relate this data item to our application. Having explicitly specified names on data items supports event routing within the dataspace, creating a virtual data flow network. For example, as we will see, a slave DataTest application watches for MyType items with a specific name, and uses them to determine its own operation. Conceptually, the use of a common data service and item name establishes an implicit connection (like a virtual wire) between the master and its slave(s).

```
item1.name = name;
```

We add the item to the data service using the convenience method 'addItem'. This constructs an 'equip.data.AddEvent' object that embodies and represents the addition of the item to the data service. In this case we specify that the item should be 'HARD' locked (not yet implemented, but means that only the owner – us – is allowed to update or delete the item), process bound (i.e. to be garbage collected if this process terminates – non-process bound items are not yet supported), non-local (i.e. globally distributable) and unleased (leases are not yet supported, but should allow time-based garbage collection as in Jini).

```
// item, locked, processBound, local, lease
```

```
dataservice.addItem(item1, LockType.LOCK_HARD, true, false,
                    null);
```

Note that the data service now owns the item and we must NOT change it directly. If you change elements within item1 directly then you are essentially corrupting the database and may also be corrupting outgoing events queued for delivery to the server. See later in the program for the correct way to change an item in the data service.

End of master-specific code (publishing the data item).

```
}
```

Create a second data service session with a different callback object, also an inner class ('EventHandler'). This data session will be used to drive data distribution.

```
// callback(s)
System.out.println("Create notification session...");
session = dataservice.createSession(new EventHandler(), null);
```

Create an equip.data.EventPattern, to express what data in the data service we are interested in.

```
// ask for stuff
EventPattern pattern = new EventPatternImpl();
```

Create a template data item that will be used for matching. It is of type equip.data.Test.MyTypeImpl, which will only match subclasses of equip.data.Test.MyType (hopefully anyway; I think this works in C++ but I will have to check Java).

```
equip.data.Test.MyType item = new equip.data.Test.MyType();
```

We will not specify a position – wildcard match.

```
item.pos = null;
```

If we are a slave application then we only want to know about the data item(s) with our name; this is the implicit data/event routing noted above that implicitly 'wires' the slave to the corresponding master.

```
if (kind=='s') {
    item.name = name;
}
```

Set up the rest of the pattern as a 'simple' 'item monitor'. 'Simple' in this context means based on a single item. An item monitor is a pattern that looks for matching items being added, deleted or updated. It also returns add events for any matching items that already existed in the data service when the pattern was added, and delete events for any matching items still in the data service when the pattern is removed. This is based on EnvCallbacks in MASSIVE-3 which in turn was inspired by aura matching in MASSIVE-2 and SPLINE/OpenCommunities API.

This is a key feature of the EQUIP data service because item monitor patterns drive data distribution to clients: when we activate our data service client it will not be told anything about the data currently in the server, although it will start to forward its own data and events to the server. Then when a pattern such as this item monitor is added to the local data service it is also forwarded to the server. The server then begins to forward relevant data items and events to the local data service, which in turn generates local session callbacks.

The master's item monitor pattern is local (the second argument); this means that the pattern is not forwarded to the server and therefore the master is not informed of matching items from other clients. The other kinds of application make a non-local pattern which therefore drives replication and data distribution.

Note that the current implementation (11/4/01) does not cope well with multiple intersecting patterns. For example, if a slave added a name-specific pattern and also a name-wildcarded pattern then it would receive duplicate events for name-specific pattern (one from each pattern). For now, try not to add overlapping non-local patterns.

```
// master is local; others are not
pattern.initAsSimpleItemMonitor(item, kind=='m');
```

Add the pattern to the session – this activates the pattern, and it may generate callbacks immediately (before `addPattern` returns), as well as due to subsequent asynchronous events such as receiving relevant network messages. Some patterns, such as the item monitor are persistent, and will continue to generate callbacks and drive distribution until they are explicitly removed from the session.

```
session.addPattern(pattern);
```

Now leave the data proxy and swing threads to do their work (end of constructor).

```
System.out.println("Running...");
}
```

The GUI is a simple Swing GUI; we won't describe it all here. Suffice to say that when a user clicks on the Canvas it calls the following method:

```
private void click(int x, int y) {
    System.out.println("Click at "+x+", "+y);
```

Which checks that we have published a data item and know its id:

```
if (publishId!=null) {
    // publish
```

If so then we get the current value of the item from the data service, looking it up using the id that we kept a copy of (actually an immutable reference):

```
equip.data.Test.MyType t =
    (equip.data.Test.MyType)dataservice.getItem(publishId);
```

Create a new object of the same type, that will become the new 'value' of this data item. We have to make a new object rather than change the existing one to avoid compromising the integrity of the data base and outgoing events. **THIS IS IMPORTANT.**

At some point I will fix up the clone/copy Object/ValueBase methods to make it easier to clone a data item.

```
equip.data.Test.MyType newt = new equip.data.Test.MyTypeImpl();
```

We copy over stuff that stays the same, in particular the id and name, and replace the stuff that has changed. Note that we have to create a new Position object for the same reason as we have to create a new MyType object.

```
newt.id = t.id;
newt.name = t.name;
newt.pos = new equip.data.Test.PositionImpl();
newt.pos.x = x;
newt.pos.y = y;
```

Now we ask the data service to replace the old value in the data base. We ask for the update to be non-local (i.e. distributed) and reliable (i.e. guaranteed delivery). This convenience method packages the new item value into an 'equip.data.UpdateEvent' event which is handled directly by the data service.

```
dataservice.updateItem(newt, false, true);
```

There is a similar 'deleteItem' operation that deletes an item based on its id. This maps to an `equip.data.DeleteEvent` event.

That all for clicking.

```
}
}
```

There's a routine to draw an X on the screen; we'll skip that.

Here are the sessions that are used for change notifications and redraw checks:

```
private DataSession session;
private DataSession redrawSession;
```

We have an inner instance class, `EventHandler`, that extends `DataCallback`. It has to extend it because `DataCallback` is a valuetype, could have instance state, and itself extends `equip.runtime.ValueBase`. For SWING

we also implement Runnable to allow synchronization with the Swing event thread (Swing, unlike AWT, is NOT thread-safe – see the Swing documentation).

```
private class EventHandler extends DataCallback implements Runnable
{
```

We implement the DataCallback ‘notify’ method; this will be called for each matching event on the associated session. In this application, this session has the item monitor pattern, i.e. notify will be called each time a matching item is added, deleted or updated.

```
public void notify(equip.data.Event event, EventPattern pattern,
                  boolean patternDeleted,
                  DataSession session,
                  ValueBase closure) {
    System.out.println("notify...");
```

For Swing thread-safeness we take a copy of the notification parameters and ask Swing’s event thread to call our run method as soon as possible.

WARNING: I’ve just noticed that this is now broken; there could be multiple notify involution queued on the Swing event thread, so the instance data could be over-written. We should create a new instance of a new class for each invocation to do this safely. It doesn’t make any difference here as our only response is to redraw.

```
// copy notify information into runnable for sync with Swing
// thread (for rendering thread-safe-ness)
this.event = event;
this.pattern = pattern;
this.patternDeleted = patternDeleted;
this.session = session;
this.closure = closure;
try {
```

Get the Swing thread to execute run() in a safe fashion.

NOTE: If we were not using Swing then we would NOT do this; just more the active code from run() straight into the notify method and respond immediately (see the RedrawEventHandler later in the file).

```
if (!SwingUtilities.isEventDispatchThread())
    SwingUtilities.invokeLater(this);
else
    run();
} catch(Exception e) {
    System.err.println("notify error: "+e);
}
```

That’s it for the notification:

```
System.out.println("...done");
}
```

The data we are passing to run():

```
private equip.data.Event event;
private EventPattern pattern;
private boolean patternDeleted;
private DataSession session;
private ValueBase closure;
```

Run() does the actual work of responding to the event in Swing:

```
public void run() {
    if (event!=null)
        System.out.println("(sync) event "+event);
    ignoreId = null;
```

For example, check if it is a delete event, and if so extract the id of the item being deleted. The application isn’t actually using this at the moment, but hopefully you can see the principle.

```
if (event!=null && event instanceof DeleteEvent) {
```

```

        System.err.println("delete...");
        DeleteEvent del = (DeleteEvent)event;
        System.err.println("ignore deleting item "+del.id);
        ignoreId = del.id;
    }
    // synchronized -> after the session event that is causing
    // this...

```

All this program does is lock the session (to reentrant calls to notify due to concurrent external events) are ask Swing to redraw the area Canvas (created in the GUI stuff we skipped over).

```

        synchronized(session) {
            // redraw all known items
            area.repaint();
        }

```

That's the end of this notification handler method and class.

```

    }
}

```

We have our own canvas class to draw Xs corresponding to all of the MyType items that this client knows about.

```

class MyPanel extends Canvas {

```

Swing (and AWT) call paint to ask us to update the screen:

```

    public void paint(Graphics g) {
        System.out.println("redraw...");
    }

```

We create a new pattern...

```

        // Generate synchronous callbacks for all known local items
        // of the given type.
        // The redrawSession will call RedrawEventHandler, which
        // will actually render each one.
        // A copy-collect is the appropriate model here - the pattern
        // is auto-deleted after checking.
        EventPattern pattern = new EventPatternImpl();

```

...to match any items of type equip.data.Test.MyType...

```

        equip.data.Test.MyType item = new equip.data.Test.MyTypeImpl();

```

...but rather than an item monitor, this time we make in a copy collect pattern. Like in tuplespaces, this will return all items that match the given pattern. This pattern is local only, and so will not be relayed to the server.

NOTE: I haven't actually implemented distributed tuple operations yet anyway, so non-local wouldn't work.

This is the recommended way to find currently known data items. Note the interaction with the item monitor pattern already seen: that pattern caused item replication (if appropriate); this pattern just polls the data base at this moment to see what is there. A copy collect EventPattern only looks for add events (i.e. items present in the data service).

```

        pattern.initAsSimpleCopyCollect(item, true);

```

Add the copy-collect pattern, causing it to generate notifications on the redrawSession for every matching item currently in the local data service. As we will see, each notification will cause the application to draw an X. Note that the copy collect pattern also deletes itself as soon as it has checked the database, and so does not need to be explicitly removed. All notifications for a local copy-collect such as this will happen on the calling thread before the addPattern method returns (which is therefore the Swing event thread in this application, and so safe to do Swing operations).

```

        // do it
        System.out.println("redraw add pattern...");
        redrawSession.addPattern(pattern);
        // done
        System.out.println("redraw done");

```

That's the end of the redraw.

```
    }  
}
```

This notification handler is also an inner instance class:

```
private class RedrawEventHandler extends DataCallback {
```

In this application this notify method will only be called from the copy collect pattern just seen in the canvas paint method. Each invocation should therefore be an AddEvent event corresponding to a matching item in the local data service.

```
public void notify(equip.data.Event event, EventPattern pattern,  
                  boolean patternDeleted,  
                  DataSession session,  
                  ValueBase closure) {  
    System.out.println("redraw notify...");
```

Check that it is an equip.data.AddEvent event; if not we have nothing to do:

```
    if (event==null || !(event instanceof AddEvent))  
        return;
```

Look inside the add event to find the actual data item to which it refers. Check that the item is really of type equip.data.Test.MyType. Otherwise we have nothing to do for it (just being careful).

```
    System.err.println("add...");  
    AddEvent add = (AddEvent)event;  
    if (add.binding.item==null ||  
        !(add.binding.item instanceof equip.data.Test.MyType))  
        return;
```

Pull out the value of the data item...:

```
    equip.data.Test.MyType t =  
        (equip.data.Test.MyType)add.binding.item;
```

...and draw an X at the position that it specifies.

```
    drawPoint(t.pos.x, t.pos.y);
```

And that is the end of our notify method and inner class:

```
    }  
}
```

And the end of the whole test application:

```
}
```

Running the application

To run this application you need to also run the EQUIP Trader and at least one EQUIP data server. You can then run any number of instances of the test application and see how they communicate through the data service.

Starting the EQUIP Trader

The EQUIP trader is in (C++ only) module 'equip_net_Trader'. Once you have configured your environment to run Bamboo and the EQUIP modules (see Equator/buildNotes.cmg and Equator/Modules/bbenv.bat) then you should be able to start the trader using Bamboo:

```
bamboo ./equip_net_Trader
```

The default trader port is currently 7883, but may change if major changes are made to low-level networking. An optional argument to the equip_net_Trader module allows an alternative port to be specified explicitly, e.g. port 9123:

```
bamboo ./equip_net_Trader -9123
```

Note that the '-' tells bamboo that '9123' is an argument for the equip_net_Trader module, rather than another module (called '9123').

Bamboo will print a number of messages (from the Object factories) about known classes and then give diagnostic output something like:

```
equip_net_Trader: hello
Plugin equip_net_Trader appears to have been loaded as
./equip_net_Trader
No port specified...using default
Starting trader on port 7883...
Starting TraderServer
ThreadedServerSocket created on port 7883
SimpleServer port = 7883
SimpleServer created on port 7883
-> port 7883
SimpleServer::serverThread running...
```

Indicating that it is running OK and is using port 7883.

Starting the EQUIP data server

The current EQUIP data service server is in (C++ only) module 'equip_data_Server'. Once the trader is running, in another shell you should be able to start the data server using Bamboo:

```
bamboo ./equip_data_Server -equip:///Foo
```

The 'equip:' URL will default to the local machine and to the default trader port (currently 7883). Alternatively, the trader machine and port may be specified explicitly as in:

```
bamboo ./equip_data_Server -equip://foo.mrl.nott.ac.uk:9123/Foo
```

The name after the last '/' is the name with which the data server will register with the trader ('Foo' in this example). Use of 'equip:' is very similar to the use of Java 'rmi:' URLs. The EQUIP trader has a flat namespace and no federation facilities at present.

The data server, after various factory notifications (from static class factories) should print diagnostics something like:

```
equip_data_Server: hello
Plugin equip_data_Server appears to have been loaded as
./equip_data_Server
Starting data server...
ThreadedServerSocket created on port 3916
DataDelegate::DataDelegate created with id
[128.243.22.35:0.3916:0:986985560]
Starting DataServer
ThreadedServerSocket created on port 3917
DataServer port = 3917
DataServer created on port 3917
-> port 3917
TraderMonikerImpl::rebind...
DataServer::serverThread running...
ProxyDelegate::activate() ok
OK
```

You can see that the server is actually running on port 3917, but you would normally access it via the Trader using an 'equip:' URL. However, if you wish to avoid the trader you can also start the data server on a specified port without using the trader at all, for example:

```
bamboo ./equip_data_Server -equip://:9123/
```

Because no name is specified ServerURL takes the 'equip:' URL to specify a SimpleTCPMoniker directly, so that the server tries to start on port 9123 (in this example).

Starting the test application

Currently Java EQUIP applications are run directly from java, rather than Bamboo; this gives greater flexibility for cross-platform deployment, e.g. on handheld devices that do not support Bamboo.

At present this requires that Java EQUIP applications have the Java CLASSPATH fully specified in advance; I will sort out dynamic name to package translation at some point. The two main ways to get everything runnable from Java are:

- copy the class files from all of the required modules (from the src/classes directory, or extracted from the .jar files) to a common directory, which can then be placed in classpath; or
- explicitly including all of the required modules' .jar files in CLASSPATH, e.g.

```
set CLASSPATH =
J:\cmg\dev\Equator\Modules\equip\equip_runtime\lib\equip_runtime.jar;
J:\cmg\dev\Equator\Modules\equip\equip_net\lib\equip_net.jar;J:\cmg\dev\Equator\Modules\equip\equip_data\lib\equip_data.jar;J:\cmg\dev\Equator\Modules\equip\equip_math\lib\equip_math.jar;J:\cmg\dev\Equator\Modules\equip\equip_data_Test\lib\equip_data_Test.jar
```

(See also Equator/Modules/bbenv.bat.)

You should then be able to run a master client with item data name 'Bar' with the above data service on the local machine as:

```
java DataTest m Bar equip://localhost/Foo
```

If successful this will open a small window with an X in the top left hand corner. When you click in the window the X will move to that position, and the data server will also print some diagnostics.

When you first start the DataTest client the data server will print (amongst other things) something like:

```
FactoryBase::findFactory failed for equip.data.Test.MyType:
- trying module ./equip_data_Test (using EQUIP_MODULE_PREFIX ".")...
- load...
FactoryBase::addFactory equip.data.Test.Position
FactoryBase::addFactory equip.data.Test.MyType
equip_data_Test: hello
Plugin equip_data_Test appears not to have been loaded from the
command line - a
suuming no args
ERROR: Usage: equip_data_Test [-<dataserverurl>]
OK - retrying class
```

This is telling you that the data server received an instance of the unknown (to it) class equip.data.Test.MyType. It then attempted to load the module './equip_data_Test', determined by prepending the value of the environment variable EQUIP_MODULE_PREFIX to the package name with '.' changed to '_'. In this case that succeeded, and as it loaded the factories for equip.data.Test.Position and equip.data.Test.MyType were found. These were then used to de-serialise the incoming object. I.e. dynamic code loading in C++ ☺

Try starting another (slave) client using:

```
java DataTest s Bar equip://localhost/Foo
```

It should show one X in the same place as the master of the same name.

Try starting another ('any') client using:

```
java DataTest a Bar equip://localhost/Foo
```

It should show Xs for current masters, whatever their name.

Try starting additional masters and/or slaves and/or any-monitors. Check that slaves and monitors only show Xs for current master applications, and that it doesn't matter if the master or the slave is started first and/or restarted, they still join up once they are both running.

That is the idea, anyway. If you get consistent bugs then check for software updates, announcements of known problems, etc. and in the last recourse email cmg@cs.nott.ac.uk with a clear bug report so I can archive it in my incoming mail box...

