How C++ Objects Can Hold XML Info That's Going To Change
Submission for OT2000 workshop on "XML and Objects" by Detlef Vollmann

Motivation

Information in XML docs complies to "Document Type Definitions (DTD)". But an important aspect of XML is its eXtensibiltiy, i.e. there is no fix set of DTDs, and DTDs itself are not fix. Quite probably, in the course of the life-time of a DTD, it will probably evolve and be modified to reflect changes in the application domain. If the information in the XML doc is stored in objects of classic typed OO languages like C++ or Java, using a simple mapping mechanism, that C++ (or Java) classes must be changed as soon as the DTD changes. As this is not acceptable in some environments, another solution must be found. Here I will present some possible solutions for C++.

Static Classes

As a running example, let's take the following DTD:

```
<?XML version="1.0"?>
<!DocType Address [
<!Element Address    (Street, City)>
<!Element Street     (#PCDATA)>
<!Element City       (#PCDATA)>
]>
```

In C++, a corresponding class definition might look like this:

```
class Address
{ public:
        string street, city;
};
```

Quite simple. Now you might need to change the DTD to allow for house name or to allow for a P.O. box or to separate the phone number into a region code and the local number. Now, your new DTD looks like this:

```
<?XML version="1.0"?>
<!DocType Address_1_1 [
<!Element Address    (POBox?, Street, City)>
<!Element POBox      (#PCDATA)>
...
]>
```

And your corresponding C++ class now looks like this:

```
class Address
{ public:
        string pOBox, street, city;
};
```

Still simple. But you have to change your application program.

Meta Data

A different approach that is much more flexible, is based on meta data.
A definition for the meta data in C++ might look like this:

```
class ClassDef
{ public:
        ClassDef(const ClassDef *base, const string & name_);
```

```
        Object * newObject() const;
        AttrIteratorPair getAttributes() const;
        void addAttribute(const Attribute &);
        Attribute const & findAttribute(string const & name) const;
    private:
        ClassDef const * const baseClass;
        string name;
        AttributeContainer ownAttributes, effectiveAttributes;
};
class Object
{ public:
        explicit Object(ClassDef const * class_);
        BaseValue & getValue(string const & name) const;
        void setValue(string const & name, BaseValue const &);
    private:
        ClassDef const * const myClass;
        vector<BaseValue *> values;
};
```

Now, the definition of the address "class" looks like this:
Defining the class:

```
ClassDef * address
        = new ClassDef(0, / / no base class for Address
                "Address"); / / name of class
```

Adding attributes:

```
address->addAttribute(Attribute("City",
                Type::getType(Type::stringT)));
```

And here you have an object:

```
Object * reto(book->newObject());
reto->setValue("Street",
        Value<string>("Berner Strasse"));
/ / ...
```

Now, how to adapt to the changed DTD? There are two options:
1) add an attribute to the original class (not recommended):

```
address->addAttribute(Attribute("POBox",
                Type::getType(Type::stringT)));
```

2) define a derived class:

```
ClassDef * address11
        = new ClassDef(address, / / no base class for Address
                "Address_1_1");
address11->addAttribute(Attribute("POBox",
                Type::getType(Type::stringT)));
```

This can easily be done at runtime through a nice GUI. So you're much more flexible. But now, all your object access must go through the meta-object protocol (MOP), even for classes you know at compile time.

Combining Meta Data and Existing Classes

An approach that combines the flexibility of meta-data with the ease of use of the static class approach must allow a MOP access to the static classes. While Java provides a reflection interface, C++ has pointer to members for this purpose.

To work, the MOP needs of all classes to know

- names of the members (this is no problem)
- types of the members (also no problem)
- access to the members (via pointer-to-member).

Sorry for omitting all the details, but I wanted to fit this on two pages (and didn't want to show too low-level C++ code.

For details, just mail me (dv@vollmann.ch).