This wiki describes how to automatically generate C++ classes defining the message formats for communicating and sending information between modules in the DARWIN cognitive architecture.

===

Two Minute Overview

Problem: yarp::os::Bottle is not type-safe at compile time.

Example: one module adds 2 strings to a yarp::os::Bottle: a key/value pair ("colour","red"), and then writes it to a Port. The receiving module mistakenly tries to find the value for key "COLOR". Sender and receiver will not discover their mismatch until run time, when the receiver is not getting the data expected.

Solution: provide strict access functions to make Bottles type-safe at compile time.

The auto-generated C++ classes provide strict access functions to the data inside Bottles. In the example above the sender will call setcolour("red"), whereas the receiver will call colour() on the Bottle to retrieve the value of the data field. If the receiver tries to call COLOR(), the compiler will report an error. When both sender and receiver modules use the same auto-generated C++ classes, the compiler will flag almost all access mistakes. The few remaining run-time access mistakes can be avoided by adopting simple strategies.

Define the structure of C++ classes with XML.

Snippets of XML define the structure of each message class. The C++ classes with all the correct access functions are automatically generated from the XML definitions. It is very easy to define complex message formats by composition from simpler building blocks. Examples follow below.

Generate C++ code using XSLT.

The XML code is transformed into C++ using an XSLT stylesheet. The transformation is done automatically by the build system (i.e., make or VS Build). Alternatively, you can manually generate C++ code in a variety of XSLT-aware tools: e.g., Visual Studio or FireFox. As long as the XML file you write is valid, the generated C++ code is stable and bug-free.

Easy to use C++ message classes

All generated message classes live in 1 header file. There is only 1 single file to include so that senders and receivers can be certain they are using the same message formats. The generated classes are simple and easy to use. Unlike the standard Bottles, where you need to remember what data you agreed to put into a Bottle and how to name it, development environments with their code completion capabilities will remind you how to set and get the data from the message class.

Easy to maintain XML code

XML is simple text, so in principle can be maintained with a text editor. However, what is valid and invalid XML code is defined by an XML schema file. When using a good XML editor, this schema definition (which is automatically loaded by the file you are editing) helps you to write valid XML code, offering error checking and code completion as you write. On Windows, you can use Visual Studio. On Linux, both Emacs and Vim offer XML extensions.

Implementation.

Under the hood all message classes are implemented as yarp::os::Bottle, so generated message classes are entirely compatible with Bottle's reading/writing functionality over Ports and BufferedPorts. There are a number of hand-written C++ base classes that provide the core functionality for the automatically generated classes. You don't need to worry about those. There is an XSLT spreadsheet providing the translation from XML to C++. You don't need to worry about that either. The generated C++ code does not need to be modified. All generated C++ code will be inlined by the compiler, so there is no performance overhead. The only thing to maintain is the XML file. As long as this file is valid, the C++ code generated from it will be correct.

===

Including DARWIN MessageFormats into Yarp Modules using CMake

1. Check out **darwin/trunk/common** from the repository. There is only 1 file to edit for all partners: **common/MessageFormats/DarwinMessages.xml,** and only 1 auto-generated header file to include in your own project: **common/MessageFormats/DarwinMessages.h**.
2. Add the following lines to the CMake file of your own modules:

```
#Find the path to the darwin/trunk/common folder
find_path(DARWIN_COMMON MessageFormats/DarwinMessages.xml
          HINTS "${CMAKE_CURRENT_SOURCE_DIR}/../../common")

#Include the CMakeLists files that exist in darwin/trunk/common
#These will generate 2 targets: MessageFormats and TypeSafeBottle
include("${DARWIN_COMMON}/CMakeLists.txt" NO_POLICY_SCOPE)

#Set your module target to be dependent on MessageFormats so that it builds
later
#You must put this command AFTER your own add_executable or add_library
command
add_dependencies(${KEYWORD} MessageFormats)
```

The above snippet assumes that your module CMakeLists.txt file resides in **darwin/trunk/modules/$PARTNER/**. Adjust the number of ../ elements in the above HINTS string accordingly if this is not the case. The first 2 statements can live in your upper most CMakeLists.txt file, or in a file for 1 specific module. The last statement needs to be specified in every module CMakeLists.txt file AFTER the add_executable statement.

In your own C++ code, use the following include directive and namespace declaration in the relevant locations:

```
#include "MessageFormats/DarwinMessages.h"
using namespace darwin::msg;
```

===

How to Add New Message Formats to the DARWIN Repository

1. Check out **darwin/trunk/common** from the repository.
2. Open the XML file **common/MessageFormats/DarwinMessages.xml** and edit.
3. When you compile/build your project, the C++ header file will automatically be updated by the build system. Alternatively, you can perform the transformation manually using an XSLT tool such as Visual Studio or Firefox browser (see below).
4. Never edit the C++ header file manually.

===

XML Editors

On Windows

Visual Studio provides support for editing XML. Simply open the XML file from your project, then edit. You should see code completion helpers and the editor will warn you when the file is invalid.

On Linux

Emacs and Vim provide support for editing XML files.

===

XSLT: XML to C++ Transformation

During Build

On both Windows (tested with Visual Studio) and Linux (tested with make), the build process automatically updates the C++ header file when the XML file has been edited.

On Windows

VS also allows you to perform the transformation manually. When done editing the XML file, select menu "XML" at the top (warning, your XML file needs to be active for the menu item to exist). Then select "Start XSLT without debugging", and this will generate a new file with C++ code. Save this text file in the correct location in your repository or copy/paste all the code into the correct file.

On Linux

Performing the transformation manually can be done by simply opening the XML file in a web browser such as Firefox (see next point). On the command line you can use **xsltproc**. For instance, in the folder common/MessageFormats, execute:

```
xsltproc -o DarwinMessages.h DarwinMessages.xslt DarwinMessages.xml
```

Web Browsers

In principle. any browser is capable of transforming XML into C++ using XSLT. When done editing the XML file, simply open the file inside a browser and you will see C++ code. Copy and paste all the code into the header file, and that's it.

Due to security restrictions not all browsers allow this XSLT transformation. Currently, the following browsers have been tested:

- Firefox: **YES**
- Chrome: **NO**

To make this work with Firefox, the repository uses "svn:externals" which means that an additional copy (under svn control) of the TypeSafeBottle folder appears in your DarwinMessages folder. Normally the 2 files in this folder do not need to be edited.

===

Tutorial Code in the DARWIN Repository

Check out **darwin/trunk/moduleTutorials/TypeSafeTutorial** for a working example with CMakeList files to build an executable, a cpp file showing how to use the code (with comments), and an example XML and auto-generated header file.

===

A Simple Example

Message formats are defined in XML. Here are a few simple examples:

```
<!--each className must be a unique identifier within the whole document-->
<Struct className="Point2D">
   <!--each name bust be unique within this Struct-->
   <!--double is a builtin type-->
   <Variable name="x" type="double"/>
   <Variable name="y" type="double"/>
</Struct>
<Struct className="Circle">
   <!--a class can contain variables from another class-->
   <Variable name="Center" type="Point2D"/>
   <!--int is another builtin type-->
   <Variable name="Radius" type="int"/>
</Struct>
<!--a vector of arbitrary length of Point2D elements-->
<Vector className="Polygon" valueType="Point2D"/>
<!--Wrap either a Circle OR a Polygon element into another message-->
<Wrapper className="Shape">
   <!--each subID is a string from length 1-4, unique within this Wrapper--
>
   <!--it defines the type of the content of the message-->
   <Bottle subID="circ" valueType="Circle"/>
   <Bottle subID="poly" valueType="Polygon"/>
</Wrapper>
```

===

XML Language Specification

Each top-level element in the XML specification defines a C++ class. Each of these needs a unique "className" attribute that becomes the name of the C++ class.

Three message types (Bottle types)

1. <Vector>: a message with arbitrary number of unnamed elements, all of the same type.
2. <Struct>: a message with known name and type for its data fields.
3. <Wrapper>: a message that wraps around other messages, to send different message formats over the same Port/BufferedPort.

<Vector> in detail

1. When defined at the top level, each <Vector> defines a type which contains an arbitrary number of elements of the same type.
2. The type of contained elements is specified in attribute "valueType", which can be either a builtin type or refer to a className in the same XML document.

```
<Vector className="Point2DVector" valueType="Point2D"/>
```

<Struct> in detail

1. Each <Struct> can contain an arbitrary number of <Variable> and/or <Vector> elements, each with a "name" attribute that is unique within this <Struct> element.
2. Each <Variable> has a "type" attribute that is either a builtin type (see below) or refers to a className in the same XML document.
3. Each <Vector> has a "valueType" argument that is either a builtin type (see below) or refers to a className in the same XML document. A Vector defined at this level can be used in the same way as a Vector defined at the top level, but will not result in a named type.

```
<Struct className="Mixture">
   <Vector name="anIntVector" valueType="int"/>
   <Variable name="aDouble" type="double"/>
   <Variable name="aString" type="string"/>
   <Variable name="aVocab" type="vocab"/>
   <Variable name="aPoint" type="Point2D">
   <Vector name="aPointVector" valueType="Point2D"/>
</Struct>
```

<Wrapper> in detail

1. Each <Wrapper> contains an arbitrary number of <Bottle> elements, message formats that can be wrapped by this type.
2. Each <Bottle> element has a "subID" attribute which is unique to the parent <Wrapper> element. It is a string of length 1-4.
3. Each <Bottle> element has a "valueType" attribute which refers to a className in the same XML document (valueType in this case *cannot* be a builtin type). The valueType does not need to be unique within the parent <Wrapper>. It is possible to specify sub-messages with the same valueType but different subID (if they need to be processed differently by the receiver).

```
<Wrapper className="Anything">
   <Bottle subID="mix" valueType="Mixture"/>
   <Bottle subID="pnt1" valueType="Point2DVector"/>
   <Bottle subID="pnt2" valueType="Point2DVector"/>
</Wrapper>
```

Builtin types

Currently, the supported builtin types are:

- int
- double
- string (translates to yarp::os::ConstString)
- vocab (translates to int, but has also a short string representation)

Builtin types, supported by yarp::os::Bottle, for which support will be added shortly:

- boolean
- property (translates to yarp::os::Property)
- blob (binary data)

Messages consisting of single variables of one of the builtin types (e.g., int or double)

If you want to send a single builtin variable across the network with the least amount of communication overhead, define the message as a <Vector>. This is the exact equivalent of sending a Bottle that contains only a single value. Alternatively, you can also define a <Struct> with a suitable name for the variable. It provides more type-safety and certainty about what is inside the Bottle, but it means that the name of the variable will also be part of the Bottle (as yarp::os::Bottle often sends data as key/value pairs for human readability).

===

C++ Code: Class Interfaces and Use

This is what the interfaces of the auto-generated classes for the example above look like.

```
#include "TypeSafeBottle/TypeSafeBottle.h"

using namespace darwin::msg;

//pre-declaration of classes derived from StructBottle
class Point2D;
class Circle;

//pre-declaration of classes derived from WrapperBottle
class Shape;

//named VectorBottles - implemented as typedefs
typedef VectorBottle<Point2D> Polygon;

//class declarations derived from StructBottle
class Point2D : public StructBottle {
public:
   Point2D();
   double x();
   double y();
   Point2D& setx(double);
   Point2D& sety(double);
};

class Circle : public StructBottle {
public:
   Circle();
   Point2D& Center();
   int Radius();
   Circle& setRadius(int);
};

//class declarations derived from WrapperBottle
class Shape : public WrapperBottle {
public:
   Shape();
   Circle& circ();
   Polygon& poly();
   void circ(const Circle&);
   void poly(const Polygon&);
   typedef SubID2Type<Circle,VOCAB4('c','i','r','c')> circID;
   typedef SubID2Type<Polygon,VOCAB4('p','o','l','y')> polyID;
};
```

The following C++ code demonstrates how you would use the classes generated from this XML:

```
//let's prepare some messages for sending
//create a point with x and y coordinates (see XML above)
Point2D p1;
//you can chain "set" operations together
p1.setx(10.0).sety(20.0);
//create a circle and set its center to p1 and radius to 5
//all access functions are generated from the XML fields defined above
Circle c1;
c1.Center() = p1;
c1.setRadius(5);
//create a few more points
Point2D p2; p2.setx(1.0).sety(2.0);
Point2D p3; p3.setx(30.0).sety(30.0);
//a polygon is a vector of Point2D objects
Polygon t1;
//add 3 Point2D objects to this polygon: a triangle
t1.add(p1).add(p2).add(p3);
//we want to send both circles and polygons over the same Port
//so we'll use a Wrapper
Shape s1;
//first we wrap the circle into s1, and write to the port
s1.circ(c1);
//"port" is a yarp::os::Port opened before
port.write(s1);
// ... do something else
//now we wrap the polygon into s1, and write to the port
s1.poly(t1);
port.write(s1);

//the receiver reads a message of type Shape
//"port" is a yarp::os::Port opened before
Shape s2;
port.read(s2);
//write out a string representation of the message (including subID string)
```

```
cout << s2.toString() << endl;
//how do we know if we just read a Circle or a Polygon object?
//remember that C++ doesn't allow overloading on only the return type of a
function
//so we must specify separate access functions for both wrapped message
types
//the message contains an integer subID specifying what message format was
wrapped
//this subID integer is derived from the XML subID string using
yarp::os::Vocab
switch (s2.subID()) {
   //case label is an integer class constant defining a "circ" Circle
element
   case Shape::circID::value: {
      //use a reference to prevent a copy of the entire message
      Circle& c2 = s2.circ();
      cout << "Radius: " << c2.Radius() << endl;
      cout << "Center: " << c2.Center().toString() << endl;
      break;
   }
   //case label is an integer class constant defining a "poly" Polygon
element
   case Shape::polyID::value: {
      //use a reference to prevent a copy of the entire message
      Polygon& t2 = s2.poly();
      //write all corner coordinates of the polygon
      for(int i=0;i<t2.size();++i) {
         cout << "Corner " << i << ": " << t2[i].toString() << endl;
      }
      cout << "The x coordinate of the last corner: " << t2[2].x() << endl;
      break;
   }
}

===
```

Generated C++ Code: Full Interface Definition

Common operations for all TypeSafeBottles

- **read()**: the read function to read data from a port
- **write()**: the write function to write data to a port
- **toString()**: generate a string representation of the Bottle contents
- **operator==():** compare the content of 2 Bottles
- **operator!=():** compare the content of 2 Bottles

Classes derived from VectorBottle (templated on typename T, where T can be a builtin type or a proper class)

- **clear()**: clear all the data
- **size()**: get the number of elements
- for builtin typenames int, double, string and vocab: **T operator[](int)**: return the *value* of index i; provides read-only access.
- for other typenames T: **T& operator[](int)**: returns a *reference* to element i, provides read and modify access.

- **VectorBottle<T>& add(const T&)**: append an element to the back of the vector. The function returns a reference to the object itself so add operations can be chained together.
- **VectorBottle<T>& set(int,const T&)**: set element i to a given value. Provides set access, especially needed for builtin types. Returns a reference to the object itself so set operations can be chained together.
- The VectorBottles templated on builtin typenames have typedef declarations: **IntVector, DoubleVector, StringVector, VocabVector.**
- **Exceptions**: the access functions **operator[](int)** and **set(int,const T&)** throw an out-of-range exception when you try to access an element at an index that doesn't exist.
- **Strategy to prevent run-time exceptions:** ensure you always test the index against the length of the vector using the size() function.

Classes derived from StructBottle

- for each variable of builtin type, there are **get/set** functions. For instance, "**double x()**"/"**Point2D& setx(double)**" in the interface of class Point2D. The set function returns a reference to the object itself so set operations can be chained together.
- for each variable field that is a proper class and for each vector field, there is **1 access function** that returns a reference to the contained object. For instance, **Point2D& Center()** in the example above. This allows both get and set access for individual fields. E.g., **c1.Center.setx(p1.x())**. Assignment is also allowed: **c1.Center() = p1** (this copies all elements of p1 to c1.Center).

Classes derived from WrapperBottle

- **subID()**: returns an int representation of the subID of the currently wrapped message.
- **subIDStr()** returns a string representation of the subID of the currently wrapped message.
- for each subID, there are **get/set** access functions of the type **Circle& circ()/void circ(const Circle&)**.
- for each subID, there is a complicated-looking **typedef** declaration. They exist to force the compiler to generate template functions in the base class, so don't worry too much about those.
- There is only one reason why you may need these typedefs (and why they are public): each typedef gives you access to a "value" field that holds the int value of a subID (which in the XML file was specified as a string of length 1-4 - the internal translation uses yarp::os::Vocab functionality). For instance: **Shape::circID::value** for the "Shape" class declared above.
- **Exceptions**: if you are trying to read the wrong subID type from a wrapper object, an exception will be thrown. Because C++ doesn't allow you to overload function return types, there's no type-safe way to solve this problem and throwing an exception is the cleanest thing to do.
- **Strategy to avoid run-time exceptions**: the following snippet of code shows you how to do this for the example of the Shape class defined above.

```
Shape s1;
port.read(s1);
switch (s1.subID()) {
    case Shape::circID::value: {
        //here it's safe to use the circ() function
        Circle& c1 = s1.circ();
        //insert your handler code for circ messages
        break;
    }
    case Shape::polyID::value: {
        //here it's safe to use the poly() function
        Polygon& p1 = s1.poly();
        //insert your handler code for poly messages
        break;
    }
}
```