ISSN 1354-3172

Overload

Journal of the ACCU C++ Special Interest Group

Issue 27

August 1998

Contents

Editorial

Once again, I've left it too late to write you a studied piece, so I'm going to produce the familiar plea, recount some recent news, and subject you to one of my rants

Patterns

This issue the software patterns section swells with some more excellent discussion of common architectural solutions. Look to your project for an elegant reusable concept that you could document for the membership.

Linux

At the end of July Linux received a healthy endorsement from a group of software vendors. Informix and Oracle are porting their database servers, and Netscape is porting both Directory and Messaging servers.

Since Linux was introduced, about eight years ago, I've viewed it as a curiosity. I remember installing Minix, the Andrew Tanenbaum tutorial Unix implementation, around that time. I suffered the various trials of downloading disk drivers, patching up the code, compiling it, and re-linking it – all on a single sided 360k floppy. Since those university days, I've been a corporate DOS, Windows, and NT programmer. Not much motivation for battling with homebrew unix.

Anyway, yesterday, I picked up a copy of RedHat 5.1 (a popular Linux distribution) at the local high-tech supermarket – which, incidentally, is now stocking propane fuelled BBQs. I slapped the CD into my homebrew PC. It was installed and running in fifteen minutes. It's amazingly speedy, but the X based administration tools are pitiful, in comparison to NT.

Looking forward, some things need to happen before Linux will be a widely acceptable alternative. PC manufacturers will have to start offering it as a pre-installed option on new machines. There need to be big name companies offering technical support contracts. And, the chip manufacturers need to open labs specifically for performance tuning Linux applications. Rumours have it that Intel and Compaq/Digital are looking to fill these roles. Hopefully, this will all come about over the next couple of years.

Anyway, try Linux, you can dual boot it with Win98 ☺

Software Installation

I've seen two projects do this. They put a one month task in the project plan for 'installation'. No one wants much to do with it. It's left unassigned, slowly creeping out to the middle of the schedule. Then someone is lumped with it, and the real requirements start trickling down from management. Suddenly, it's the behemoth of all installation scripts - implemented in that awful windows package that everyone uses. It has Typical install, Custom, install, Upgrade install, Silent install, Migrate 1.x install, Migrate 2.x install, Deploy Standalone, Deploy Cluster, Add-on Package #1, etc, etc.

If you're the unlucky developer creating the software to be is delivered by the 'Mother of all Installers' you're tightly coupled to its progress. You can't install to test when it's broken, or out of date. To try a feature you need to build the code, package the bits, run the installer, curse, fix the installer, package the bits, run the installer, etc, etc.

Advice 1: Make your software self installing. If the expected configuration information doesn't exist, then dump out a default one from static memory. This de-couples the developers, makes the software more resilient to environmental changes, and makes the installation software simpler.

Advice 2: Don't write all those fancy wizards in the installer. Put them in the administration tools so they may be used over and over again. The installation delivers the bits, then launches the regular administration interface.

So, thumping fist on pub table, the installation package should do only what's necessary to bootstrap the software, then you're on to administration.

John Merrells
merrells@netscape.com

Copy Deadline

All articles intended for publication in *Overload 28* should be submitted to the editor by September 1st, and for *Overload 29* by November 1st.

Software Development in C++

UML Interactions & Collaborations

By Richard Blundell

Introduction

In earlier articles we have covered a number of techniques for documenting and designing the static behaviour of systems. We saw one way of representing dynamic behaviour when we looked at State-Transition diagrams [1], but these diagrams only really deal with a single object at a time.[1] This month we shall look at collaborating objects and their interactions over time. The charts we will use are useful for documenting real-time systems as well as for complicated processes involving many calls between the objects involved.

Interactions

Conceptually, objects interact by exchanging *messages*. A message is typically 'sent' using a normal function call, but can also be sent as an inter-thread or inter-process signal, or an event triggered by a hardware device or operating system interrupt such as a timer. An *interaction* is a set of message exchanges that collectively achieve some purpose, usually one that represents some higher-level action. In other words, an interaction is the collection of inter-object messages that produces some outcome.

In system design and documentation, it is a common requirement to document the (non-trivial) interactions of a system, and there are two types of *Interaction diagram* with which this can be done – *Collaboration diagrams* and *Sequence diagrams*.

[1] Although of course that object can be an aggregation of other objects, or a system or subsystem with conceptual 'states'.

Collaborations

A *collaboration* is a set of objects involved in completing some action or operation, combined with the interaction that produces the action. The collaboration contains only those objects that are involved in the action (or actions – a collaboration can describe several related or even unrelated operations or interactions).

Collaboration Diagrams

Collaboration diagrams document collaborations. Visually, a collaboration diagram looks like an object diagram, complete with associations between the objects shown in the normal way. On top of this diagram are superimposed an ordered set of *message flow* arrows showing the pattern of messages that form the interaction. A simple example is shown in figure 1.

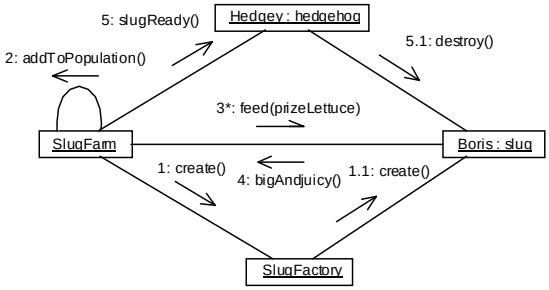


Figure 1 – A collaboration diagram showing the creation of a slug in a slug farm, and its ultimate consumption, after fattening, by a hedgehog.

A message flow shows the transfer of a message from one object to another. A short arrow is used to denote a message flow. Typically, the arrowhead is solid, indicating sequential or nested operation as is used in normal single-threaded procedural design. A half stick arrowhead is often used to denote asynchronous calls. If an object sends a message to itself (to show iteration, for example) the arrow can follow a self-association line, as shown in step 2 of the

figure (and can optionally be labelled with the stereotype «self»). Each arrow can be labelled with a set of expressions that show details of the messages, and the conditions under which the message is sent.

Simple message flows are labelled with a *sequence number* and a message name, possibly with a return type. For example:

1.1.4: ret := process(arg, …)

The sequence number is the part before the first colon. Sequence numbers show the order in which the message flows occur. The number of decimal points shows the call depth. In the above example, this message flow is the fourth at the current nesting level, and is nested three levels deep. An example numbering scheme can be seen in the figure. After the colon, we have the return value, name of the message, and argument list for the message.

There are several extensions to this basic syntax. A *guard condition* can be added (in square brackets) before the sequence number to show a conditional message. The message is only sent if the condition is true:

[t > last] 1.3: update()

As well as digits, letters can be used in a sequence 'number' to show concurrent threads. Thus, a step 1.2 could be followed by 1.2.1a and 1.2.1b, showing that it passes control to two threads.

Before the guard condition, another form of condition can be added that shows the *predecessor* of the message flow. This is a list of sequence numbers of messages (followed by a forward slash) that must all have occurred before the current message will fire. This allows threads to be synchronised by requiring different threads to all have reached some designated point:

1.4a,1.2b/ 2: theyveFinished()

Here, message 2 is only sent after the first thread has sent message 1.4a, and the second thread has sent 1.2b.

Branches and iteration can be shown by appending a *recurrence* term (in square brackets) after a sequence number. An example of a branch would be message labels such as 1.1[t = t0] and 1.1[t < > t0], and the message actually sent depends upon the value of t. If the recurrence conditions are mutually exclusive (as here), then a single procedural branch is suggested. If the conditions overlap, concurrent sequence numbers can be used to show the start of multi-threaded processing. Iteration is shown using an asterisk and an expression showing the details of the iteration, for example 2.5*[i := 0..n-1]. In this example, the labelled message fires n times in succession.

The *lifetime* of an object in a collaboration can be shown using the stereotypes «new», and «destroyed», with «transient» meaning a combination of the two. Sometimes these are shown as constraints in curly braces instead of stereotypes. As we shall see later, object lifetimes can be shown more explicitly on sequence diagrams.

Design patterns are collaborations (plus additional information such as examples of use, limitations, usage guidelines, etc.), and as such can be partly documented using collaboration diagrams. Once a pattern has been documented and named, it can be shown on diagrams using the dashed-oval representation described in an earlier article [2], with the actual objects that enact the pattern bound to the roles within the pattern definition.

That is pretty much it for collaboration diagrams. A normal static structure diagram with message flows to show the interactions that occur.[2] Collaboration diagrams are useful because they show not only the sequence details of the interaction, but also its full context – which objects are involved and how they are related. The disadvantage of these diagrams is that for complicated

[2] A few other capabilities include showing active objects with a heavy border to their rectangular symbol, and nesting objects within others.

interactions they can become very cluttered and difficult to interpret.

Sequence Diagrams

Sequence diagrams are the second type of interaction diagram, and these primarily show the interaction details, omitting much of the information about the collaboration. The objects involved in the interaction are shown, but no relationships are given. To offset this restriction, however, the time-order of the message flows and object lifetimes are much more obvious, and very involved interactions are much simpler to interpret.

A typical sequence diagram has time down the page, and individual objects laid out across the top of the diagram, as shown in figure 2. *Lifelines* for each object are drawn as vertical dashed lines. The lifeline begins at the top of the page or at the point that the object is created in the interaction, if later. The lifeline stops at the bottom of the diagram, or at the point of a large X where the object is destroyed, if earlier. Message flows are shown as horizontal arrows[3] from one object to another, and the messages are laid out in time-order down the diagram. Conditional (or concurrent) behaviour can be shown using multiple message arrows, each labelled with a guard condition. Target lifelines can split in two to show alternative or parallel scenarios, with a recombination possible further down the page.

A feature known as *focus of control* can be added to diagrams to show the intervals over which each object is 'active' in the sense that it is either processing itself or waiting for another object to finish processing (sometimes these two cases are distinguished by shading the box (described next) in the former case). When an object is active, its lifeline temporarily becomes a long thin box. Recursive calls are shown by drawing an additional activation box offset from the main one. The depth of nesting can be shown by multiple offset boxes if you care to go to that much effort!

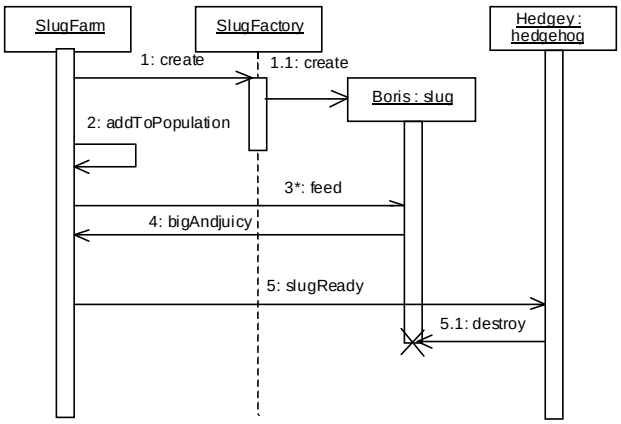


Figure 2 – A sequence diagram corresponding to the collaboration diagram in figure 1. The slug farm, slugs and hedgehogs each have their own thread, so I have shown their focus of control boxes as continuously active (assuming they do other processing in the background), whereas the factory is only active when the create method is called by the farm.

As with collaboration diagrams, message flows can be labelled with guard conditions, message name, arguments, etc. Sequence numbers are usually omitted in single-threaded interactions because the sequence of messages is shown explicitly by the ordering of the arrows. In addition, returns from procedure calls can be shown using dashed return arrows.

Conclusion

Over the months we have seen how to document some of the static and dynamic details of a system. State-Transition diagrams are useful for describing the behaviour of a single class, but to document collaborating objects the Sequence and Collaboration diagrams (collectively called Interaction diagrams) described above are very handy. Next time I'll go back to the design stage taking a look at Use Cases and Use Case diagrams, and see how these can be used to define the scope and behaviour of a system.

[3] … in the case of messages that can be considered to be instantaneous. If there could be a significant delay in the receipt of the message, and if this delay could mean the sequence of messages could be interrupted, a downward-slanted arrow can be used to show this.

References

[1] "UML – State Transition Diagrams", Overload 24 pp 2 – 5

[2] "UML – Objects and Patterns", Overload 23 p 6

Richard Blundell
RichardBlundell@dial.pipex.com

Patterns in C++

Exploring Patterns Part 2 by Francis Glassborow

Before I focus on the pattern for this issue I want to tell you about something that I learnt whilst doing further research for the Singleton pattern.

As most of you know, declaring a copy constructor for a class inhibits the compiler from generating its own. After considerable pressure from several people (I think I was among the most vociferous) WG21 & J16 specified that all the following were classified as copy-constructors (for the sake of example, I am assuming that I am dealing with a class `MyType`):

```
MyType(MyType &);
MyType(MyType const &);
MyType(MyType volatile &);
MyType(MyType const volatile &);
```

In addition all constructors with a first parameter matching one of those four and defaults for all the other parameters will also be copy constructors. The clarification concerned whether the volatile qualified parameters resulted in inhibiting the compiler from generating a copy constructor or a default constructor. Note my wording, I fondly believed that any constructor would only inhibit one or other but not both of the possible compiler generated ones. This believe seems to be shared by many good, even expert, writers. On numerous occasions I have had cause to point out the flaw in code such as:

```
class MyType
{
  MyType();
public:
  // whatever
};
```

where the clear intent is that it should not be possible to create public instances of `MyType`. The flaw is due to a quirk of the grammar for declarations that results in the following code being syntactically correct (though usually hiding undefined behaviour because storage is copied before it has been initialised):

```
Mytype mt = mt;
```

It is the quirky grammar for initialisation in this form that leads me to strongly recommend that you always use the function form for user defined types. If you wrote:

```
MyType mt(mt);
```

You would get a compile time error unless an `mt` of appropriate type had been declared in an outer scope. In other words, always call a constructor explicitly rather than use an implicit constructor and call to the copy constructor. Actually, as more programmers recognise that object types (as opposed to value types) should not have publicly accessible copy constructors the problem will occur less often.

Back to the main point. Unless you have declared one of the copy constructor forms, the compiler is at liberty to attempt to generate one of the form `MyType(Mytype const &)`.

However when I have raised the issue of the missing declaration of a private copy constructor, all I have ever had is '*Ahh… I missed that*' and the writer has added a copy constructor. None of them has ever removed their declaration of a default constructor. I think, that like me, many of them have always thought in terms of two almost disjoint sets of constructors, copy constructors and non-copy constructors (with an overlap caused by

defaulting parameters so as to leave a more general constructor useable as a copy constructor). Even though I have explicitly made that statement (about two disjoint sets) in the presence of some of the World's greatest C++ experts, none have ever corrected me (perhaps some of them just heard what they expected, and some were too polite – though I doubt it, that kind of politeness is unhelpful).

In fact, declaring a copy constructor inhibits the compiler from generating a default constructor as well as any other copy constructor. This means that if you want to prevent public creation of a type all you need to do is to declare a private copy constructor. For example:

```
class MyType
{
  MyType(Mytype &);
public:
  // whatever
};
```

correctly does what was intended by the earlier flawed case.

So we see that the correct idiom for a class that must not be publicly constructable is to declare a private copy constructor.

Now we also need to know what to do if we want to prevent copying but are happy to allow default construction (needed for example if you are intending to provide dynamic arrays of the type – but not the STL containers that require access to a copy constructor). The fix is easy:

```
class MyType
{
  MyType(Mytype &);
public:
  MyType(){};
  // whatever
};
```

That default constructor with an empty body tells the compiler to do exactly what it would have done had it been able to generate a default constructor for itself.

The Visitor Pattern

When you read the following please do not take my word for it, wait until the experts have had a look and either confirm, extend or correct my interpretation. Think of this as an essay being read by a student at a seminar, only after the discussion is complete will those involved know confidently what is true and what is not.

In general we are concerned with providing stable, well-defined behaviour for our abstractions. At the same time we want to reserve the right to change implementation details. This is the main motive for the concept of `private`/`protected`/`public` interfaces. However there are cases where we have a clear idea about how we wish to provide our data but wish to reserve the right to alter behaviour.

One way of tackling this problem is by using a base class to provide the data (together with `protected` read and write member functions) and using derived classes (or classes with pointers to the desired data structure) to provide the behaviour. This works well where we have a single fundamental data type but it does not work when we need a hierarchy. One example given (in Design Patterns) of such a hierarchy is that of the different node types required for a parse tree used by a compiler for a computer language. For example in C the node for an assignment must provide a left pointer to an expression node that evaluates as a modifiable lvalue and a right pointer to an expression node that evaluates to an rvalue. Arithmetic operator nodes need left and right pointers to nodes for expressions evaluating to rvalues.

While the data requirements are very stable (fortunately computing languages infrequently add features requiring new node types) the desired behaviour can change. Indeed the desired behaviour will depend on exactly what you are trying to do(compile, optimise etc.). It is usually unwise to have an interface cluttered with a large number of member functions, particularly if these are only tenuously related to each other. Even if

we could provide an exhaustive list of all the behaviour we want in such a class hierarchy it is probably a poor idea to provide it in the classes.

The idea behind the Visitor pattern is to allow programmers to encapsulate coherent behaviour across a number of classes (not necessarily even in the same hierarchy) into a single class. Typically, we have something like:

```
class A_type;
class B_type;
class C_type;
// etc.
```

These declarations can be replaced with definitions and the various classes are usually part of a hierarchy but this is not necessary for the pattern to work.

```
class MyVisitor
{
public:
  virtual MyVisitor&
      VisitorToA_type(A_type *) = 0;
  virtual MyVisitor&
      VisitorToB_type(B_type *) = 0;
  virtual MyVisitor&
      VisitorToC_type(C_type *) = 0;
  // etc
  virtual ~MyVisitor() throw() {};
};
```

This is an abstract base class from which concrete classes providing single behaviours can be derived. Actually all the functions could share the same function name as overloading would resolve which one to use. Whether you use function overloading or not is a matter of style (shorter names requiring thoughtful reading versus longer names providing specific information). If the visitor is not intended to mutate (change the state of) the host objects then the parameters in the above should be of type `const *`.

Each of the classes to be visited must include a member function of the form:

```
MyVisitor & host(MyVisitor &);
```

Again, `const` qualification should be used as appropriate: `const` member function if the Visitor is non-mutating and `const` qualified parameter if the Visitor is not mutated by visiting.

The body of `host()` depends upon the class in which it is placed so that it calls the appropriate member function of the Visitor. For example:

```
MyVisitor & A_type::host(MyVisitor & v)
{
  return v.VisitorToA_type(this);
}
```

If you have chosen the function overloading mechanism then all host types will have apparently identical bodies (though the type of 'this' will select the correct overload version from the visitor's members functions):

```
MyVisitor & A_type::host(MyVisitor & v)
{
  return v.Visitor(this);
}
```

The return type of the host functions and the members of the visitor could be void but I have a strong preference for returning an object for possible reuse. It doesn't cost much but provides one extra resource for those that wish to use it.

Let me offer a trivial example where you want to be able to dispatch the data to some form of output. You would write something such as:

```
class StoreData: public MyVisitor
{
  istream & output;
  // inhibit copying
  StoreData(StoreData const &);
  void operator = (StoreData const &);
public:
  explicit StoreData(istream & out =
cout) : output(out) {}
  virtual MyVisitor&
      VisitorToA_type(A_type *);
  virtual MyVisitor&
      VisitorToB_type(B_type *) ;
  virtual MyVisitor&
      VisitorToC_type(C_type *) ;
  // etc
  virtual ~StoreData() throw() {};
};
```

Now we come to a problem. The bodies of the member functions of `StoreData` require access to the specific data of the host classes. This means that each of these classes must provide `public` access functions for its data (this does not break encapsulation but it does restrict the owners of the host classes

(but remember that a pre-condition for the use of the Visitor pattern is stable data structures).

Remember that the advantage of the Visitor pattern is that you can retrofit behaviour to a bunch of (usually related) classes. Of course, you can install any specific behaviour into the classes themselves, but one major advantage of the Visitor pattern is that it supports extensible behaviour.

It also works well where you have collections of objects, or even composites of heterogeneous objects. *Design Patterns* gives the example of something like a piece of equipment (such as a computer) that is built from components that are themselves built from components etc. If you want to cost such equipment you could (if you thought far enough ahead) provide a visitor object that was passed around collecting cost information from each sub-component (to do that it would have to visit each sub-sub component recursively). This may sound complicated until you realise that all that is required is that the `host()` function of a component dispatches the visitor to each of the sub-components before calling the specific member function of the visitor object on itself.

I think that the Visitor pattern is a powerful program technique that deserves to be widely known. If you are serious about software development you should work through at least one implementation of Visitor to ensure that you understand it and will remember to provide the groundwork where it has potential use. For example, the Harpist's Hotel project might benefit from a Visitor facility in all classes that provide charges (a bill is made up from a variety of costs that are certainly not all from objects in the same hierarchy; think about meals and rooms.) Of course this problem has many other solutions (such as ensuring that each object includes a reference/pointer to a bill object) and the lack of the need for extensible behaviour probably makes other methods more appropriate.

Before I wrap this up I want to speculate a bit on the possibility of avoiding the need for public read/write access functions for data in host classes.

Keeping Data Out of Reach

The first thought when tackling this problem (restricting access to data) is to consider using friendship. Unfortunately there is no mechanism for granting friendship to a hierarchy of classes and part of the fundamentals of the design of the Visitor pattern seems to require a hierarchy. We need a base class so that the parameter of the `host()` functions can provide polymorphic behaviour (select the type of behaviour that the visitor is going to add). But that does not mean that we need a hierarchy of derived types (to which friendship can only be granted on a one by one basis, which would rather defeat the object of the exercise).

We have another possibility via templates:

```
class MyVisitor
{
public:
  virtual MyVisitor&
      VisitorToA_type(A_type *) = 0;
  virtual MyVisitor&
      VisitorToB_type(B_type *) = 0;
  virtual MyVisitor&
      VisitorToC_type(C_type *) = 0;
  // etc
  virtual ~MyVisitor() throw() {};
};

enum Operation {Op1, Op2};

// to provide a tool for instantiating
template classes
template <Operation op> class Guest :
public MyVisitor
{
  // inhibit copying
  Guest(Guest const &);
  void operator = (Guest const &);
public:
  ~Guest() throw() {}
};

// Note this is still an Abstract Base
Class and so instances
// must be specialised, or be used as
ABC's
// Here is an example of a specialisation
template <> class Guest<Op1>
{
  istream & output;
public:
  explicit Guest(istream & out = cout) :
output(out) {}
  MyVisitor& VisitorToA_type(A_type *);
  MyVisitor& VisitorToB_type(B_type *) ;
```

```
  MyVisitor& VisitorToC_type(C_type *) ;
  // etc
};
```

Unfortunately, though templates can declare friends and ordinary classes can declare instances of templates to be friends there is no syntax available to declare a template class as a friend. So this idea may be interesting but it does not solve the problem of finding a way whereby the various host classes can provide privileged access rights to a Visitor hierarchy.

My next idea was to try and increase the security by using namespaces coupled with fuzzing the types used (remember that the data types/structures for the host classes must be stable if we are using the Visitor pattern.) This is my first attempt:

```
namespace ControlAccess
{
  typedef int Int; // just as an example
  class Object
  {
    Int value;
  public:
    Object(int i=0):value(i){}
    void set_value(Int v){value = v;}
    Int get_value(){ return value;}
  };
}

int main()
{
  ControlAccess::Object obj;
  obj.set_value(3);
  cout << obj.get_value();
  return 0;
}
```

I hoped that by hiding the `typedef` in a namespace that its implicit use outside the namespace would create an error. I agree that this was a pretty vain hope because a `typedef` does not create real type. Of course this code compiled.

So next I tried replacing the `typedef` by a real type:

```
class Int
{
  int value;
public:
  Int(int i=0):value(i){}
  operator int () {return value;}
};
```

Unfortunately, the so called Koenig lookup allows the compiler to find the `Int` type in the context of `obj.set_value(3)` and `obj.get_value()`. I had one last shot in my locker (remember that my purpose is to make it possible to force programmers to think about using the access functions they have in host classes). Consider:

```
class Int
{
  int value;
public:
  explicit Int(int i=0):value(i){}
  int convert_to_int () {return value;}
};
```

Now the two function calls in question fail and require an explicit cast (for `seti`) and a call to `convert_to_int` (for `geti`) to make the compiler happy.

So, if there is data in a host class that you are reluctant to make easily accessible to the world at large, but that you do need to make available to Visitor, you can add this extra layer. Visitors will have to use this as well, but we are assuming that we are catering for data that needs thoughtful use.

Conclusion

I have learnt a lot while putting this article together, and I know I have ranged further afield than strictly required for the topic. However, I think some of the ideas and failed attempts may prove instructive. What I do know is that the process of active exploration rather than passive reading is what provides the value to me. I think the same will apply to you.

Postscript

It has just occurred to me that there is one other mechanism available. Have each host class that has data that you do not want to make generally accessible declare the visitor ABC (MyVisitor for example) a friend. Now you can place those access functions that you want to restrict in the protected interface of the ABC. That way all the concrete visitors will have the access they need but no-one else will. Here is some code by way of example.

```
class MyVisitor;
// sample host classes
class First
```

```
{
  friend class MyVisitor;
  int val;
public:
  First(int i=0):val(i){}
  MyVisitor & host(MyVisitor &);
};

class Second
{
  friend class MyVisitor;
  float val;
public:
  Second(float f=0.0):val(f){}
  MyVisitor & host(MyVisitor &);
};

class Aggregate
{
  First f;
  Second s;
  int val;
public:
  Aggregate(int v=2,
            int first = 1,
            float second=1.0)
    :f(first), s(second), val(v){}
  MyVisitor & host(MyVisitor &);
  int geti(){return val;}
  void seti(int i){val = i;}
};

// now the visitor base class
class MyVisitor
{
protected:
  int getFirstval(First & f){return f.val;}
  void setFirstval(First & f,int i){f.val=i;}
  float getSecondval(First & f){return f.val;}
 void setSecondval(First &f,float i){f.val=i;}
public:
  virtual MyVisitor& visitFirst(First *) = 0;
  virtual MyVisitor& visitSecond(Second *) =0;
  virtual MyVisitor& visitAggregate(Aggregate
*) = 0;
  virtual ~MyVisitor() throw() {};
};

class PrintData: public MyVisitor
{
public:
  MyVisitor& visitFirst(First *);
  MyVisitor& visitSecond(Second *);
  MyVisitor& visitAggregate(Aggregate *);
  ~PrintData()throw(){}
};
```

The implementation of the three member functions might go like this:

```
MyVisitor& PrintData::First(First * f)
{
  cout<< “First is ” << getFirstval(*f);
  return *this;
}

MyVisitor& PrintData::Second(Second * s)
{
  cout<< “Second is ”<< getSecondval(*s);
  return *this;
}

MyVisitor& PrintData::Aggregate(Aggregate
* a)
{
  cout << "Aggregates own data is:" <<
                                  a.geti();
}
```

And the implementations of the host functions are:

```
MyVisitor & First::host(MyVisitor & v)
{
  v.visitFirst(this);
  return v;
}

MyVisitor & host(MyVisitor & v)
{
  v.visitSecond(this);
  return v;
}

MyVisitor & host(MyVisitor & v)
{
  f.host(v);
  s.host(v);
  v.visitAggregate(this);
  return v;
}
```

And finally a very short program to use this:

```
int main()
{
  Aggregate test; // use defaults
  PrintData pd;
  test.host(pd);
  return 0;
};
```

The above code is untested so it is up to you to debug it, in doing so you will need to understand it.

Post-Postscript

Before writing this article I had thought there was a way of declaring a template class a friend. When I failed to get my code to compile I checked with a couple of UK C++ experts who opined that it was not possible, hence the assertion. I have now had a chance to check the FDIS and find that my original belief is justified though the syntax is counter-intuitive.

The inclusion of the line:

template<Operations> friend class Guest;

in each host class should provide the desired access. However, I cannot find a compiler to compile it. Anyway, I believe that my

postscripted solution is technically better as well as being compilable with the current compiler releases.

Francis Glassborow
francis@robinton.demon.co.uk

Almost a Pattern
By Alan Griffiths

Introduction

This article describes a recurring problem in program design and presents both a method of design and an implementation of part of that solution. The problem in question is that of separating the application logic that governs the changes a user may make to objects within the application from the detail of the user interface.

I first documented this problem and solution as part of the development of Experian's "Micromarketer" application. This formed the basis of my presentation at the AGM. Kevlin Henney informs me that he's used a similar design for a similar problem. (This makes two uses: one more and I can call it a pattern!)

The context

Many applications (including "Micromarketer") can be divided into three conceptual layers:

- GUI
- Business abstractions
- Core functionality

Each of these provides services to the layers above and makes use of services provided by the layers below.

Abstraction layer components have attributes (e.g. names) that may be accessed and amended via the user interface (in the case of Micromarketer, wizards & property dialogs). The mechanisms for validating these updates should be independent of the user interface. For instance, the same component attributes may be exposed through several parts of the user interface and the validation needs to be consistent.

Some early parts of Micromarketer were developed with the validation of changes in the user interface. It has proved difficult to ensure that these remain consistent. In particular it is possible to change the name of most component in three ways: via the component "browser" (similar to "Windows Explorer"), via the component properties dialog, or via a wizard. At one stage it was possible to place "invalid" characters into a component name via the browser and to subsequently crash the property dialog by cancelling out of it.

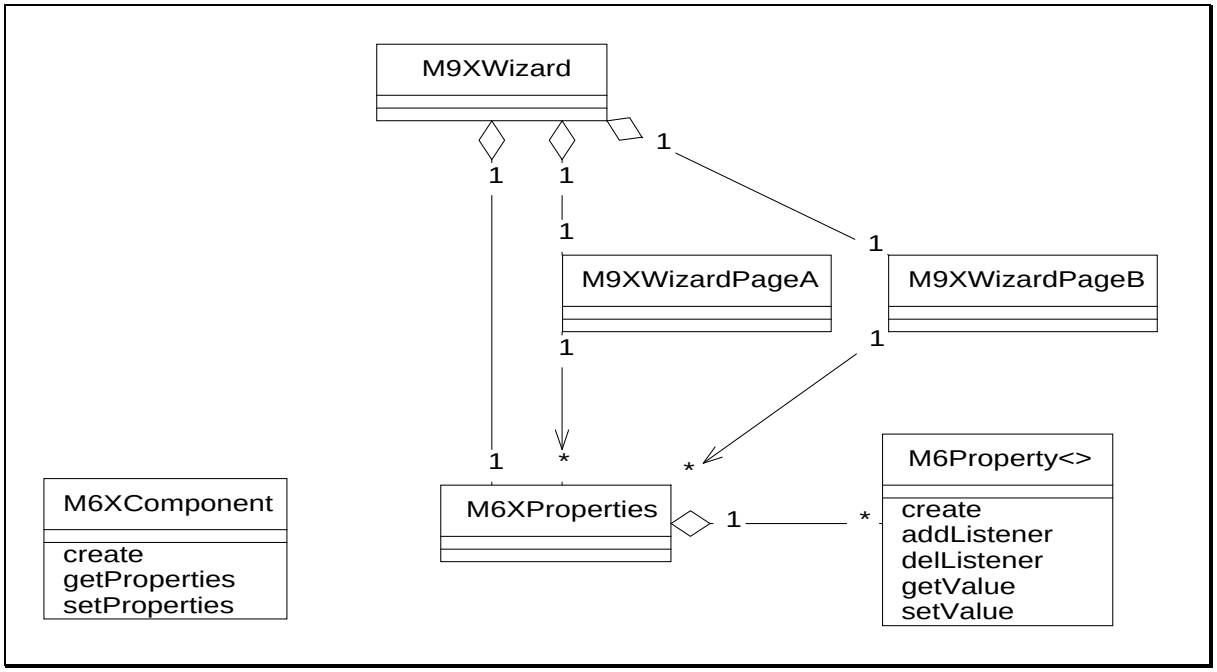
A related problem is that changing some component attributes via wizard page/property sheet may impact another wizard page or property sheet. This could be because the value is displayed there, or because there are some options that may be enabled/disabled accordingly. In practice an approach in which the wizard pages or property sheets implement these notifications has proven error prone, hard to maintain and clearly breaks encapsulation.

Finally it may be observed that changes may not be made (and validated) directly on the component because:

- the component may not yet exist (as in a wizard that creates the component),
- the changes may not be complete (so that the attributes are temporarily inconsistent), or
- because the wizard/properties may be dismissed without performing the update.

Consequently, the wizard/property dialog needs to keep a copy of the component attributes.

The Solution



The Property Template

The behaviour of a "property" is generic (and is templated on the value type):

- it holds a value,
- if an attempt is made to change a value then the change is "validated",
- "interested" objects are notified of value changes.

Validation of changes will be the responsibility of the *ComponentProperties* class (e.g. `M6XProperties`). The wizard, wizard pages, and, possibly, the *ComponentProperties* register as "interested" objects.

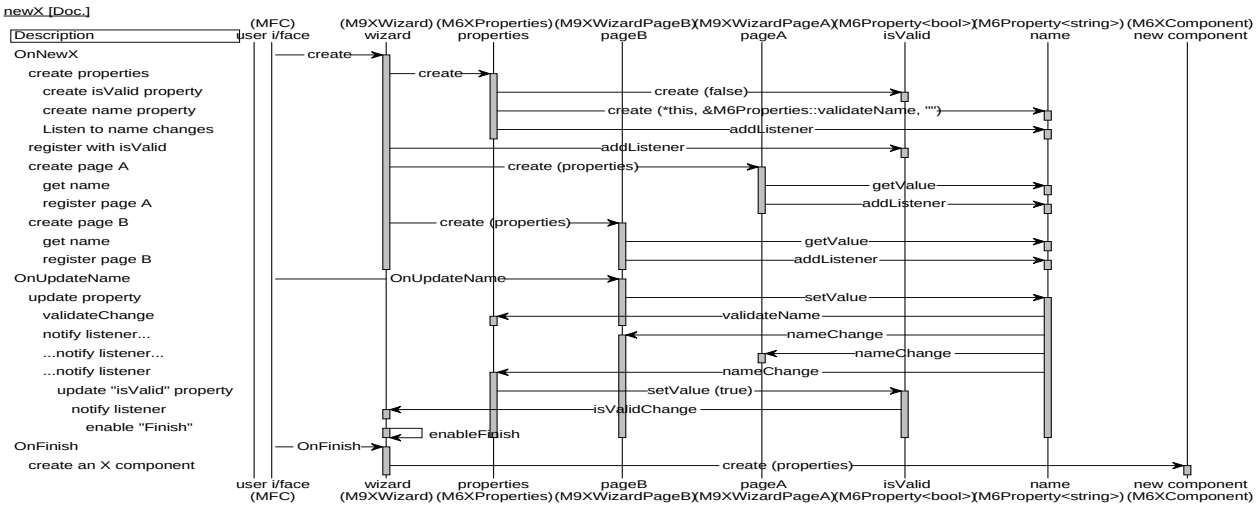
Component Properties Classes

Corresponding to each abstraction layer component type (e.g. `M6XComponent`) there should be a *ComponentProperties* class (e.g. `M6XProperties`). This is implemented in the abstraction layer alongside the component and exposes the accessible attributes of the component as "properties".

The *ComponentProperties* class implements any validation methods required for the component attributes (and attaches them to the appropriate "properties"). In many cases it needs to implement a validation check that cross checks properties for consistency. This can be used to maintain an additional "*isValid*" property.

Each abstraction layer component has a factory method (or a constructor) and instance methods "*getProperties*" and "*setProperties*" all of which accept the corresponding *ComponentProperties* class.

Object interactions



When a wizard (for example) is invoked it creates an instance of the corresponding *ComponentProperties* object. (The *ComponentProperties* object could then be initialised from an existing component if this is appropriate - which is the case for a properties dialog.)

During construction the *ComponentProperties* object sets up the validation for any properties and also attaches listener methods to any properties that have an overall effect. (For example properties that affect the overall self-consistency of the *ComponentProperties*.)

The wizard adds "listener" methods on itself to selected properties. That is, to any properties that affect the wizard globally - for example requiring adding/removing wizard pages, or enabling/disabling "finish".

Each wizard page is initialised with a reference to the wizard's *ComponentProperties* object. It then controls the associate between dialog controls and the properties and can add "listener" methods on itself to any properties that affect the content or behaviour of the page.

When the "Finish" button is selected the component is constructed using (or has its attributes set from) the *ComponentProperties* object.

An outline implementation

The following code outlines an implementation of the "Property" generic used in the above solution (full source code has been was supplied - I presume it will find its way onto the C Vu disc.):

```
template<typename MyValueType> class
M6Property
{
public:
```

A constructor for an unvalidated value:

```
M6Property(MyValueType initValue);
```

This constructor accepts both an initial value and an object and a method on that object that provides the validation check:

```
template<typename MyValidatorObject>
M6Property(
  MyValueType initValue,
  MyValidatorObject& validator,
  int (MyValidatorObject::*
method)(MyValueType));
```

Methods to access and modify the value. "setValue" returns a non-zero error code if the validation fails:

```
MyValueType getValue() const;
int setValue(MyValueType newValue);
```

Methods to allow objects (normally the "user interface" and the owning "component properties") to register for notification of changes to the value of the property.

```
template<typename MyListenerObject>
void addListener(
```

```
  MyListenerObject& listener,
  void
(MyListenerObject::*method)(MyValueType))
;
template<typename MyListenerObject>
void delListener(MyListenerObject&
listener,
void (MyListenerObject::*
method)(MyValueType));
};
```

Known uses

As stated in the introduction Kevlin Henney of QA says he's used a similar implementation (although he's currently too busy to give details). In addition John Merrells (the overload editor) has used a similar idea in a client server environment in which the aggregated properties are passed across the network.

References:

Diagrams are basically OMT (ISBN 0-13-630054-5 Rumbaugh et. al.) with pointless modifications by Select software.

Alan Griffiths
alan@octopull.demon.co.uk

Self Registering Classes – Taking polymorphism to the limit By Alan Bellingham

In this article, I wish to propose a method of allowing easy addition and removal of classes from an application. This will use registration of class-factory functions to emulate virtual constructors.

Introduction

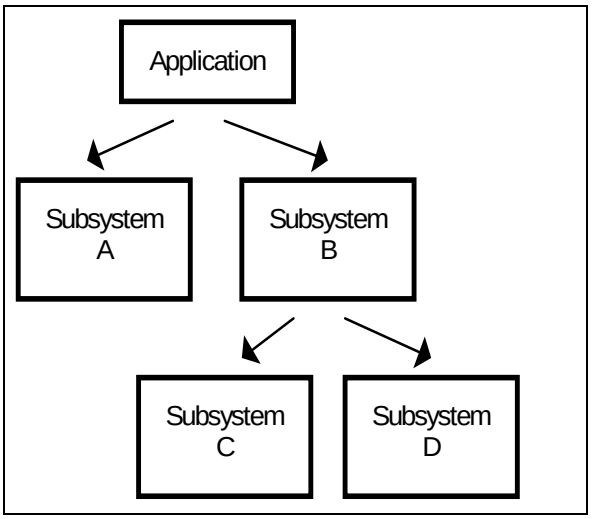
One of the main aims of an Object-Oriented programming language is to attempt to reduce coupling between the parts of a program by encapsulating the functionality and state of data structures within class instances, and for those classes to expose as little as possible to the outside world. Taken to an extreme, this becomes component-based software development, in which an application may comprise components written using a variety of languages and possibly running on disparate machines and architectures, but for now, we'll consider a single monolithic application.

Coupling

Firstly, what is the coupling problem?

Simply stated, it's the tendency for a subsystem A to know how subsystem B works, and vice versa. Any change to A requires a change to B, any change to B requires a change to A. Extend this to subsystems C, D and E, and a combinatorial explosion of dependencies occurs. Since larger systems tend to have more subsystems, one of the primary tasks of the software engineer on such projects is to avoid such reciprocal knowledge.

Ideally, then, a subsystem should have no knowledge of any subsystem that knows about it, and the grand design then tends toward the composition of more complex subsystems from simpler ones, somewhat like this, where an arrow means 'knows about':



In this case, whoever is implementing B doesn't need to know about A, and the implementor of C needs to know only about C.

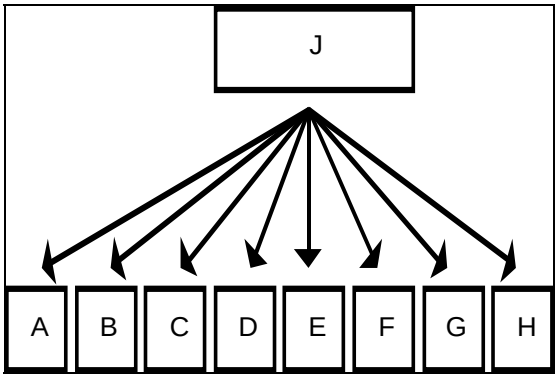
In general, an attempt to design in this way will lead to reduced maintenance problems, and produce cleaner code. It shouldn't be hard to see that conceptually each subsystem roughly corresponds either to a single class,

or to a class with helper classes that the client need not know about.

Back to reality

In real life, it's rarely this easy. Subsystems may need to notify their parents of changes, proxy classes may be returned that multiple subsystems need to understand, and the result becomes somewhat more of a cobweb. However, with suitable use of callback functions, notifications mean that a subsystem doesn't actually know anything about its owner, and common classes should be considered almost as built-in types and changed about as frequently ☺.

However, there is another potential problem, and that is hinted at by the "Law of 5 plus or minus 2". It is well known that human beings have problems really understanding what's going on when a large number of entities is under consideration, unless all the entities are the same, as in an array or list. In this case, consider the following:



In this case, subsystem J has to know how all the subsystems from A to H all work. However, much of the time, many of these subsystems, although different in detail, do similar work, and this is where a language such as C++ can simplify things by presenting all of these as being effectively the same class, by allowing the designer to use polymorphism.

By providing an abstract base class which exposes a common interface for all of these classes, instead of 9 subsystems A to I, we should be able to treat it as 9 copies of a single subsystem that just happen to be different internally.

The problem of creation

Indeed, careful use of C++ virtual functions does allow us to use polymorphism to dramatically reduce the number of times that an owner actually has to know about which concrete class it is currently using. However, there is one major function that cannot be made virtual: the constructor. As a result, there is often a switch statement, that looks something like this:

```
void Figure1Func(int objectType, int
param)
{
  GraphicItem * AC = NULL ;
  switch(objectType)
  {
  case 0:
    AC = new TextItem(param) ; break ;
  case 1:
    AC = new Box(param) ; break ;
  //...
  case 99:
    AC = new FilledEllipse(param); break;
  }

  if (AC)
  {
    AC->DoWhatever();
    delete AC ;
  }
}
```

Figure 1 - calling constructors from a switch statement

Also, it is frequently the case that there will be a requirement to serialise such items in or out of memory. Serialising out is easy - it just requires a suitable virtual function call, and the object will write itself out. Serialising into memory, though, is harder - because there is no existing object that can be called that is known to be of the right type. So, a switch statement will occur there as well:

```
void Figure2Func(istream& inputstream)
{
  int objectType ;
  GraphicItem * AC = NULL ;

  inputstream >> objectType ;
  switch(objectType)
  {
    case 0:
      AC = new TextItem(inputstream) ;
      break ;
    case 1:
      AC = new Box(inputstream) ;
      break ;
    // ...
    case 99:
```

```
      AC= new FilledEllipse(inputstream);
    break ;
  }
}
```

Figure 2 - serialising from a switch statement

If the application is only ever to have a fixed number of such classes, there wouldn't be too much of a problem. Unfortunately for software developers, there is rarely such a creature as a finished program. New classes get added in. Special versions get written that have classes deliberately *left out*. Menus exist listing the options, and these need to be changed. Sooner or later, someone is going to miss updating the switch statements correctly, and all hell will be let loose.

Banishing the constructor

The whole problem is that the owner has to know exactly what concrete classes are available. It would be so much simpler if a list could be built automatically. And who knows better than the classes themselves?

Consider a class:

```
class GraphicItem
{
protected:
  GraphicItem(int param) { ; }

public:
  virtual ~GraphicItem () = 0 ;
  virtual void DoWhatever () = 0 ;
} ;
```

Figure 3a: GraphicItem.h

We may then derive the concrete types from it, like this:

```
class FilledEllipse : public GraphicItem
{
private:
  FilledEllipse(int param) ;

public:
  virtual ~ FilledEllipse () ;
  virtual void DoWhatever () ;

  static GraphicItem *
                Construct (int param) ;
  enum { ID = 99 } ;
  //  Different for each class
} ;
```

Figure 3b: FilledEllipse.h

This class has a private constructor, and a public class factory function - i.e., a function that returns a constructed instance of the class. The class factory function actually uses the private constructor.

We could have a table (or better yet, a map), of these class factory functions against class IDs, and the client code could then scan the table for the right function to call in order to construct a new FilledEllipse given only an ID:

```
#include "GraphicItem.h"
// typedefs to reduce typing later
//
typedef GraphicItem *
        (*ClassFactoryFn)( int params) ;
typedef std::map<int, ClassFactoryFn>
FactoryMapType ;
typedef FactoryMapType::const_iterator
FactoryMapIter ;

FactoryMapType FactoryMap ;

//  Somehow FactoryMap is initialised ...

void Figure5Func(int objectType, int
param)
{
  FactoryMapIter it =
            FactoryMap.find(objectType) ;
  if ( it != FactoryMap.end())
  {
    GraphicItem * AC =
                  (*it).second(param) ;
    AC->DoWhatever();
    delete AC ;
  }
}
```

Figure 4: using a factory map

You will see that, if FactoryMap is constructed to contain object IDs and function pointers to the class factories, the client has no idea at all what the real objects constructed are. This is polymorphism taken to the limit. Note especially that it doesn't have to include the subsidiary include files for the individual concrete types - all it needs to know is listed in the abstract base class declaration.

Since there should only be a single instance of the Factory and it should exist for the whole program run, it's probably best implemented using the pattern:

```
FactoryMapType& FactoryMap()
{
  static FactoryMapType FMT ;
```

```
  return FMT ;
}
```

Figure 5: a singleton factory map

This means that anything attempting to access it cannot see it before it's constructed.

Building the class factory map

"Aha," I hear you say, "this has only moved the problem elsewhere. Something has to build the Factory map, and that something has to know about the functions."

Well, not quite.

What if the classes themselves cooperate in building the map, or at least, helper classes do. All the client has to supply is a function for the classes to register themselves:

```
void RegisterFactory(int ID,
ClassFactoryFn fn)
{
  FactoryMap()[ID] = fn ;
}
```

Figure 6a: registering with the factory

Now all that is required is to ensure that this function is called for each of the classes. That can be done by a helper class:

```
template<class T> class FactoryRegistrar
{
public:
  FactoryRegistrar()
  {
  RegisterFactory(T::ID, T::Construct);
  }
} ;
```

Figure 6b: FactoryRegistrar.h

```
#include "FactoryRegistrar.h"
#include "FilledEllipse.h"

static FactoryRegistrar<FilledEllipse> FRFE ;

//  Implementation of FilledEllipse
```

Figure 6c: FilledEllipse.cpp

The construction of the static helper class does the class registration. Assuming one module per concrete object, then all that needs to be done is to link the required modules to the main client code, and on program startup, the FactoryRegistrars get constructed, the class factory functions get registered and the client suddenly "knows" about the available classes.

The snake in the grass

But there is a problem with this approach. In fact, there are two, closely related.

According to the ISO C++ Standard, §3.6.2 (Initialization of non-local objects [basic.start.init]):

"It is implementation-defined whether the dynamic initialization (_dcl.init_, _class.static_, class.ctor_, _class.expl.init_) of an object of namespace scope with static storage duration is done before the first statement of main or deferred to any point in time after the first statement of main but before the first use of a function or object defined in the same translation unit."

This means that the implementation *may* decide not to construct our FactoryRegistrar at all, since until it has been constructed, there is no way that any function or object in that translation unit is used.

Secondly, it might be useful to build a library of these classes. However, modern linkers making use of such a library will only include those units which they can see are used. Again, because no function call is made into these units, the linker will totally ignore them. This becomes even more obvious when you consider a set of ten classes, of which you want five - only pure telepathy on the part of the linker would help it.

So, we need an answer.

The huge source unit option

The first method is crude, but it should work - compiler limits aside. Simply create a source file that *will* be linked in, and #include within it all the source files for the classes you want. It will also need a function called within it before the Factory map is used for the first time:

```
void InitGraphics ()
{
}
```

```
// Change these lines to change
// which classes are available
//
#include “FilledEllipse.cpp”
#include “Box.cpp”
```

Figure 7: AllGraphics.cpp

You’ll need to ensure that the headers can be multiply included, and it would be an extremely good idea to put the contents of each of the sources within its own namespace. This solution means that the statics should be constructed, as long as some function in this unit gets called. However, putting the classes into a library is no longer possible, and a full compilation of this unit is required, which may be quite time consuming, whenever a configuration change occurs.

The one call option

An alternative method is somewhat cleaner. Again, we define a function that the client code should call. But this time, it calls a function in each of the class units to be used in this configuration:

```
extern void InitialiseFilledEllipse() ;
extern void InitialiseBox();

void InitGraphics ()
{
  // Change these lines to change
  // which classes are available
  //
  InitialiseFilledEllipse() ;
  InitialiseBox() ;
}
```

Figure 8a: AllGraphics.cpp

```
#include “FactoryRegistrar.h”
#include “FilledEllipse.h”

void InitialiseFilledEllipse()
{
  static FactoryRegistrar<FilledEllipse> FRFE;
}

// Implementation of FilledEllipse
```

Figure 8b: FilledEllipse.cpp

Now we can place the class units into a library, and because we know that the class factory registrar *will* be constructed, we know that the class factories *will* be registered. Also, when a configuration is changed, it’s a much smaller unit that gets recompiled.

Cleaning up

By now, we have a two functions that are global, but that deal with the singleton FactoryMap, either directly or indirectly: RegisterFactory() and InitGraphics(). It makes sense to make them member functions of the FactoryMap itself, and for the functionality in InitGraphics() to be called by the constructor. So let’s see what our final result looks like:

```
class GraphicItem
{
protected:
  GraphicItem(int param) { ; }

public:
  virtual ~GraphicItem () = 0 ;
  virtual void DoWhatever () = 0 ;
} ;
```

GraphicItem.h

```
#include "GraphicItem.h"
#include <map>

typedef GraphicItem * (*ClassFactoryFn)( int
param) ;

class GraphicsFactoryMapImpl : public
std::map<int, ClassFactoryFn>
{
public:
  GraphicsFactoryMapImpl() ;
  void Register(int ID, ClassFactoryFn fn) ;
} ;

typedef GraphicsFactoryMapImpl::const_iterator
GraphicsFactoryIter ;

GraphicsFactoryMapImpl& GraphicsFactoryMap() ;

template<class T> class
GraphicsFactoryRegistrar
{
public:
  GraphicsFactoryRegistrar()
  {
    GraphicsFactoryMap().
                Register(T::ID, T::Construct);
  }
} ;
```

GraphicsFactoryMap.h

```
#include "GraphicsFactoryMap.h"

GraphicsFactoryMapImpl &
GraphicsFactoryMap()
{
  static GraphicsFactoryMapImpl FMT ;
  return FMT ;
}

#define INCLUDE_UNIT(a) extern void
Initialise##a();Initialise##a() ;

GraphicsFactoryMapImpl::GraphicsFactoryMa
pImpl()
```

```
{
  // Change these lines to change
  // which classes are available
  //
  INCLUDE_UNIT(FilledEllipse)
  INCLUDE_UNIT(Box)
}

void GraphicsFactoryMapImpl::Register(int
ID, ClassFactoryFn fn)
{
  (*this)[ID] = fn ;
}
```

GraphicsFactoryMap.cpp

```
//  No need for a separate header
// since nothing else includes it
//
#include "GraphicsFactoryMap.h"

namespace {
class FilledEllipse : public GraphicItem
{
private:
  FilledEllipse(std::string params) ;

public:
  virtual ~FilledEllipse () ;
  virtual void DoWhatever () ;

  static GraphicItem *
                  Construct(int param) ;
  enum { ID = 99 } ;
} ;

// Actual implementation here ...

} /* namespace anonymous */

extern void InitialiseFilledEllipse () ;
void InitialiseFilledEllipse ()
{
 static
GraphicsFactoryRegistrar<FilledEllipse>
GFR ;
}
```

FilledEllipse.cpp

```
#include "GraphicsFactoryMap.h"

void SomeFunc(int objectType, int param)
{
 GraphicsFactoryIter it =
GraphicsFactoryMap().find(objectType) ;
 if ( it != GraphicsFactoryMap().end())
 {
  GraphicItem * AC = (*it).second(param)
;
  AC->DoWhatever();
  delete AC ;
 }
}
```

Actual usage

Conclusion

In reality, there are likely to be more functions than just a simple class factory that will want to be registered - and it's quite feasible that the registration will insert string descriptions into menus as well. This example should be sufficient to demonstrate a methodology that can be extended to such cases safely and easily.

Alan Bellingham
alanb@episys.com

Whiteboard

The Harpist has been writing a series of articles discussing the design and implementation decisions made for a sample case study. Roger Lever and Detlef Vollmann have written questioning some of the decisions made so far.

Hotel Case Study Comments By Roger Lever

The code review for the Hotel application illustrated a number of useful points, such as the use of the canonical class form and the liberal use of const. But, these are all implementation level details.

Well defined objectives are the key to good design. From the article we don't know what all of the objectives are, but that doesn't stop us from making some intelligent guesses! At a high level we want to achieve a number of good 'ilities' such as maintainability and extensibility as part of the objectives. However, we want to achieve a balance between these and delivering a timely and effective solution.

Achieving this balance is difficult because of the number of competing and possibly conflicting project requirements. Considering top level issues provides a valuable scoping mechanism. It forms a basis for documenting

design assumptions, which typically are not included in project documentation. This applies to corporate developer and hobbyist alike. So, let us assume our objectives incorporate scalability, distribution, etc. That leaves us to focus on the application level design (the Hotel) and the important facets here are perspective, boundary and granularity.

Perspective addresses the focal point for the design of the application (presumably the hotel staff), who is it for, what do they want it do now, and later. It also provides the first part of the mechanism to scope reuse. Reuse is a term that has been applied at many different levels and has plenty of baggage with it particularly following the market hype. So to focus our thoughts on reuse here, we are concerned with being able to reuse design via patterns (which is not considered here) and code via reusable objects (components) and extension of code via inheritance and polymorphism.

Boundary, addresses the system boundary, what is within the system and by definition what is therefore excluded. This is also the second part of the mechanism to scope reuse. The questions that we are interested in here are what are the current boundary points and which ones are likely to extend outwards? For example, what information about a customer needs to be captured? If initially that is scoped to include only their name, and payment method is separate for now, that might later be extended to include their address for future marketing mailshots.

Granularity relates to the question of what level of detail do we need to decompose the problem. This helps to define internal boundaries achieving the appropriate balance between the general and the specific, the complex and the simple. In principle we want a design that exposes a simple and general interface and hides the specific and complex implementation. In our example, we want our hotel design to hide detail that is too complex or unnecessary. For example, our hotel might be built up from objects that are contained within a room, (bed, bathroom, TV, fridge) but is that necessary? Is it decomposing the rent-a-room issue to a level of detail that is not useful? However, we definitely need to know how many people the room can accommodate. In addition, if these rooms are conference rooms we will need to know what presentation equipment is available. This forms the third and final part of the mechanism to scope reuse, which is discussed next.

Now that we have a perspective, an idea on the system boundaries and the granularity that we need to work at, we need to reconsider these in terms of likely boundary extensions or future reuse. For example, initially all of the bedrooms or meeting rooms had the same equipment level and were only differentiated by capacity and price - is that likely to change? Will these rooms have further factors that we may need to model in our design? For example, adjoining bedrooms for families, or special facilities for children. What about other hotel facilities such as lounge, diner, bar or staff facilities? Considering the hotel design model in this fashion allows us to make meaningful choices regarding reuse and explicitly understand what choices we are making and why.

So, to look at the design in terms of perspective, boundary and granularity, we could start by producing a design such as below. The numbers refers to indentation levels.

(1) Hotel (Top level class to provide hotel facilities for customers)

(2) Facilities (customer facilites, free or paid for)

(3) HireFacilites (in use for a period of time)

(4) BedRooms (bedroom for one or more)

(4) Conference Rooms

(4) Presentation Equipment (used in meetings)

(3) Bar Lounge (recreation)

(3) Breakfast/Dining Room (restaurant facilities)

(2) Staff

(2) Private areas

(3) Kitchen

(3) Storage

To interpret this poor man's class inheritance diagram, a hotel is composed (HAS-A relationship) of customer facilities, which are either hired or rented for a period of time, or are generally available to customers. Notice that the Presentation Equipment is currently considered as part of the HireFacilities (IS-A relationship) since it is in use for a period of time and it is not a permanent fixture of a conference room. We can also reason about the design in terms of reuse and possible mechanisms to extend the design such as other hotel facilites (gymnasium, swimming pool...) or new HireFacilities (video equipment to record presentations). Does the current design support that?

So far we are considering reuse as primarily an extension mechanism (inheritance and polymorphism) and additional selective composition. We could also think in terms of reusable objects or components, such as the hotel itself. What interface specification should it support? Is that specification a general or specific case? Can we use the hotel generalisation in another similar context? What effort should we put into being able to use the hotel abstraction for similar concepts such as Bed and Breakfasts? Or could the HireFacilities concepts be used for a company to manage its own facilities?

In terms of perspective, boundary and granularity we may have created the above model of the hotel that is sufficient for our purposes. But, we will not really know that until we place it into context with other important classes that compose the solution's design. It is the essential Customer class that I wish to deal with next. From the implementation perspective Paul's original customer class had a number of problems (p19), however, although these were dealt with in terms of design consideration at implementation level coding with C++ (p20) I think it missed the point. Let me explain. A customer books one or more rooms for a defined time period and can pay in any number of ways (cash, cheque, credit card, account...) or that customer may not pay at all as the customer's company will pay the bill instead. Therefore we clearly have a number of concepts that emerge from this simple statement (a) Customer and (b) Payment. From the problem statement we already know that the Payment may not actually be related to the Customer, since it may be separately settled by the company. However, a further relationship ties these together, (c) Booking. The Hotel knows that once a booking is made, that room is now unavailable for a defined time period and a charge is due that will be paid. Therefore, in class design terms ,we could be considering the problem as:

(1) Booking (transaction that ties hotel, customer and payment together)

(2) Hotel (facilities being hired or rented)

(2) Customer (customer who will be using those facilities)

(2) Payment (mechanism to settle hotel bill for use of facilities)

We assume that the customer also books the facilities so need not model a separate entity for the booking customer. However, if a company secretary is booking the facilities then that may need to be captured for reference so we might do that like this:

(1) Booking (transaction ties hotel, customer and payment together)

(2) Hotel (facilities being hired or rented)

(2) Customer (name, address, telephone number)

(3) Guest (person going to the hotel)

(3) Contact (person doing the booking of facilities)

(2) Payment (mechanism to settle hotel bill for use of facilities)

Now we must question our Booking, Customer and Payment classes before looking

at their implementation. For example, one question might be why is Booking the top level class and is composed of a Hotel, Customer and Payment? Why not let the Hotel be a collection of Bookings? Good question! Perhaps booking could be a centralised function, as when a group of hotels has a central booking office. This saves duplication and offers other advantages such as a customer may find that one hotel, in the area, is fully booked but that another hotel has spaces. If the bookings were part of the hotel only then that facility would not be available to a would-be customer.

Another possibility is to look at this in terms of distributing logic and what it exposing interfaces. For example, if the hotel simply informs the booking office of what is available it can attend to all of the low level day-to-day management issues of staff, rooms, etc. The booking office concentrates on providing a single point of contact for prospective customers.

In summary, after a sufficent period of poking, prodding and adjustment to our overall class design we would move on to the implementation and coding level considerations, which is really the point where The Harpist's code review comes in. The general coding advice offered seems sound and is not something I would question, except (given the design above) that the CustomerRecord (p20) is trying to do too much. Also, an opportunity was missed to discuss the design in more general terms and to offer concrete advice on how to apply the concept of reuse to a design. But more than that, in not considering the design as opposed to the implementation, a useful separation of Customer and Payee did not occur. Consequently we have a CustomerRecord which mixes up static members and pointers in an effort to address what may be considered a flawed design.

Roger Level
RogerLever@aol.com

Hotel Case Study Comments By Detlef Vollmann

Dear Harpist,

First, I share your expierience of being a local expert, and so I'm not sure about my own ideas. But I put them in anyway and would like to receive your and other readers' comments.

I currently don't want to say anything about the general design, as I liked your comments on Paul's code. Some remarks about some subtle points might follow when I have seen more of your design. Here, I only want to share some thoughts about exception declarations. These thoughts relate to your following definition of class Customer:

```
class Customer
{
  string name;
  string payee;
  Customer(Customer const &);
  Customer & operator=(Customer const &);
public:
  Customer();
  ~Customer() throw();
  string const &getName() const throw();
  string const &getPayee() const throw();
};
```

Is it a good idea to put an empty exception specification to the read access functions? You write "...reading data should not cause an exception". But then you continue "...[this] might not always be the case", which is certainly true. With the empty exception declaration you give garanties about your class which unnecessarily narrow your possibilities to change your implementation later. E.g., you might later decide to store your Customer objects on a database, and your access functions will read directly from that DB. Then, these functions might well throw an exception. But if you then change your interface and define "string const & getName() const throw (DB_lost);" you might break existing code. And if you leave your empty exception declaration, and catch the possible exceptions inside getName, you have to handle the exception inside the class Customer, which might not be the best place

to handle environmental exceptions such as loosing the connection to the DB, and you prevent your client application, which might well be prepared to cope with the DB exception, to receive the exception and handle it. So, I believe its not so good an idea to add any exception specification to the access functions.

The same reasons which apply to the access functions hold true for the destructor as well. It might well be that in a future implementation of your Customer class the destructor has to commit some DB transactions and so has to throw an exception in case of a connection failure. I think, the long time proclaimed rule that a destructor should not throw any exception simply is not true. And this probably is excactly the reason why the standards committee added the "uncaught_ exception()" function. So, your destructor might do anything like this:

```
Customer::~Customer() // no exception
specification!
{
  if (uncaught_exception())
  {
    try
    {
      // do everything necessary, but
      // perhaps performing a rollback
      // instead of a commit, something
      // probably went wrong.
    }
    catch(...)
    {
      // this might do nothing, or might
      // set a global flag or anything
      // else to signal the ignored
      // exception to the client app
      // do not rethrow the exception!
    }
  }
  else
  {
    // normal destructor execution,
    // which well might throw an
    // exception
  }
}
```

So, you prevent an exception leaking out of a destructor only in case of stack unwinding due to another exception, but allow normal exception handling otherwise.

What's the bottom line? Should you omit exception specifications completely? Perhaps, this is the easiest way which gives you maximum flexibility for the future. But this flexibility is not always required. For me (and this goes much to general design questions), there are different kinds of C++ classes. You have general application classes (which typically map directly to corresponding classes from the analysis), which essentially give the interface which is used by all your application programs. For these classes, you need maximum implementation flexibility.

On the other hand, you have basic building blocks, or components, which you use to implement the general application classes. E.g. you might have classes like SimpleCustomer (which just implements the interface of Customer straightforward directly in memory), DBCustomer (which maps the class to a DB table), CorbaCustomer (which uses a Customer object anywhere in your distributed network), etc. These classes state their implementation in their names and interfaces, and so might well give garanties about exceptions without locking future implementation changes more than they are anyway locked by the name.

Of course, there are cases where (empty) exception declarations are absolutely necessary, e.g. for most member functions of an exception class itself, but this is not the scope of my remarks here.

These are my thoughts on exception declarations, but certainly there are other opinions on this topic, and I would like to see them.

Detlef Vollmann
dv@vollmann.ch

Object (low-level) Design and Implementation by The Harpist

I was delighted to receive not just one but two responses to my last article, see the preceding articles. Before I go any further let me respond to the substance of these items.

Using exception specifiers

The introduction of exceptions into C++ raises a number of design issues and it has taken several years for the best C++ programmers to refine their understanding of their correct use. The concept of an exception specification caused considerable trouble. It is my understanding that the UK originally wanted them removed because they could not be used for static checking of code. That left the problem that they required a runtime feature to support them.

While in the strictest terms this is correct, exception specifications provide a number of positive benefits. While complete static checking cannot be provided, some static checking is possible. For example:

```
void fn() throw()
{
  Mytype * ptr = new Mytype;
  // rest of function
}
```

Can be checked. It does not catch the `bad_alloc` exception that new can produce and so is clearly making a promise that cannot be kept. Any halfway reasonable compiler should raise an objection to such code.

The second thing, that exception specifiers provide, is a statement of intent for the benefit of other programmers. It is a condition applied to the function, and like all other features of a declaration it provides a constraint that users can (or should be able to) rely on. Once I decide, as part of my low-level design, that a function does not allow exceptions to leak it is a commitment that I must abide by. Like the return type, a throw specifier is part of the signature of a function that cannot be overloaded.

Readers of the latest edition of '*The C++ Programming Language*' will know that there are some clever fixes that can be applied to handle functions that are not supposed to throw exceptions by providing special versions of the handler for '`unexpected`', but I will leave that to experts.

Whether read functions should or should not have an empty exception specifier is a class design decision, however the logic of Detlef's letter would be that we should never use exception specifiers because they commit us for all time to a specific policy with regards to a function. I find this too negative. So, let me explore some options.

The first is to provide overloading via an extra dummy parameter. For example, suppose we declare a global `enum` type:

```
enum CanThrow {canThrow};
```

Now suitable pairs of functions can co-exist:

```
string const & getName() const throw();
string const & getName(CanThrow) const;
```

This empowers the user of the `Customer` class to write either:

```
cout << customer.getName();
```

or

```
cout << customer.getName(canThrow);
```

depending on whether the user wants to handle exceptions or not. That means that the version with an empty exception specifier must handle exceptions internally and provide some dummy return if no genuine value is available. The definition of the 'throwing' version should use the anonymous parameter facility to handle the `CanThrow` parameter because there is no practical use of the parameter in the body of the function. The parameter is purely to provide overloading, and the type name and value are chosen to alert the user to the need to handle exceptions.

The second option is to have a specific exception type as the only one that can be thrown by the function. Something along the lines of:

```
enum NoName{noName};
string const & getName() const
throw(NoName);
```

and the definition would be:

```
string const & getName() const
throw(NoName)
{
  try
  {
    // body of function
  }
  catch (…) { throw noName; }
}
```

Of course there could be many more catch clauses that handled individual problems but each would terminate with either a return of some string or with `throw(noName)`.

Programmers should know which exceptions they may have to handle. Until library designers get in the habit of providing exception specifiers, programmers must assume that all exceptions may need handling. We can argue about the merits of different strategies, but pretending that we need do nothing isn't a professional option. Like too many things however, exception specifiers are going to be ignored by most authors of books because they will seem like just another complication.

Exceptions & Destructors

While I agree with Detlef that it is a (low-level) design decision as to what exception specifier should be attached to ordinary member functions I completely disagree when it comes to destructors. I think he has misunderstood the purpose of `uncaught_exception()`. But I will return to that in a moment.

Constructors can and should be able to throw exceptions. If something goes wrong during the process of constructing an object some way is needed to get your program back onto safe ground. The biggest problem was finding a mechanism to handle an exception thrown from some part of a constructor-initialiser list. I believe that this problem actually generated some new syntax so that entire function definitions could be encapsulated in a try block but I have never seen this used. Suffice to say that the exception mechanism is particularly useful for dealing with problems during the process of construction.

But what about the other end of an object's life? Suppose that a destructor throws an exception, what am I supposed to do? In general all I will know is that I am handling an exception, no clue that I have an incompletely destroyed object on my hands. For example, suppose that I have some local object that handles a file and a serial port. The destructor is called for the object when the function is cleaning up before returning. Something happens during the process of closing the file that results in an exception, unless that exception is handled locally the serial port is never released. OK that is a bit obvious and the programmer of the destructor should handle that but what if he doesn't? Your program is now unstable and should raise an '`unexpected`' exception.

I am not going to claim that no destructor should **ever**, under any circumstances, throw an exception. What I do claim is that if a designer finds it necessary to allow a destructor to throw then the exact nature of the exception must be documented and a full justification for allowing it should be required.

In my opinion, destructors should always have exception specifiers. In the overwhelming majority such a specifier should be empty. I believe that writing a destructor without an exception specifier is unprofessional and a sign of incompetence or ignorance. Yes we all forget sometimes, but we should be embarrassed if it happens very often.

Now a brief word about '`uncaught_exception`'. When we write functions that may be used inside exception handlers we have to consider that possibility and arrange some tolerable behaviour (if possible) when they would normally throw an exception. The purpose of `uncaught_exception()` is to provide the tool that programmers can use when they have no other viable alternative. This is particularly true of mission critical programs that must not abort. Every effort should be made to ensure that functions used during the process of handling an exception do not

throw exceptions. Only in the most unusual circumstances might you tolerate different behaviour from a function depending upon whether it was called during exception handling or otherwise. Such special behaviour during EH would be some compromise (such as letting a resource leak) that was undesirable but less so than aborting the process.

I would welcome alternative views on this subject because I currently can see no justification for allowing a destructor licence to throw anything and everything.

Design Issues

I am very grateful for Roger Lever's thoughtful commentary on the subject of design. I think one problem is that the term is used in several ways. I think that I ma largely focused on the low-level aspects. To me, design is a matter of deciding what a class (or function) shall do while implementation is a matter of deciding how it shall do it. I consider design to be a matter of deciding what the interfaces of a class shall be. Roger, quite correctly, is taking the broader view that design is a matter of deciding what a class is for. Let me try to elucidate, and Roger can come back next time to correct me as appropriate.

What constitutes a 'room' depends upon whose viewpoint you take. An architect has one view, and architectural engineer (responsible for considering such things as the loading on floors, the stresses on walls etc.) has another. The architect might be concerned about the placement of windows, the shape of the room, the location of a fireplace etc. without too much regard as to other rooms adjacent to the one in focus. The architectural engineer has to consider what is adjacent. It is her job to note external walls and the potential for heat loss, the existence of upper floors with the consequential requirements for load bearing walls.

One of the UK TV channels has been running a series of programs on design. The second of these was about designing a new toilet for a leading UK manufacturer of bathroom suites. They had commissioned two designers. It became clear during the course of the program that the designers and the company directors meant very different things be a design and design brief. The company was mainly concerned with the external appearance and just wanted a new (but not too new) 'shape'. The designers wanted to consider the function and produce something that better met the needs of male and female users, was easier to keep clean etc. There was another aspect to this in that those actually responsible for production (the 'implementors') had another view – what could practically be produced by the equipment available. Moulds must work, the items must be fired without too much wastage etc.

Professional designers should provide design documents for their code. These should be based on an understanding as to what aspects of objects are to be represented. The hotel designer, the builder and the receptionist have very different views as to what is important about a room.

In the days before we focused strongly on reuse the context of the application we were writing implicitly defined the design (making it explicit would have been a good thing and much of the abuse of code by cut and paste coding might have been avoided had programmers had a better understanding of the relevance of viewpoint to code design). Now that we increasingly focus on reuse we need to be conscious of reuse at all levels.

My view (and I think that intended by Paul) is that of the manager of the hotel. Roger suggests that we should up this a level to that of the manager of a chain of hotels. I think this is an excellent extension and the kind of thing that comes with increased experience of using object-oriented techniques. However I am mainly focusing on low-level design because, no matter how elegant the high-level design, without good low-level design everything falls apart. Look at an ordinary building brick. There is a lot of low-level design involved. For example, the shape is a cuboid whose dimensions are approximately 3:2:1. The dimensions are intended to be

exactly 3:2:1 when the thickness of the mortar is taken into account. If you did not know how bricks were used you might be puzzled by the approximations.

Objects & Copying

There seems to be a widely held belief that the default behaviour of providing copy constructors and copy assignment is correct. I reject this. Consider my favourite 'playing card' type. How many Spade Aces should there be in a pack of cards? One, and if you were playing Poker and two Spade Aces turned up you would know someone was cheating. Each card in a back is a unique item in context. It might be possible to duplicate that item but such duplication should be a careful and considered action, not some by-product of a desire to have a second object that was identical to the first. This is even more the case when it comes to assignment. It should be completely meaningless to assign one object to another.

I think that object types should never have public copy constructors and copy assignments. Sometimes it may be desirable to provide such functionality `private`ly, or even to other class designers via the protected interface. The existence of a public copy constructor is what distinguishes a value type from and object type. Values may be freely copied, objects should only be cloned. If you do not understand this distinction you do not understand object based/oriented programming.

Unfortunately we get very casual about our use of terminology. We often talk about throwing an exception object. We should never do this. We should throw an exception value (remember that exception 'objects' are always copied to the point where they are caught). This looseness does not matter as long as we understand what we mean, sadly many of those listening do not and so get confused.

So let me consider my `Customer` type. Should this be a value or an object type? I think we must be careful about what we mean. There is nothing to prevent us from having multiple but identical objects. Indeed my junk mail shows that many companies are quite happy with having multiple instances of me in their databases. What I am asking is should we allow a '`Customer`' to be copied without explicitly choosing to do so? My feeling is that the answer should be 'no'.

There is a problem with strictly adhering to the concept of an object and removing publicly available copy constructors: all the STL containers are value based. In other words the STL containers require access to copy constructors. We would expect to be able to produce a customer list, yet to do so we must provide access to a copy constructor. Before we consider possible solutions we must ask ourselves about our concept of a customer and how we expect it to be used. Is 'customer' intended to be a base class? In other words, do we expect to derive from customer? If so we cannot have a simple container of customers because the STL containers do not work well with polymorphic types (unless they all have the same size, which is unlikely). If we want to manage collections of polymorphic objects we must provide a surrogate or handle type, a smart pointer or use a raw pointer. A suitably designed smart pointer (not, PLEASE, `auto_ptr`, because that was not designed for such use) would be best because it would handle extensions to customer easily (the cost is in designing the smart pointer, anyone offer a smart pointer for container use?) might be best but a well designed surrogate would be good as well. I would not be keen on using raw pointers as they would be responsible for large scale resource leakage.

Our collections would have to manage our objects via (smart-)pointers or surrogates which might have public copy constructors. Actually, I am slightly uneasy with the concept of a surrogate with a public copy constructor.

On the other hand if you have an essentially non-polymorphic object type (playing cards would be a good example) then we can fix the problem in a different way. Let me give you an example:

```
class PlayingCard
{
  friend vector<PlayingCard>;
  PlayingCard(PlayingCard const&);
  void operator =(PlayingCard const &);
  // rest of class interfaces
};
```

By making `vector<PlayingCard>` a friend of `PlayingCard` I have provided it access to the `private` copy constructor. Of course the only containers you can have will be vectors, perhaps you might want to add:

```
friend list<PlayingCard>;
```

as well.

I think that this is a legitimate use for `friend`. What do you think? I wish that there was a way to provide special access to the `protected` interface so that I could grant special access rights to third parties without having to go the whole way and give them access to everything.

Mixins

I am never very happy with this term and suspect that it is often misused. I understand that it originated from the idea of basic ice creams to which a selection of extras could be added. In programming terms it seems to refer to a basic class to which various extras can be added by multiple inheritance (Java, I guess, would use interfaces for this purpose). The idea is that these extras are free standing abstract base classes that represent some specific abstraction. In the context of our hotel as a commercial enterprise we have a couple of candidates for 'mixins'.

The concept of being hireable is one that applies to much more than rooms and presentation equipment. Complementary with the concept of being hireable is the concept of being billable.

`Hireable` might be provided by something along the lines of:

```
class Hireable
{
  ChargeInfo * rates;
public:
  Hireable(ChargeInfo *lookup = 0)
    : rates(lookup){}
  // despite the pointer,
  // shallow copies work
  Currency getRate(TimePeriod)
                  throw(Invalid) const;
  void setRate(ChargeInfo *) throw ();
  virtual ~Hireable() throw() = 0;
};
```

This class raises a number of issues. The first is that several other ADTs naturally arise and will have to be designed and implemented. Anything that is hireable will have to have some form of rate-table. We will also need some form of time information (hourly, daily, weekly etc.) and something to represent the currency used. I am not providing details of these but have added them to highlight the kind of thing that starts to happen as you try to work in an OO fashion. It would seem that `ChargeInfo` should be some kind of external data structure that can be accessed with `TimePeriod` data. I have used a pointer rather than a reference because it seems likely that you might want to replace the rate-table, you will also need to handle the creation of hireable objects even if you do not know what rate-table to use. The nature of `ChargeInfo` is left for consideration by the designer of that class with the proviso that it should work with the `TimePeriod` class to generate `Currency` information.

There is another interesting aspect of this class in that it is a user of `ChargeInfo` but is not responsible for its creation. That means that the raw pointer can be copied by both copy-constructor and copy assignment. It is not always the case that you must provide the copying functions if one or more data element is a pointer. On the other hand this is a risky technique because we are using a pointer to data that is outside the control of the object. Such pointers are always vulnerable to becoming hanging pointers if the object they are pointing to is removed or relocated.

The reason for taking this risk is that many objects may need to share a look-up table of rates. Such a table would be subject to amendment and so needs to be unique. There is another option. We can allow each object to hold its own local copy and register this information with the master copy. The

functions that change the master copy would then be responsible for notifying the copy-holders. The destructor for the master would then be responsible for notifying all current holders of local copies to reset their pointers either to null or to some substitute. In the long term a technique such as this is preferable and professional class designers should be familiar with the idea and the principles for implementing it. Experience suggests that few are.

The reason that an empty destructor has been declared is to provide a hook for making `Hireable` an abstract base class. `Hireable` is an abstraction and we do not want free standing instances. What we want to be able to produce is something like:

```
class RentableRoom :
    public Room,
    public Hireable
{
  // what ever
};
```

As we think deeper and deeper into this problem we become aware of many other classes that we should work on. For example we will need to consider the payment method (cash, credit card, cheque etc.) This seems a good target for a class hierarchy with an abstract base class `PaymentMethod` and shallow hierarchy to provide the various options.

I think it is because of this requirement to add layers of classes that so many programmers retreat to simple non-reusable solutions.

This article is already late and getting rather long. I think it is about time that I looked at some more of Paul's code. This time I am going to look at some of his implementation.

Implementing the Original Customer Class

```
#include <iostream.h>
#include <string.h>
const int maxName = 30;
// reserve storage for the static
int Customer::customerCount;
Customer::Customer(
{
  char temp[maxName];
  int size;
  cout << "Enter customer name:  ";
  cin >> temp;
  size = strlen(temp);
  name = new char[size + 1];
  strcpy(name, temp);
  cout << "Enter payee name";
  cin >> temp;
  size = strlen(temp);
  payee = new char[size + 1];
  strcpy(payee, temp);
  customerCount++;
}
```

When we look at the above code we will realise that several poor decisions relate back to his original design. The pollution of the global namespace by maxName (an unfortunate choice of identifier as it is certain to be popular in other code written at the same level of expertise – a good reason for hiding such in a named namespace) can be avoided by recognising that the only code that depends upon this value is the temporary array of char used to capture the data. In such a case the manifest constant should be declared close to its point of use (like immediately before its first use). Of course once we feel comfortable with using string instead of char[] the problem goes away, though might still want to apply some form of validation by restricting the number of characters used. I always feel unhappy with the use of int for the sizes of things. Surely this should either be size_t or unsigned int?

The next problem is that no attempt has been made to ensure that the input does not over-write the provided storage, nor has code been provided to handle names that include embedded whitespace.

I offer the following re-write (I am focusing on implementation here because writing good implementation code is also important).

```
// select standard library identifiers
using std::cin;
using std::cout;
using std::istream::get;
// reserve storage for the static
int Customer::customerCount = 0;
Customer::Customer()
{
  const int maxNameLength = 30;
  char temp[maxNameLength];
  size_t size;
  cout << "Enter customer name:  ";
  cin.get(temp, maxNameLength);
  size = strlen(temp);
```

```
  // clear input buffer
  while(cin.get()!= '\n');
  name = new char[size + 1];
  strcpy(name, temp);
  cout << "Enter payee name";
  cin.get(temp, maxNameLength);
  size = strlen(temp);
  // clear input buffer
  while(cin.get()!= '\n');
  name = new char[size + 1];
  strcpy(payee, temp);
  customerCount++;
}
```

As I wrote this I became very conscious that a large chunk of that code is almost duplicated. That provides a maintenance problem as well as making the function larger than necessary (pragmatically the error **rate** goes up as function size increases). Consider the following alternative:

```
void initName(char const * prompt, char *
& dest)
{
  const int maxNameLength = 30;
  char temp[maxNameLength];
  size_t size;
  cout << prompt;
  cin.get(temp, maxNameLength);
  size = strlen(temp);
  // clear input buffer
  while(cin.get()!= '\n');
  dest = new char[size + 1];
  strcpy(dest, temp);
}

Customer::Customer()
{
  initName("Customer name:  ", name);
  initName("Payee name)", payee);
  customerCount++;
}
```

Note the type of the second parameter of initName(). This handles a situation that many programmers get wrong. I want to pass a pointer for modification. By passing a reference to a pointer I make it more likely that I will write what I intend. By the way should initName be a private member function? Perhaps it should be a utility function in namespace Harpist, or perhaps there should be a third parameter passing the maximum acceptable length. As I would use string instead of char[] I am not going to worry too much this time around, but this kind of small utility function is a prime candidate for very low-level reuse.

```
Customer::~Customer()
{ customerCount--;
  delete name;
  delete payee;
}
int Customer::getCustomerCount()
{
  return customerCount;
}
char* Customer::getName()
{
  return name;
}
char* Customer::getPayee()
{
  return payee;
}
```

Of course these last two functions are badly flawed because they provide write access to private data and hence allows fraudulent changes to the data. We know that the original design was faulty because it failed to qualify these functions as const (read only) and without that qualification we can get the return type wrong. With the qualification the compiler knows that we have just provided illegal write access to instance data. The various uses of const are designed to reinforce each other. However I am just improving the implementation of the original so these two functions should at least become:

```
char const * Customer::getName()
{
  return name;
}
char const * Customer::getPayee()
{
  return payee;
}
```

Well I think that is all I have time for this time. Keep the comments flowing so that we all become better C++ programmers.

The Harpist

Broadvision – Part 2
By Sean Corfield

Recap

In last issue's piece, I described what Broadvision was and gave a flavour for how it worked, looking in particular at extending some of its classes to work around design

blind spots. I said that this time I'd look at database access and encapsulating the Broadvision API as well as more information on session management. The pressures of work have meant that I haven't had much time to prepare this piece as I'd like - ironically, I'm working on another Broadvision-powered web site and it's keeping me very busy!

Sessions

I touched briefly on the concept of web site sessions in the first article in this series and in order to explain what follows, I shall elaborate a little on session management. The web is essentially stateless: you visit a site, the browser requests the page, the server fetches (or generates) it and sends it to your browser. At that point, you can request another page from that site or another and the browser maintains only a history of which pages you've requested. The server on the other hand, spends its time fetching or generating pages in response to requests from any number of users. For a personalised web experience, someone has to maintain information about your choices within a web site, for example a 'shopping basket' within an e-commerce (electronic commerce) web site.

Broadvision chooses to maintain this session 'state' information on the server. When you connect to a Broadvision site, it allocates a unique session 'ID' and uses it in every generated link and form so that with each incoming request, it can work out which of its currently active users is making the request. Naturally, an application built using Broadvision will need to maintain its own state information: an example from our travel site is the set of destination / price range / date range choices that you make as you wander around the site.

Broadvision provides a very simple interface for application data: an associative array of strings called the application data dictionary. In C++ terms, this is effectively map<string,string> but wrapped up in a session ID based lookup mechanism (i.e., map< SessionID, map<string,string> >). The base Broadvision class, Dyn_Object, provides 'store_app_data' and 'find_app_data' methods to access the application dictionary (thus hiding the session ID lookup - the object already knows about session IDs).

The first part of the application we tackled was the foreign currency section of the site. We needed to store the user's choices of currency and amounts for the conversion process. We tried to fit in with Broadvision's application dictionary by mapping our array of currencies, rates and amounts down to strings and then converting them back when we needed them. It worked but it was painfully ugly and very inefficient. I looked in vain for a more generic way to deal with state data - Broadvision had none.

After a bit of thought about the way Broadvision worked, I realised that I could solve the problem with static member data - the Broadvision CGI application ran continuously in the background so static data would have a suitable lifetime. So I wrote a small template class called 'SessionStore<T>' to store data of type 'T'. Since I could now store arbitrarily complex data structures on a per session basis, I decided to allow only a single object of each type to be stored for any one session and to have it accessed through a writable reference, i.e., the one and only copy of each state object for a session lived in its own 'SessionStore' container. We didn't bother rewriting the currency handler - after all, it worked and we were somewhat pressed for time - but we used the new 'SessionStore' for our search context objects and our customised shopping basket. It's a trivial template class and it seems an obvious omission from the Broadvision framework. Another design blind spot.

See API?

I've been very critical about many aspects of the Broadvision architecture, taking up design issues with everyone from technical support up through the development team in the USA. One of my main gripes about the architecture was that most of the API smelled like C. You know what I mean, you've seen this in other frameworks: a huge slab of unrelated API calls disguised as methods in an umbrella class. Broadvision suffers from this in spades:

pretty much the entire session management API lives in one class. In fact, it goes further than just session management, it also includes database access functions and many other tasks.

Naturally, I don't like this. If I'd wanted that architecture, I'd be programming in C. I like C++ because it can provide a better match to the problem domain by encapsulation. Part of our application generates emails from the web site to back office staff containing certain details of the user's profile (name, address etc). As part of our OO design, we wanted to pass a UserProfile object into the EmailRequest object so that it could interrogate the profile to fill out fields in the email message. Broadvision doesn't have a UserProfile object, instead it provides the equivalent of get_user_profile_field / set_user_profile_field as API calls. UserProfile became the first of many classes that we created to wrap up parts of the API so we could pass suitable objects around within our application. Like many such wrapping classes it is rather crude and was tedious to write:

```
class UserProfile
{
public:
  UserProfile() { }
  RWCString name() const
  {
  return get_user_profile_field("NAME");
  }
  UserProfile& name(RWCString n)
  {
    set_user_profile_field("NAME",n);
    return *this;
  }
  // about a dozen similar methods
};
```

Now, I don't think of myself as an OO purist, nor do I think I'm an unreasonable man, but given a framework for an application where concepts such as 'visitor', 'profile', 'shopping basket' are absolutely key, doesn't it seem somewhat disappointing that equivalent classes are not provided within the framework?

An object lesson

I'm going off on one of my tangents now, as I'm wont to do. I mentioned in the 'Recap' that the reason I haven't had as much time to work on this article as I'd have liked, is that I'm currently building another Broadvision web site. In fact, I'm not building it, I'm just designing it: through the vagaries of office politics and a natural evolution of my position at my client's, I no longer do any development per se, I'm a designer. I recently evaluated 'Together/J' from Peter Coad's Object International company (http://www.oi.com). 'Together' comes in several flavours dealing with C++ and Java (and other languages). I chose the Java version because I wanted to avoid the temptation of generating C++ and then slipping back into development (our target development language is C++). I downloaded the Whiteboard edition that provides object diagram functionality and the ability to browse 'use case' diagrams produced by the full edition. The object diagram editor has four panes: a navigation pane showing either a thumbnail of the entire diagram with a scrollable shadow or a directory hierarchy showing packages and their contents; an attribute pane allowing direct editing of object attributes; a diagram pane showing standard UML notation for classes and relationships; and a code pane showing Java class & method code for the corresponding elements of the class diagram. The latter is very slick: change the diagram or attributes and the code updates immediately, change the code and the diagram and attributes change to match. The whole product has a very responsive feel and is very flexible. I was sufficiently impressed to purchase the full version of the product which adds the ability to create and edit use case diagrams as well as state and sequence diagrams.

Using such a UML based CASE tool on a Broadvision project helps you separate the meat of the design from the continual workarounds that arise from struggling with the application framework. One of the first tasks was to create diagrammatic (and, hence, Java) representations of the main Broadvision classes on which to build the application-specific classes. It's been a much more pleasant way to work, being able to concentrate on the design and get that right. I'd like to think that the rigours of working

within a strict OO design framework have made me a better designer overall and I'm off on a QA Training course for OOA/D using UML shortly so I'll probably write up specific pieces both on the course and the 'Together' tool in due course.

What's next?

We've recently completed the data model for the current project and we've taken on board much of what we've learnt about Broadvision over the past year. We're approaching the current project from a rather different angle to the one I've been describing here. In the next article, I'll look at database access, as promised, but with a 'compare & contrast' showing how we fought with the framework last time and how we're attempting to work within it this time, hopefully illustrating how suitable compromises make frameworks more effective. Since we're trying to stick more closely within Broadvision's framework this time around, I'll also look at other aspects of Broadvision. In the fourth and final article in the series, I'll go back to more of the C++ customisation we've done - on both projects - and again, make comments about blind spots within the framework.

Sean A Corfield
sean.corfield@issolutions.co.uk

Reviews

Designing Components with the C++ STL

Author: Breymann

Published by: Addison-Wesley

ISBN: 0-201-17816-8

Format: Hardback 306pp

Price: 29.95 UKP

Supplied by: Addison-Wesley Longman

Target Audience

This book is intended for anyone involved in C++ development who already has a basic (I would say 'intermediate' at least) knowledge of C++. In three parts, the book begins with an introduction the STL, its concepts and components, moving on to a catalog of the STL algorithms illustrated with examples and finishing with examples and discussions about how to build applications and complex components on top of STL. That latter part of the book makes up about half of the book and is where the meat of the material lies. As usual with books of this type, all the code examples are available over the Internet.

The Book

Breymann moves the reader on at a cracking pace - ten pages into the introduction we've already moved from a simple array-based example to a full STL container / algorithm / iterator example with templates and then on to writing our own singly-linked list class with an iterator to illustrate how to make your own containers interact with STL. I initially had a bit of a problem with Breymann's narrative style - an artifact of the translation from German to English - but soon got used to his precise, if occasionally unusually constructed, prose. Regardless of this minor criticism, it is certainly a lot more readable than many of the pure reference materials available on STL.

After the initial introduction of concepts, Breymann lays bare iterators in a compact but comprehensive manner, then moves on to deal with containers. In both cases, the interfaces are explained in detail and certain implementation details are considered, partly to shine light on aspects of the interface but mainly to provide a deeper understanding of how the iterators and containers perform their tasks.

The second, central, section of the book covers all the algorithms of STL in about 70 pages. By its nature, this section is almost strictly a reference manual since their are so many algorithms. Breymann provides intelligent commentary throughout, especially

where several similar algorithms need clear explanation of their differences, e.g., the sorting and merging algorithms.

Moving on to the third section of the book, Breymann introduces various algorithms and containers that provide solutions to some of the annoying limitations of the STL specification. In each case he explains the benefits and drawbacks of the approach, illustrating each with clear code examples. A particularly good case in point is the chapter on 'Fast Associative Containers' which are implemented using hashing. Hash-based containers were omitted from the (draft) Standard C++ library mainly due to time constraints and the committee has been publically criticised for this omission. Without grinding any particular axe, Breymann provides full implementations of hash-based containers with intelligent commentary and analysis while conceding that no standard exists for these containers, therefore his are just examples.

In chapter 8, we move on to applications constructed on top of STL, cross-referencers, permuted index generators, thesaurus, matrices and so on. Some of these build on the raw STL, others on Breymann's own extensions to STL. Breymann shows how generic applications can be built that allow selection of implementations (using different STL-like containers) and discusses the performance tradeoffs involved. I was particularly impressed by chapter 11 which deals with data structures and algorithms for handling various types of graph (directed, undirected, cyclic, acyclic, weighted etc). Breymann manages to cram a lot of content into a seemingly tiny amount of pages yet the discussion, and code, make the solutions seem comprehensible to the point of simplicity.

Throughout the book, Breymann presents exercises - most with solutions given in the appendix - that test the reader's comprehension and are often non-trivial. In fact, the book scores highly on the integrity of its examples and exercises by dealing with complete programs that illustrate the concepts and components being described.

There are occasional technical errors (e.g., using 'InputIterator_type' in a template declaration followed by 'Input_iterator' within the body) but these are few and far between so readers are unlikely to be confused by them. The technical reviewers have clearly done a good job but with such a highly technical subject under discussion, one or two errors are bound to crawl by unnoticed.

Conclusion

Although the book is presented as more than a reference manual for STL, it is still a bit dry in places and might have benefitted from more discussion material, expanding it perhaps by a third. That said, it manages to cover a lot of ground in clear, concise text and examples and it should provide something of value to anyone either currently using STL or planning to in the near future. There is no doubt about Breymann's expertise and I certainly found some very useful information in the book, despite my long association with the machinations of the C++ committee - in particular the section on graph algorithms impressed me. For nearly thirty pounds, you have to consider whether it is 'the' STL book to own as there are quite a few around. I don't think it quite lives up to the hype of its title but as a reference manual for STL and a good jumping off point for applications built on top of it, this book deserves a very solid recommendation.

Sean A Corfield
sean.corfield@issolutions.co.uk

Beyond ACCU... Patterns on the 'net

C++

http://www.research.att.com/work/

This site alone is enough to keep you occupied all month, as long as you can do without eating or sleeping.

C++ and OOP articles by Bjarne Stroustrup can be downloaded.

- A brief look at C++
- A Perspective on ISO C++
- What is "Object-Oriented Programming"?
- Why C++ is not just an Object-Oriented Programming Language.

The site also has software tools that can be acquired, usually by Universities.

http://web1.ftech.net/~honeyg/articles/pda.htm

The Role of Patterns in Enterprise Architecture

www.sgi.com/Technology/STL/other_resources.html

A collection of STL links.

http://www.cyberdyne-object-sys.com/oofaq2/

Object FAQ. An ambitious web site that is being built around a FAQ.

Article source

http://www.byte.com/art/art.htm

BYTE articles archive. There are plenty of "brochure-ware" web sites for magazines but this seems to be one of the useful sites.

Bulk sites

These are sites providing masses of links or huge archives of files.

http://sunsite.doc.ic.ac.uk/

Plenty of material to keep UK web surfers occupied, this site is invaluable when the internet slows down to a crawl.

http://www.devinfo.com

"The developer information site" by Christopher Sokol, a gold mine of links for archives & languages.

http://www.jumbo.com/pages/developer/

A large archive of files, not just for development. I found it easier to navigate by using

http://www.devinfo.com/operating_/index.html.

It covers Linux, Mac-OS, MS-Dos, NeXTStep, OS/2, Unix and Windows

http://www.devinfo.com/networking/rfcs/index.html

An Internet Standards repository. This set of RFCs define how the internet interoperates.

http://developer.intel.com/design

Intel has a web site for developers. Strangely Intel is offering free CD copies of its website. I think this kind of behaviour should be encouraged - instead of having common downloads clogging up the internet, free CDs and magazine CDs are a good distribution medium.

Compiler resources

http://cuiwww.unige.ch/freecomp

Free compiler list can be searched or viewed by category. Worth visiting and searching for key items of interest - you may stumble across something you ought to know about. Unfortunately it is no longer being maintained...

www.geocities.com/SiliconValley/9498/watcom.html

Good source of Watcom C/C++ links.

http://www.cygnus.com/misc/gnu-win32/

GNU-Win32 - NT/95 port of the GNU development tools.

The GNU-Win32 tools are Win32 ports of the popular GNU development tools for Windows NT and 95. They function through the use of the Cygwin32 library which provides a UNIX-like API on top of the Win32 API.

- Develop Win32 console or GUI applications, using the Win32 API.
- Port significant UNIX programs to Windows NT/95 with few changes.
- Use many common UNIX utilities (from the bash shell or the standard Win32 shell).

Next issue... *Java* ?

Send links and suggestions to **ACCU.general**.

Credits

Editor

John Merrells
merrells@netscape.com

c/o Einar Nilsen-Nygaard
65 Beechlands Drive
Clarkston, GLASGOW, G76 7UX.
UK

P.O. Box 2336,
Sunnyvale, CA 94087-0336,
U.S.A.

Readers

Ray Hall
Ray@ashworth.demon.co.uk

Ian Bruntlett
IanBruntlett@duzzit.globalnet.co.uk

Einar Nilsen-Nygaard
EinarNN@atl.co.uk
einar@rhuagh.demon.co.uk

Production Editor

Alan Lenton
alan@ibgames.com

Advertising

John Washington
accuads@wash.demon.co.uk
Cartchers Farm, Carthouse Lane
Woking, Surrey, GU21 4XS

Membership and Subscription Enquiries

David Hodge
davidhodge@compuserve.com
31 Egerton Road
Bexhill-on-Sea, East Sussex. TN39 3HJ

Copyrights and Trademarks

Copy deadline

All articles intended for publication in *Overload 28* should be submitted to the editor by September 1st, and for *Overload 29* by November 1st

PAGE 30 IS THE 32nd PAGE.

IT IS RESERVED FOR THE BACK COVER