C++ INTERFACE CLASSES – STRENGTHENING ENCAPSULATION

Separating a class's interface from its implementation is fundamental to good quality object oriented software design/programming. However C++ (when compared to, say, Java) provides no indigenous mechanism for expressing such a separation. Therefore, a number of idioms supporting this separation have evolved in C++ practice, and were the subject of an article in Overload 66 I co-authored with Alan Griffiths [1]. The idioms covered in that article do not just cover object oriented programming, but other approaches (such as value based programming) as well.

For object oriented programming, the principle mechanism of separation is the *Interface Class*. An *Interface Class* contains only a virtual destructor and pure virtual functions, thus providing a construct analogous to the interface constructs of other languages (e.g. Java). I discussed *Interface Class*es in [2] and explored an example of their application and usefulness in [3] (published in Overloads 62 and 68 respectively).

In this article I would like to discuss the role played by *Interface Class*es in strengthening encapsulation. In doing so, I hope to extend the discussion to use *unit testing* as an example of how *Interface Class*es underpin encapsulation (while taking a swipe at the *Singleton* design pattern [4] in the process).

A Motivating Example

Consider a GUI based drawing program, where the user manipulates shapes – such as lines and circular/elliptical arcs – within a window. It is an old chestnut that serves well as a motivating example.

First, we have a class hierarchy for the shapes. This will be headed up by an *Interface Class* called (guess what) `shape`:

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

  virtual void move_x(distance x) = 0;
  virtual void move_y(distance y) = 0;
  virtual void rotate(angle rotation) = 0;

...
};
```

Second, the `shapes` are stored in a drawing, let's represent this programmatically with an *Interface Class* called `drawing`:

```
class drawing
{
public:
    virtual ~drawing();
    virtual void save(repository& r) const = 0;
...
};
```

Please take note of the `drawing::save(repository& r)` member function – specifically, its `repository&` parameter. This means we need a definition for `repository`:

```
class repository
{
public:
  virtual ~repository();
  virtual void save(const shape* s) = 0;
...
};
```

The `repository` class is a programmatic representation of the repository where drawings are kept when not in memory, i.e. the storage (e.g. a database).

Having introduced the participants in this example, it is time to move on. Before I do though, there are a couple of things I would like to point out:

1. I have introduced the participants only as *Interface Class*es, without any *implementation* classes. This example does not require all of them to have implementations shown. Therefore, implementations will be introduced as (and if) needed.
2. In reality, the `shape` class would need member functions for the extraction of its state; this is so its state can be stored in a database (or other storage mechanism used in the implementation of `repository`). However these are (once again) not needed for this example and are therefore omitted for brevity.

Repository as a Singleton

Presumably there will only be one instance of `repository` needed by the drawing program, and in this example, I am assuming this is the case. Therefore, it seems reasonable (or does it? – but I'm coming to that) to apply the *Singleton* design pattern – i.e. to make it such that:

– There can be only one instance of repository in the program

– The one instance is globally accessible wherever it is needed

There are many ways to implement *Singleton*, but the one used below to implement `repository` is quite a common one:

```
class repository
{
public:
  static repository& instance()
  {
      static repository inst;
      return inst;
  }
  void save(const shape* s);
...
};
```

Now for the part played in this article by unit testing – it is time to write a (single) unit test for the `drawing::save` member function. I'm going to assume we have a drawing instance that contains five shapes. The unit test I want to write saves a `drawing` object in the `repository`, and then verifies that the number of `shapes` actually saved is equal to five:

```
void unit_test(const drawing& d)
{
  d.save(repository::instance());
  ...
}
```

As you may have noticed, there is no such test in the above function. This is because it suddenly becomes apparent that some work needs to be done. For this test, what I need is a `repository` implementation that can count `shapes`.

With the *Singleton* approach to `repository`'s implementation, there are (at least) the following associated issues:

- In order to use repository implementations specialised for unit tests, it is necessary to link in a specialised test version of `repository`.
- As `repository` is – and with this approach, *must* be – hard coded in the test by name, it is not possible to have more than one repository implementation to test different things. Therefore, one test version of `repository` must support *all* tests, and must be modified when a new test is added.
- As a result of one test repository implementation supporting all tests, it is more difficult to test specific pieces of code. That is, implementing *unit* tests becomes more difficult.

The above issues are a direct result of accessing the `repository` in a global context – that is, a consequence of bypassing `unit_test`'s *programmatic* interface.

This approach can be made to work. However, at this point, I'm suggesting there is a simpler method.

Using Mock Object

Let's put `repository` back the way it was when first introduced:

```
class repository
{
public:
  virtual ~repository();
  virtual void save(const shape* s) = 0;
...
};
```

It is now, once again, an *Interface Class*, and this means different implementations are possible. The unit test under discussion requires a `repository` implementation that can count `shapes` stored in it:

```
class counting_repository : public repository
{
public:
  counting_repository() : count(0) {}
  virtual void save(const shape* s)
  { ++count; }
  unsigned int num_saved() const
  { return count; }
...
private:
  unsigned int count;
};
```

There you have it: `counting_repository::save` does not (in this particular test implementation) actually save anything, it just increments a counter. This approach is known as *Mock Object* (sometimes known as *Mock Implementation*). It's time to take stock of how the unit test now looks:

```
void unit_test(const drawing& d)
{
  counting_repository counter;
  d.save(counter);

  assert(counter.num_saved() == 5);
}
```

Note that because `counting_repository` is specific to this particular unit test, it can be defined within the unit test's code (e.g. within the same source file as the `unit_test` function). As a (pleasant) consequence, there is no need to link in any external code, and the unit test assumes full control over the test to be performed.

Finally

The approach of using *Mock Object* with `unit_test` is an example of a pattern known as *Parameterise From Above*. One perspective on *Parameterise From Above* is that it is the "alter-ego" of *Singleton* (and other approaches involving globally accessible objects). *Singleton* is a dysfunctional pattern – one that transforms the design context for the worse, rather than for the better. *Parameterise From Above* is a pattern that is "out there", but for which (to the best of my knowledge) there is (so far) no formal write up.

Encapsulation is fundamental to object oriented design – and *Interface Class* is an idiom that can underpin the strengthening of encapsulation. The *Mock Implementation* of `repository` is possible because `repository` is an *Interface Class*. Observe how `unit_test` can use different `repository` implementations, without `unit_test`'s implementation being affected.

References

[1] Mark Radford and Alan Griffiths, *Separating Interface and Implementation in C++*, Overload 66 (www.twonine.co.uk/articles/SeparatingInterfaceAndImplementation.pdf)
[2] Mark Radford, *C++ Interface Classes – An Introduction*, Overload 62 (www.twonine.co.uk/articles/CPPInterfaceClassesIntro.pdf)
[3] Mark Radford, *C++ Interface Classes – Noise Reduction*, Overload 68 (www.twonine.co.uk/articles/CPPInterfaceClasses-NoiseReduction.pdf)
[4] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.