C++ Design For Research

T.F. Cootes

December 18, 2003

1 Aims

- To promote thought/discussion about design
- To suggest a common approach
- To encourage good practise

Sources:

- ‘Design Patterns’ Gamma *et al*
- ‘Effective C++’ and ‘More Effective C++’, Scott Meyers

2 Contents

- Why Design for Research?
- The Strategy Pattern
- A Basic Abstract Base Strategy
- Loading/Saving by Pointer to Base Class
- Iterators
- Formatting Nested Output
- Hiding Templates

3 Why Design for Research?

- Good OO design requires detailed understanding of problems.
- In research, we don't know the problems
  (until it's too late).
- Aim to make designs as flexible as possible without compromising OO principles.

4 The Strategy Pattern

- Encapsulates an algorithm from a family
- Allows algorithm to vary independently of the client
- Example:

```
class DistanceBase
{
public:
    virtual ~DistanceBase() {}
    virtual double distance(const vnl_vector<double>&) = 0;
};




class L1Distance : public DistanceBase
{
    virtual double distance(const vnl_vector<double>& v)
    {
        double sum=0.0;
        for (int i=1;i<=v.size();++i) sum+=vcl_fabs(v);
        return sum;
    }
};

class L2Distance : public DistanceBase
```

```
{
    virtual double distance(const vnl_vector<double>& v)
    { return vcl_sqrt(dot(v,v)); }
};
```

Client:

```
class NearestNeighbour
{
private:
    DistanceBase* _d;
public:
    NearestNeighbour(DistanceBase& d) : d_(&d) {}
    int nearest(const vcl_vector<vnl_vector<double> >& data,
                const vnl_vector<double>& v);
        // Uses d_->distance(dv)
};
```

We can change the functionality without changing the implementation:

```
L1Distance distance1;
L2Distance distance2;
NearestNeighbour nearest1(distance1);  // Uses L1 metric
NearestNeighbour nearest2(distance2);  // Uses L2 metric
```

4.1 Improved Implementation

Consider:

```
class NearestNeighbour
{
private:
    DistanceBase* d_;
public:
    NearestNeighbour(DistanceBase& d) : d_(&d) {}
...
```

- This uses a pointer to the external DistanceBase object

- The external object must be in scope at all times
- Nasty.
- There is an elegant solution: clone()

4.2 clone()

The clone() function should create a copy of an object on the heap.

```
class DistanceBase
{
public:
...
    virtual DistanceBase* clone() const = 0;
};

class L1Distance : public DistanceBase
{
...
    virtual DistanceBase* clone() const
    { return new L1Distance(*this); }
};
```

4.3 Using clone()

```
class NearestNeighbour
{
private:
    DistanceBase* d_;
public:
    NearestNeighbour(const DistanceBase& d)
        { d_ = d.clone(); }
    ~NearestNeighbour()
        { delete d_; }
...
};
```

Note: We must also write explicit operator=() and copy constructors.
Implementation of copy and copy constructor

```
NearestNeighbour::NearestNeighbour(const NearestNeighbour& n)
    : d_(NULL)
{
    *this = n;   // Uses operator=()
}

NearestNeighbour&
NearestNeighbour::operator=(const NearestNeighbour& nn)
{
    delete d_; d_=NULL;
    if (nn.d_) d_ = nn.d_->clone();
    return *this;
}
```

4.4 Default constructors

- We often need to be able to construct objects with no arguments, then set them up later.
- One must be able to check that they've been set up correctly when used.
- Solution: Access objects through functions
- eg: DistanceBase& distance()

```
class NearestNeighbour
{
private:
    DistanceBase* d_;
public:
    ...
    NearestNeighbour() : d_(NULL) {}
    void distance(const DistanceBase& d) { d_ = d.clone(); }

    DistanceBase& distance();  // Checks d_ is valid

    int nearest(const vcl_vector<vnl_vector<double> >& data,
```

```
                const vnl_vector<double>& v);
        // Uses distance().distance(dv)
};
```

Implementation: Return error if object not set up.

```
DistanceBase& NearestNeighbour::distance()
{
   if (d_) return *d_;
   else
   {
       vcl_cerr<<"NearestNeighbour::distance() : ";
       vcl_cerr<<"No object defined."<<vcl_endl;
       vcl_abort();
   }
}
```

5 Loading/Saving by Base-Class Pointer

- Suppose we wish to save the NearestNeighbour class to disk, then reload it later.
- To load correctly, we must be able to work out what type of DistanceBase was saved, and create a new copy.
- The vsl_binary_loader class can do this for us.
- To use it we must extend the base class to include
  - virtual void b_read(vsl_b_ostream&);
  - virtual void b_write(vsl_b_istream&) const;
  - virtual const vcl_string& is_a() const;
- Where the b_write() must first save the name of the class, is_a().

For example:

```
class DistanceBase
{
public:
    virtual ~DistanceBase() {}
    virtual double distance(const vnl_vector<double>&) = 0;

    virtual DistanceBase* clone() const = 0;
    virtual void b_read(vsl_b_istream&) =0;
    virtual void b_write(vsl_b_ostream&) const =0;
    virtual vcl_string is_a() const =0;
private:
static vcl_string classname_; // To record name of class
};
```

Implementation for L1Distance:

```
// Declare and define the vcl_string to hold the class name
vcl_string DistanceBase::classname_ = vcl_string("DistanceBase");

short L1Distance::version_no() const { return 1; }

vcl_string L1Distance::is_a() const
{
  return vcl_string("L1Distance");
}

void L1Distance::b_write(vsl_b_ostream& bfs) const
{
   vsl_b_write(bfs,is_a());
   vsl_b_write(bfs,version_no());
   // No other data
}

void L1Distance::b_read(vsl_b_istream& bfs)
{
  if (!bfs) return;

    vcl_string name;
    vsl_b_read(bfs,name);
```

```
    if (name!=is_a())
    {
        vcl_cerr<<"L1Distance::b_read() Unable to load ";
        vcl_cerr<<name<<" into L1Distance object."<<vcl_endl;
        vcl_abort();
    }
    short v;
    vsl_b_read(bfs,v);
    switch (v)
    {
        case (1):
            // No other data
            break;
        default:
            vcl_cerr<<"Unexpected version number "<<v<<vcl_endl;
            vcl_abort();
    }
}
```

In addition, we need the following associated functions in the base class:

```
void vsl_b_write(vsl_b_ostream& bfs,
                         const BaseClass* b)
{
    if (b)  b->b_write(bfs);
    else    vsl_b_write(bfs,vcl_string("VSL_NULL_PTR"));

}

void vsl_b_write(vsl_b_ostream& bfs,
                         const BaseClass& b)
{
    b.b_write(bfs);
}

void vsl_b_read(vsl_b_istream& bfs, BaseClass& b)
{
    b.b_read(bfs);
}
```

5.1 vsl_binary_loader$< T >$

- The vsl_binary_loader is responsible for loading objects by base class pointer
- You must declare one for each base class, typically in the .cc file, using the following syntax:

```
#include "vsl/vsl_binary_loader.h"

// For load/save by pointer:
vsl_binary_loader<BaseClass>*
      vsl_binary_loader<BaseClass>::_instance = NULL;
```

- The Loader must be given one example of each derived class that might be on the disk.
- A useful associated function to declare is:

```
void vsl_add_to_binary_loader(const BaseClass& base)
{
    vsl_binary_loader<BaseClass>::instance().add(base);
}
```

If this is all done for DistanceBase, an example of its use is:

```
// Configure loader
vsl_add_to_binary_loader(L1Distance());
vsl_add_to_binary_loader(L2Distance());

DistanceBase d_ptr1 = new L1Distance();
// Save by pointer:
vsl_b_ofstream bfs("data.out",vcl_ios_out);
vsl_b_write(bfs,d_ptr1);
bfs.close();

// Load by pointer:
DistanceBase d_ptr2 = NULL;
bfs.open("data.out",vcl_ios_in);
vsl_b_read(bfs,d_ptr2); // Invokes vsl_binary_loader
```

```
bfs.close();

// d_ptr2 now points to a new L1Distance object
```

We can then define load/save methods for NearestNeighbour:

```
void NearestNeighbour::b_write(vsl_b_ostream& bfs) const
{
   vsl_b_write(bfs,version_no());
   vsl_b_write(bfs,d_);   // Save by pointer
}

void NearestNeighbour::b_read(vsl_b_istream& bfs)
{
  if (!bfs) return;

    delete d_; d_=NULL; // Zap current distance

    short v;
    vsl_b_read(bfs,v);
    switch (v)
    {
        case (1):
            vsl_b_read(bfs,d_);  // Load by pointer
            break;
        default:
            vcl_cerr<<"Unexpected version number "<<v<<vcl_endl;
            vcl_abort();
    }
}
```

6 Formatting Nested Output

- We often end up with nested hierarchies of classes containing classes.
- Printing out details can be difficult.
- Nested output makes it easier to read.

- The vsl_indent class can help.

```
#include "vsl/vsl_indent.h"

class Fred
{
public:
    void print_summary(vcl_ostream& os) const { os<<vsl_indent()<<"Fred's data"; }
};

vcl_ostream& operator<<(vcl_ostream& os, const Fred& fred)
{
    os<<"Fred: "<<vcl_endl;
    vsl_indent_inc(os);
    fred.print_summary(os);
    vsl_indent_dec(os);
    return os;
}


class Jim
{
private:
    Fred _fred;
public:
    void print_summary(vcl_ostream& os) const
    { os<<vsl_indent()<<_fred<<vcl_endl;
      os<<vsl_indent()<<"Jim's other data"; }
};

vcl_ostream& operator<<(vcl_ostream& os, const Jim& jim)
{
    os<<"Jim: "<<vcl_endl;
    vsl_indent_inc(os);
    jim.print_summary(os);
    vsl_indent_dec(os);
    return os;
}
```

```
main()
{
    Jim jim;
    vcl_cout<<jim<<vcl_endl;
}
```

Output:

```
Jim:
    Fred:
        Fred's data
    Jim's other data
```

7 Example Abstract Base Class

See the RADIAL proformas

```
class BaseClass {
public:
    virtual ~BaseClass();

    // Standard IO
    virtual vcl_string is_a() const =0;
    virtual BaseClass* clone() const =0;
    virtual void print_summary(vcl_ostream& os) const =0;
    virtual void b_read(vsl_b_istream& bfs) =0;
    virtual void b_write(vsl_b_ostream& bfs) const =0;
};
```

7.1 Associated Functions

```
void vsl_add_to_binary_loader(const BaseClass& model);

void vsl_b_write(vsl_b_ostream& bfs, const BaseClass& b);
void vsl_b_write (vsl_b_ostream& bfs, const BaseClass* b);
void vsl_b_read(vsl_b_istream& bfs, BaseClass& b);
```

```
vcl_ostream& operator<<(vcl_ostream& os,const BaseClass& b);
vcl_ostream& operator<<(vcl_ostream& os,const BaseClass* b);
```

8 Examples (AppModel Library)

The TC_VectorModelBase represents mappings from one vector space (eg parameters) to another:

```
\\ Declare classes only refered to by pointer/reference:
class vpdfl_pdf_base;
class vsml_vector_model_builder;

class vsml_vector_model {
private:
    vpdfl_pdf_base  *_param_density;
    vsml_vector_model_builder  *_builder;
...

public:
    vsml_vector_model();
    vsml_vector_model(const vsml_vector_model&);
    vsml_vector_model& operator=(const vsml_vector_model&);
    virtual ~vsml_vector_model();

    virtual void instance(vnl_vector<double>& x, const vnl_vector<double>& b) =0;
    virtual void params(vnl_vector<double>& b, const vnl_vector<double>& x) =0;
...
    // Distribution of parameters:
    const vpdfl_pdf_base& paramDensity() const;

    const vsml_vector_model_builder& builder();

    // Standard IO
    short version_no() const;
    virtual vcl_string is_a() const =0;
    virtual vsml_vector_model* clone() const = 0;
    virtual void print_summary(vcl_ostream& os) const;
    virtual void b_write(vsl_b_ostream& bfs) const;
```

```
    virtual void b_read(vsl_b_istream& bfs);
};
```

This demonstrates:

- Pre-declaring classes only refered to by pointer/reference.
  This speeds compilation and can reduce dependencies.
  You need only include the headers when explicit interface required.
- 'Model' class returning a suitable 'Builder' object.
  (See below)

```
class vsml_vector_model;

class vsml_vector_model_builder {
public:
    virtual ~vsml_vector_model_builder() {};

    virtual void build(vsml_vector_model& model,
                        mbl_data_wrapper<vnl_vector<double> >& data) =0;
    ....

    // Standard IO
    virtual vcl_string is_a() const = 0;
    ....

};
```

9 Iterators

- Objects which step through data.
- Hide container from client
- Can load/generate data object (eg image) when required (saving time/memory)
- (AppModel doesn't use STL iterators - 'Template Bloat').

```
class mbl_data_wrapper<vnl_vector<double> > {
public:
    virtual ~mbl_data_wrapper<vnl_vector<double> >();

    virtual int n_samples() const =0;
    virtual int n_dims() const =0;
    virtual const vnl_vector<double>& first() =0;
    virtual const vnl_vector<double>& current() =0;
    virtual const vnl_vector<double>& next() = 0;

    ...
};

// Example:
int n = data->n_samples();
vnl_vector<double> sum = data->first();
for (int i=1;i<n;i++)
    sum += data->next();
vnl_vector<double> mean = sum/n;
```

Processing data on the fly:

```
class TC_NormalisedVectorData : public mbl_data_wrapper<vnl_vector<double> > {
public:
    TC_NormalisedVectorData(const mbl_data_wrapper<vnl_vector<double> >& data,
                     const TC_VTransformBase& normaliser);
    mbl_data_wrapper<vnl_vector<double> >& data();
    TC_VTransformBase& normaliser();

    virtual int n_samples() const { return data().n_samples(); }
    virtual int n_dims() const { return data().n_dims(); }

    virtual const vnl_vector<double>& first()
       { v_ = data().first();
         normaliser().normalise(v_);
         return v_; }
    ...
};
```

10 Hiding Templates

- Templates are useful to avoid multiple implementations (eg mil_image_2d_of$< T >$)
- Problem: Any classes using such classes may have to be templated themselves
- Too many templates: Slow compilation and 'code bloat'
- We can hide templating using abstract base classes
- For instance mil_image is parent to mil_image_2d_of$< T >$:

```
class mil_image_2d : public mil_image
{
public:
    virtual ~mil_image() {};
    const mil_transform_2d& world2im() const;

    virtual int nx() const =0;
    virtual int ny() const =0;
};

template<class T>
class mil_image_2d_of : public mil_image_2d
{
public:
...
}

class TC_ImageToVectorBase {
public:
    ...
    virtual void sample(vnl_vector<double>& v,
                        const mil_image& image,
                        const vcl_vector<vgl_point_2d<double> >& pts) =0;
    // Warp image using pts and sample into v
};
```

- Derived classes cast image to appropriate templated class
- Clients need not worry about details of image type, as long as the derived implementations of ImageToVectorBase match the actual image type.

For instance:

```
class TC_RegionModel {
public:
    TC_RegionModel(const TC_ImageToVectorBase&,
                   const vsml_vector_model&,
                   const TC_VTransformBase& normaliser);

    // Find b which best fit model to image given pts:
    void params(vnl_vector<double>& b, vnl_vector<double>& g_trans,
                const mil_image& image,
                const vcl_vector<vgl_point_2d<double> >& pts);

    // Reconstruct image given b and pts:
    void reconstruct(mil_image& image,
                     const vnl_vector<double>& b, const vnl_vector<double>& g_trans,
                     const vcl_vector<vgl_point_2d<double> >& pts);
    ...
};
```

11 Conclusions

- Think about design issues when starting new code
- Think about how you may change it as research progresses
- Strategy approach allows rapid change of algorithm
- Iterators hide containers from client

Also:

- Document your code (perceps)
- See if anyone has done it before