MAUS API Framework and Data Structure

Alexander Richards

Imperial College Sci., Tech. & Med. UK
(IC)

18th Oct 2012

Science & Technology Facilities Council
Rutherford Appleton Laboratory

MICE Collaboration Meeting 34, R.A.L.

Contents

1 API

2 Data Structure

API Framework - Intro

http://www.ehow.com/info_12227143_api-characteristics.html

In essence, the API framework consists of one base object type *Module* and four derived object types.

API Framework Objects (Modules)

Input

Getting data into the software chain. (MC/file etc.)

Map

Transforms the data on each spill. (no internal state)

Reduce

Reduces data by combining multiple splills. (internal state)

Output

Outputs data. (persistent file/web server etc.)

API Framework - Basics

Objects

For each type of object we have:

- An interface (pure virtual) class that defines the minimal implementation required for this object
- An abstract base class that implemements the corresponding interface but adds a level of abstraction.

- At present the abstraction layer is used to perform certain error handling
- The approach is scalable however and allows for any further abstracted behaviour
- Seperation of interface from abstraction means that developers are not forced to utilise the abstracted behaviour

Everything Starts With a *Module*...

The Module is the base object from which all others inherit. It provides minimal functionality and abstraction.

Module Interface - IModule

```
class IModule {
  public:
    virtual void birth(const std::string&) = 0;
    virtual void death()                   = 0;
};
```

- *birth* function takes a config string and initialises the module
- *death* function cleans up before destruction

Adding Abstraction

Abstract Module Base Class - ModuleBase

```
class ModuleBase : public virtual IModule {
  public:
    // Constructors & Destructor
    explicit ModuleBase(const std::string&);
    ModuleBase(const ModuleBase&);
    virtual ~ModuleBase();

  public:
    void birth(const std::string&);
    void death();

  protected:
    std::string _classname;

  private:
    virtual void _birth(const std::string&) = 0;
    virtual void _death()                   = 0;
};
```

Interfaces

In addition we also have the following interfaces built upon Module:

API Interfaces

```
template<typename T>
class IInput  : public virtual IModule {
  public:
    virtual T* emitter() = 0;
};
template<typename T>
class IReduce : public virtual IModule {
  public:
    virtual T* process(T* t) = 0;
};
template<typename T>
class IOutput : public virtual IModule {
  public:
    virtual bool save(T*) = 0;
};
```

As with the IModule, also have corresponding abstract base classes.

Developer Implementation

Minimal Implementation

```
class MyInput : public InputBase<Spill> {
  public:
    explicit MyInput(const std::string& s) :
        InputBase<Spill>(s) {}
    MyInput(const MyInput& m) : InputBase<Spill>(m) {}
    virtual ~MyInput() {}

  private:
    virtual void _birth(const std::string& s) {
      // Your initialisation code here
    }
    virtual void _death() {
      // Your finalisation code here
    }
    virtual Spill* _emitter() {
      // Your emitter code here
    }
  };
```

Mappers

Mappers are slightly different...

Mapper Interface

```
template <typename INPUT, typename OUTPUT>
class IMap : public virtual IModule {
  public:
    virtual OUTPUT* process(INPUT*) const = 0;
};
```

- Note the const as mappers have no internal state.
- Specification for mappers required asymmetry in the type of input/output.
- Additionally specification desired ability to dynamically convert to expected input type in an abstracted way to remove burden from developer.

Mappers

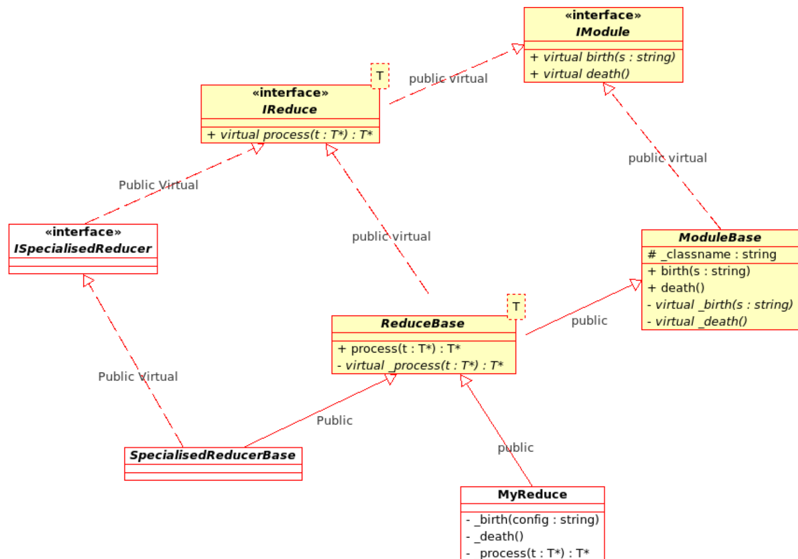
Abstract Mapper Base Class - MapBase

```
template <typename INPUT, typename OUTPUT>
class MapBase : public virtual IMap<INPUT, OUTPUT>,
                public ModuleBase {
  public:
    // Constructors & Destructors
    explicit MapBase(const std::string&);
    MapBase(const MapBase&);
    virtual ~MapBase();

  public:
    OUTPUT* process(INPUT*) const;
    template <typename OTHER> OUTPUT* process(OTHER*) const;

  private:
    virtual OUTPUT* _process(INPUT*) const = 0;
};
```

Inheritance Ladder - Scalability



Python API

- Python API mirrors that of C++
- Python doesn't like const but can emulate this for the mappers in a non-thread safe way (probably ok as focusing on multi-processing)
- Ultimate aim was to maintain API in just one place, the C++, and have the python API auto generated from it using SWIG.
- Unfortunately neither SWIG nor BOOST python can handle the nested templates from the mapper base class.
- Additionally python doesn't have overloading so cant have an exact 1:1 'mapping' between C++ and Python
- May be possible to extend Python using the C++ of the API (havent looked into this yet)

Contents

1. API
2. Data Structure

Motivation

- The goal here was to develop a binary data structure to compliment/replace the existing ASCII Json one
- ROOT TTrees were chosen as a popular, versitile and HEP orientated data format
- First step was to develop streamers that would simplify process of storing and retreiving any arbitrary data structure
- Any arbitrary data structure can be stored provided that ROOT has been made *aware* of it and all it's components.
- This can be done using the *rootcint* dictionary generator tool and (I think I'm right in saying) is performed automatically for MAUS data structure.
- From a new data type developer's point of view, one must include the *ClassDef* macro from ROOT's TObject header.

'ROOT Aware' Objects

As mentioned one must include the *ClassDef* macro from TObject.h

Minimal Data structure

```
#include <TObject.h>
class Spill {
  ...
 private:
  ...
  ClassDef(Spill, 1)
};
```

Must also generate root dictionary by adding line like:

```
#pragma link C++ class Spill+;
```

into *LinkDef.hh* and generate using:

```
rootcint -f SpillDict.C -c Spill.hh LinkDef.hh
```

ROOT Streamers - Output

example...

```
int main(){

  B *ob1 = new B();
  B  ob2;

  orstream file("TestData.root");
  file << branchName("TestBranch1") << ob1;
  file << branchName("TestBranch2") << ob2;

  ob1->a = 1;
  ob1->b = 2;
  ob3.a  = 12;
  ob3.b  = 14;

  file << fillEvent;
  file.close();
}
```

ROOT Streamers - Input

example...

```
int main(){

  B *ob1 = new B();
  B  ob2;

  irstream  file("TestData.root");
  file >> branchName("TestBranch1") >> ob1;
  file >> branchName("TestBranch2") >> ob2;

  while(file >> readEvent){
    cout<<"Event:"<<endl;
    cout<<"branch1: a = "<<ob1->a<<" b = "<<ob1->b<<endl;
    cout<<"branch2: a = "<<ob2.a <<" b = "<<ob2.b <<endl;
  }

  file.close();
}
```

For Data Structure Developers

- These streamers are wrapped in the Output/InputCppRoot python tools.
- OutputCppRoot automatically converts from Json document to C++ 'ROOT aware' data structure before streaming
- InputCppRoot automatically converts from C++ 'root aware' structure to Json document after streaming
- The conversion is performed by a converter which sets up and populates the C++ data sturcture using a *processor* object.
- Developers of new data structure elements must also provide a processor object which knows how to populate the new structure from a given (Json) input.

Data - Spill -

DAQData

- V830
- TriggerRequestArray
  - V1290Array
- TOF1DaqArray
  - V1724Array
  - V1290Array
- CkovArray
  - V1731Array
- TOF2DaqArray
  - V1724Array
  - V1290Array
- UnknownArray
  - V1290Array
- KLArray
  - V1724Array
- TagArray
  - V1724Array
- TOF0DaqArray
  - V1724Array
  - V1290Array
- TriggerArray
  - V1290Array

Scalars
EMRSpillData
MCEventArray

- Primary
- VirtualHitArray
- SciFiHitArray
- TOFHitArray
- SpecialVirtualHitArray
- TrackArray
  - StepArray

ReconEventArray

- TOFEvent
  - TOFEventSlabHit
    - TOF1SlabHitArray - Pmt1&2
    - TOF0SlabHitArray - Pmt1&2
    - TOF2SlabHitArray - Pmt1&2
  - TOFEventSpacePoint
    - TOF0SpacePointArray
    - TOF1SpacePointArray
    - TOF2SpacePointArray

- TOFEvent (Cont.)
  - TOFEventDigit
    - TOF1DigitArray
    - TOF0DigitArray
    - TOF2DigitArray
- SciFiEvent
- CkovEvent
  - CkovDigitArray
    - CkovA
    - CkovB
- KLEvent
- EMREvent
- TriggerEvent
- GlobalEvent

Final Thoughts...

- The MAUS data structure is *very* nested.
- The upshot of this is that one cannot Draw() or Scan() the data from the files interactively.
- This is a ROOT *feature* which they have no intention of changing.
- Not the end of the world as one can still read data out of the structure and plot it manually.
- One can also explore the data structure layout interactively.
- It is however annoying...