overload|79

June 2007
ISSN: 1354-3172
www.accu.org

Live and Learn with Retrospectives
Rachel Davies

auto_value: Transfer Semantics for Value Types
Richard Harris

Continuous Integration with CruiseControl.Net
Paul Grenyer

The Policy Bridge Design Pattern
Matthieu Gilson

Working with GNU Export Maps
Ian Wakeling

OVERLOAD 79

June 2007

ISSN 1354-3172

Editor

Alan Griffiths
overload@accu.org

Contributing editor

Paul Johnson
paul@all-the-johnsons.co.uk

Advisors

Phil Bass
phil@stoneymanor.demon.co.uk

Richard Blundell
richard.blundell@gmail.com

Alistair McDonald
alistair@inrevo.com

Anthony Williams
anthony.ajw@gmail.com

Simon Sebright
simon.sebright@ubs.com

Paul Thomas
pthomas@spongelava.com

Ric Parkin
ric.parkin@ntlworld.com

Roger Orr
rogero@howzatt.demon.co.uk

Simon Farnsworth
simon@farnz.co.uk

Advertising enquiries

ads@accu.org

Cover art and design

Pete Goodliffe
pete@cthree.org

Copy deadlines

All articles intended for publication in Overload 80 should be submitted to the editor by 1st July 2007 and for Overload 81 by 1st September 2007.

ACCU

ACCU is an organisation of programmers who care about professionalism in programming. That is, we care about writing good code, and about writing it in a good way. We are dedicated to raising the standard of programming.

The articles in this magazine have all been written by ACCU members - by programmers, for programmers - and have been contributed free of charge.

Overload is a publication of ACCU For details of ACCU, our publications and activities, visit the ACCU website: www.accu.org

4 The Policy Bridge Design Pattern
Matthieu Gilson offers some thoughts on implementing policy bridge in C++.

12 Live and Learn with Retrospectives
Rachel Davies presents a powerful technique to help with learning from experience.

15 Continuous Integration with CruiseControl.Net
Paul Grenyer asks: ‘Is CC any good? Should we all give it a go?’

20 Working with GNU Export Maps
Ian Wakeling takes control of the symbols exported from shared libraries built with the GNU toolchain.

24 auto_value: Transfer Semantics for Value Types
Richard Harris looks at std::auto_ptr and explores its reputation for causing problems.

Copyrights and Trade Marks

Some articles and other contributions use terms that are either registered trade marks or claimed as such. The use of such terms is not intended to support nor disparage any trade mark claim. On request we will withdraw all references to a specific trade mark and its owner.

By default, the copyright of all material published by ACCU is the exclusive property of the author. By submitting material to ACCU for publication, an author is, by default, assumed to have granted ACCU the right to publish and republish that material in any medium as they see fit. An author of an article or column (not a letter or a review of software or a book) may explicitly offer single (first serial) publication rights and thereby retain all other rights.

Except for licences granted to 1) Corporate Members to copy solely for internal distribution 2) members to copy source code for use on their own computers, no material can be copied from Overload without written permission from the copyright holder.

Rip It Up and Start Again

In my early days as a programmer, I often found myself responsible for maintaining poorly written, buggy code with no clear references as to the intended behaviour (or design). Much of it I didn't even write myself. A common reaction to such circumstances is a desire to re-write the code in a much more understandable and comprehensible form. I even tried that a couple of times.

lProgrammers like to rewrite systems

In most projects code "rots" over the course of time. It is natural for a programmer to make the simplest change that delivers the desired (results rather than the change that results in the simplest code). This is especially true when developers do not have a deep understanding of the codebase – and in any non-trivial project there are always some developers that don't understand part (or all) of the codebase. The result of this is that when functionality is added to a program or a bug corrected then there is more likelihood that the change will result in less elegant, more convoluted code that is hard to follow than the converse.

As a result of my early attempts at rewrite, I discovered that there are problems in rewriting software one doesn't understand: the resulting code often doesn't meet the most fundamental requirement – to do all the useful stuff that the previous version did. Even with an existing implementation the task of collecting system requirements isn't trivial – and getting the exercise taken seriously is a lot harder than for a "green field" project. After all, for the users (or product managers, or other domain experts), the exercise has already been done for the old system – and all the functionality is there to be copied.

Of course, for a developer writing new code of one's own is less frustrating than trying to work out what is happening in some musty piece of code that has been hacked by a parade of developers with varying thoughts on style and levels of skill. So developers rarely need convincing that they could do a better job of writing the system than the muppets that went before. (Even on those occasions that the developers proposing a rewrite are those the muppets responsible for the original.)

That doesn't stop them trying.

Managers don't like to rewrite systems

Managers are typically not sympathetic to the desire to rewrite a piece of existing software. A lot of work has been invested in bringing a system to its current state – and to deliver the next enhancement is the priority. With software-for-use the users will be expecting something from their "wishlist" with software-for-sale then a new version with more items on the marketing "ticklist" is needed. In both cases, even if the codebase is hard to work with, the cost of doing that is usually trivial compared to the cost of writing all the existing functionality again together with the new functionality.

A typical reason for rewriting to be considered is that the old system is too hard to change effectively – often because the relationship between the code and the behaviour is hard to understand. So it ought not to be a surprise that the behaviour is not understood well enough to replicate. The consequence of this is that, all too often, by the time the programmers realise that they can't deal effectively with making changes to the codebase a point has been reached where they are incapable of effectively rewriting it.

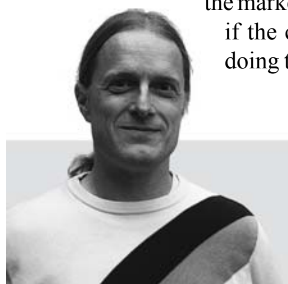
In view of this it is understandable that management prefer to struggle on with a problematic codebase instead of writing a new one. A rewrite is a lot of work and there is not guarantee that the result of the rewrite will be any better than the current situation.

And they know from experience that programmers cannot estimate the cost of new work accurately.

When is a rewrite necessary?

There are, naturally, occasions where a rewrite is appropriate. Sometimes it is necessitated by a change of technology – one system I worked on rewriting was a Windows replacement for the previous DOS version. This example worked out fairly well – the resulting codebase is still in used over a decade later. Another project replaced a collection of Excel spreadsheets with programs written in C++, at least that was the intent – years later there are still Excel spreadsheets being used. (The project did meet some of its goals – the greater efficiency of the C++ components supported a massive increase in throughput.)

I remember one system I worked on where there was one module that was evil. Only five hundred hundred lines of assembler – but touching it made brains hurt (twenty five years later I still remember the one comment that existed in this code "account requires" - which, as the code dealt with stock levels, was a non-sequeter). Not only was the code a mess, it was tightly coupled to most of the rest of the modules: every change caused a cascade of problems throughout the system. After a couple of releases where things went badly following a change to this module - weeks of delay whilst unforeseen interactions where chased down and addressed - I decided that it would be cheaper to rewrite this module than to make the next change to the existing code. I documented the inputs, the outputs, the improved internal design and the work necessary and asked permission. As luck



Alan Griffiths is an independent software developer who has been using "Agile Methods" since before they were called "Agile", has been using C++ since before there was a standard, has been using Java since before it went server-side and is still interested in learning new stuff. His homepage is http://www.octopull.demon.co.uk and he can be contacted at overload@accu.org

would have it the piece of code in question had sprung from the brain of the current project manager. He didn't believe things were as bad as I said. We tried conclusion: he would make the next change (whatever it was), and if it took longer to get working than the time (three weeks) I had estimated for a rewrite then I got to do the rewrite. The next change was a simple one – he had a version ready for testing in a couple of hours. He nearly got it working after two months and then conceded the question. I got to do the rewrite, which worked after my estimated three weeks and was shorter and measurably more maintainable (even by the boss).

But that was one of the good times – on most occasions a rewrite doesn't solve the problem. There are lots of reasons why maintaining a system may be problematic – but most of them will also be a problem while rewriting it. A poorly understood codebase is a symptom of other issues: often the developers have been under pressure to cut corners for a long time (and that same pressure will affect the rewrite), on other occasions they are new to the project and, in addition to lacking an understanding of the codebase they don't have a grasp of the system they propose to reproduce.

So, the caution is, to be sure that the underlying problem (the real cause of the problems with the codebase) will be addressed by the rewrite. An organisation (it is usually the organisation and not individuals) that produced messy code in the original system is prone to produce messy code in the rewrite.

The worst of all possible worlds

There is one thing worse than rewriting a system without a clear understanding of the causes of problems in the code. And it is seen far more often than common sense would suggest. This is to split up people with knowledge of the system and put some of them to rewriting the system and some of them to updating the existing one. The result is that both groups come under the sorts of pressure that cause code rot, and that the rewrite is always chasing a moving target. It is hard to maintain morale in both groups – both “working on the old stuff” and “chasing a moving target” demotivating. This is probably the most expensive of all solutions – it is more-or-less guaranteed that a significant amount of effort and skill will be wasted. And quality will suffer.

A curious change of roles

Recently I got was recruited to develop the fourth version of a system in three years. This rewrite was largely motivated by the developers – who felt they could offer a much more functional system by starting again. We never found out the truth of this conclusion as, several months into the project and following a change of line management, the rewrite was cancelled and our effort was diverted into other activities (such as getting the existing system deployed throughout the organisation).

As part of the re-evaluation that followed I had occasion to take a look at the the existing codebase and, after talking to the users of the system, it was clear that the most important of the desired new functions were easy to add. The principle problems with the existing code was a lack of pecifications and/or tests (and the little user documentation that did exist was wildly inaccurate). Addressing these issues took some time and has involved a few missteps, but after a few months and a couple of release we have a reasonably comprehensive set of automated test for both the legacy functionality and for the new functionality we've been adding. It may take another release cycle but we will soon be in the position to know that if the build succeeds then we are able to deploy the results. (And, since the build and test cycle is automated, we know whether the build succeeds half an hour after each check-in.)

So we have discarded the work on the fourth version and are now developing the third and have addressed the most serious difficulties in maintaining it. As I see it our biggest problem now is the amount of work needed to migrate a change from development into production – we need to co-ordinate the efforts of several support teams in each of a number of locations around the world. It can take weeks to get a release out (to the extent that code is now being worked on that will not be in the next release to production, nor the one that will follow, but the one after). In these circumstances it seems strange that a rewrite of the software is now being mooted. The reason? Because it is written in C++ and, in the opinion of the current manager, Java is a more suitable language.

This situation is a novelty for me: none of the developers currently thinks a rewrite is a good idea (neither those that worked on the rewrite, nor those that prefer Java) and, rather than the developers, it is the manager suggesting a rewrite. There may, of course, be good reasons for a rewrite – but the developers have a bug-free, maintainable system to work with and are not keen on replacing it with an unknown quantity.

The Policy Bridge Design Pattern

Matthieu Gilson offers some thoughts on using a policy based mechanism to build efficient classes from loosely coupled components.

Abstract

The problem considered in this paper is not new and relates to generic programming: how to build a class made of several 'components' that can have various implementations where these components can communicate with each other, while keeping the design of the components independent? For instance, an object would consist of an update mechanism plus a data structure, where both belong to a family of various mechanisms and data structures (distinct mechanisms perform distinct processes, etc.; thus some couples of mechanism-data are compatible and others are not).

To be more precise, such components would be implemented in classes (or similar entities) and use methods from other 'classes', while their design is kept orthogonal. The introduction illustrates this problem and then presents a solution using member function templates.

An alternative solution (so-called Policy Bridge) is then presented, where the 'components' are implemented in policies [Alexandrescu01], [Vandevoorde/Josuttis03] and the pattern 'bridges' them to construct an instantiated type. The term 'bridge' here refers to the usual `Bridge` pattern: the instantiated type can be seen as an interface that inherits the functionality of all the policies it is made of. The compatibility between the policies (for instance checking the interface at a call of a function from another policy) is checked at compile time. A common interface can be implemented with the cost of a virtual function for each call.

So far, the functionality achieved by the policy bridge is similar to the ones using member function templates and a brief comparison with meta-functions is explored as well. The code may be heavier but in a sense the design is more modular with policies. A combination of policies with meta-functions, to bring more genericity to encode the components, is then introduced. Policy-based visitors that enable visiting of some of the policies involved in the construction of an object via its interface.

A generic version of the policy bridge is then presented and the automation of such a code for an arbitrary number of policies (using macros from the preprocessor library of Boost [Boost]).

The example code runs under g++ (3.4.4 and 4.02), Visual C++ (7.1 and 8), and Comeau.

Introduction

The "why" and the "why not"

Let us play with neurons. Such objects can be abstracted mathematically, with various features and mechanisms to determine their activity: activation mechanism (their internal update method); synapses to connect them; etc. In order to simulate a network made of various types of neurons interconnected with various types of connections, and keep the design as flexible as possible to be able to combine the different mechanisms together at will.

Without even going as far as connecting neurons, a fair number of design issues arise. Say we have a data structure which represents a feature of one type of neuron:

```
class Data1
{
public:
  Data1(double a) : a_(a) {}
  const double & a() {return a_;}
protected:
  const double a_;
};
```

and an update mechanism to model the state of our neuron where the update method uses the parameter a implemented in the data:

```
class UpdateMech1
{
public:
  const double & state() const {return state_;}
  UpdateMech1() : state_(0.) {}
  void update(double a) {state_ += a;}
protected:
  double state_;
};
```

Our neuron is somehow the 'merging' of both classes, and another update method at the level of `Neuron1` has to be declared so as to link the parameter of `UpdateMech1::update` with `Data1::a()`:

```
class Neuron1
  : public Data1
  , public UpdateMech1
{
public:
  Neuron1(double a) : Data1(a) {}
  void update() {UpdateMech1::update(Data1::a());}
};
```

To make it more generic for various data and mechanisms (say we have `Data1` and `Data2`, `UpdateMech1` and `UpdateMech2`), we can templatise the features and mechanisms:

```
template <class Data, class UpdateMech>
class Neuron
  : public Data
  , public UpdateMech
{
  ...
  void update() {UpdateMech::update(Data::a());}
};
```

Matthieu Gilson Matthieu is a PhD student in neuroscience at the University of Melbourne and the Bionic Ear Institute, mainly playing with mathematics but also computer simulation. He started to develop in C++ a couple of years ago and strangely finds design patterns somehow beautiful (almost as maths). He still remains sane thanks to music, sport, cooking and beer brewing.

the template version does not require virtual functions, and thus the internal mechanisms remain fast

Yet, all of the **DataX** would now require a method **a** and all the **UpdateMechX** would require an update method with a double as parameter, which is not flexible. The alternative using abstract base classes for both the family of the data classes (**DataBase**) and the family of the update mechanisms (**UpdateMechBase**), while **Neuron** would hold one member variable for each family, would face similar limitations:

```
class Neuron
{
  ...
  void update() {u_.update(d_.a());}
  DataBase * d_;
  UpdateMechBase * u_;
};
```

Indeed, the interface **DataBase** should have a pure virtual method **a**:

```
struct DataBase
{
  virtual const double & a() = 0;
};
```

and this would cause the interface to be rigid (even more than with the template version): it could be necessary for other update mechanisms to get a non-constant **a**, which does not fit this method **a**; if some more variables are needed by only a particular **UpdateMechX**, the whole common interface still has to be modified; etc. Actually, the family of the data classes needs no common interface here, provided that for each combination with a specific **UpdateMechX**, the latter knows how to call the variable it needs. On the contrary, the machinery implemented in **DataX** and **UpdateMechX** and their way to communicate with each other should remain as unconstrained as possible (in terms of constructors, etc.), only **Neuron** would need a common interface in the end.

Speed could be another reason to discard this alternative choice, since the template version does not require virtual functions, and thus the internal mechanisms remain fast (only function calls).

The former version of **Neuron** can be modified to make the combinations between the **DataX** and the **UpdateMechX** more flexible. An option is to change **UpdateMechX** into class templates where **DataX** is the template argument of **UpdateMechX**, and to derive them from the classes **DataX**, as shown in Listing 1.

```
class Data1
{
  ...
  const & double a();
};

template <class Data>
class UpdateMech1
  : public Data
{
  ...
  void update() {state_ += Data::a();}
};

template <class UpdateMech>
class Neuron
  : public UpdateMech
{ ... };

Neuron<UpdateMech1<Data1> > nrn1;
nrn1.update();
```

Listing 1

This way, we can define various update mechanisms and various data classes, and combine them together. The **Data** family and the **UpdateMech** family are kept somehow independent in terms of design: basically, **UpdateMech1** only requires to know the name of the method **a** implemented in all of the **DataX** to be combined with **UpdateMech1**; type checks require that **a** returns a type convertible into **const double**.

Interdependence between features and mechanisms

The construction **Neuron<UpdateMech<Data> >** still has quite strong limitations, in the sense that **Data** and **UpdateMech** are no longer on the same level in the derivation tree above **Neuron**. Imagine that one of the data classes also needs to use a method that belongs to the family of the **UpdateMechX**. Or imagine there are not just one update mechanism but two (e.g. a learning mechanism **PlasticityMech** in addition to the **UpdateMech** for some specific neurons), and they both would have to use something from the **Data** class family; both would derive from **Data**, and **Neuron** would derive from both, so the derivation from **Data** must be virtual, as shown in Listing 2.

Since there was no virtual machinery before, this may imply costs. But even if we would not care, there is a last reason to seek for another solution: there may eventually be a need to implement mechanisms with 'new'

```
class Data1;

template <class Data>
class UpdateMech1
  : virtual public Data
{ ... };

template <class Data>
class PlasticityMech1
  : virtual public Data
{ ... };

template <class UpdateMech, class PlasticityMech>
class Neuron
  : public UpdateMech
  , public PlasticityMech
{ ... };
```

Listing 2

any further change would possibly mess up our beautiful (but rigid) derivation tree

```
class UpdateMech1
{
  ...
protected:
  template <class TypeImpl>
  void update_impl(TypeImpl & obj) {state_ +=
obj.a_impl();}
};

class Data1
{
  friend class UpdateMech1;
protected:
  const double & a_impl();
  ...
};
```

Listing 3

interactions that we cannot really predict at this stage; and any further change would possibly mess up our beautiful (but rigid) derivation tree.

Let us go back to thinking of our design and what we wanted of it. The key idea here is that **UpdateMech1** requires that **Neuron** has a method **a** that is used by **UpdateMech1**, but the fact that **Neuron** inherits **a** from **Data1** does not matter to **UpdateMech1**. Other mechanisms may have ‘requirements’, which makes mechanisms compatible or not (to be detected at compile time). They would need to be all at the ‘same level’ in the derivation tree, i.e. any mechanism or feature is equally likely to need something in another one. In other words, we want a design likely to combine any compatible update and data ‘classes’ in a flexible way and thus define a neuron out of them; these mechanisms can communicate together while they are kept as ‘orthogonal’ as possible (they only know the name of the methods belonging to other mechanisms they need to call); and as few virtual functions as possible should be involved in the internal machinery (within **Neuron**), to try to keep fast computation.

A solution with member function templates

One solution can be to use member function templates (in the classes that define the features or mechanisms) for the methods that require a feature from outside their scope (Listing 3) and to call **update_impl** from the class **Neuron** with ***this** as an argument (Listing 4).

Note that **UpdateMech1** has to be a friend of **Data1** if **a_impl** is protected. We would need to add similar declarations for any other mechanism to be combined with **Data1**, or define all the methods as **public** in the features and mechanisms. Another slight drawback is that the calls to the member function templates must come from **Neuron**, which means that having a pointer to one of the mechanisms (such as **UpdateMech**) is not sufficient to call the update method if it is a template method (we still would need to pass the **Neuron** object as an argument).

In the end, this solution is valid and would be suitable for our requirements. Yet, we will now consider an alternative design, which brings further

```
template <class Data, class UpdateMech>
class Neuron
  : public Data
  , public UpdateMech
{
  ...
  void update() {UpdateMech::update_impl(*this)}
};
```

Listing 4

possibilities. The trade-off is mainly the increase in complexity in the design.

Details of the design

Another implementation of the ‘direct’ call: making use of policies

The purpose is still to allow any mechanism to call ‘directly’ any method from another ‘mechanism’ from which **Neuron** derives (via the class **Neuron**). To give a rough idea, instead of the function template nested in the mechanism class (**UpdateMech1::update_impl<class TypeImpl>(TypeImpl &)**), the whole class **UpdateMech1** is now a class template with a non template function (**UpdateMech1<class TypeImpl>::update_impl()**). This idea has something to do with the well-known trick used together with curiously recurring pattern (CRTP [Alexandrescu01], [Vandevoorde/Josuttis03]):

```
template <class TypeImpl>
class UpdateMech
{
  void update_impl()
  {
    state_ += static_cast<TypeImpl &>(*this).a();
  }
};
```

Defined this way, **UpdateMech** can be seen as a policy [Alexandrescu01], [Vandevoorde/Josuttis03] on the implemented type **TypeImpl** with the template argument **Neuron**. It allows us to call the method **a** that **Neuron** inherits from **Data** by means of **static_cast**, which converts the self-reference ***this** to the instantiated policy object back to **Neuron**.

This use of policies is somewhat different from [Alexandrescu01]: a usual example is the pointer class **SmartPtr** defined on a type **T** (to make it shorter than **TypeImpl**) and some properties of the pointers can be implemented by policies. For instance, let us build a policy to count the references of the pointed element (Listing 5) and the smart pointer class template is then as shown in Listing 6.

The policy **CountRef** here is a mechanism that uses the type **T**, and **SmartPtr** uses **T** and **CountRef**, so there is no recursion in the design. However, we want to build **Neuron** which uses **DataPolicy<Neuron>** and **UpdatePolicy<Neuron>**.

```
template <class T>
class CountRef
{
private:
  int * count_;
protected:
  void CountPolicy() {count_ = new int();
     *count_ = 1;}
  bool release()
  {
    if (!--*count_) {
      delete count_; count_ = NULL; return true;
    } else return false;
  }
  T clone(T & t) {++*count_; return t;}
  ...
};
```

Listing 5

```
template <class T
  , template <class> CountPolicy>
class SmartPtr
  : public CountPolicy<T>
{
public:
  SmartPtr(T * ptr) : CountPolicy<T>(),
     ptr_(ptr) {}
  ~SmartPtr()
  {
    if (CountPolicy<T>::release())
      delete ptr_;
  }
  ...
protected:
  T * ptr_;
};
```

Listing 6

Merging the protected 'spaces' of several policies

First, we rewrite the mechanisms **Data1** and **UpdateMech1** as policies (see Listing 7).

```
template <class TypeImpl>
class Data1
{
protected:
  typedef double VarTypeInit;
  Data1(VarTypeInit a) : a_(a) {}
  const double & a_impl() {return a_;}
private:
  const double a_;
};

template <class TypeImpl>
class UpdateMech1
{
public:
  const double & state() const {return state_;}
protected:
  UpdateMech1() : state_(0.) {}
  void update_impl()
  {
    state_ +=
       static_cast<TypeImpl &>(*this).a_impl();
  }
private:
  double state_;
};
```

Listing 7

Note that the **typdedef VarTypeInit** was added in **Data1** to define the type of the parameter required to initialise this policy.

Then the **Neuron** pattern straightforwardly combines these two policies, i.e. it derives from the two policies applied on itself (see Listing 8).

The requirements for the policy any **DataPolicyX** is to define a type **VarTypeInit** to be used in its constructor and have a method **a_impl** that returns a type that can be converted to **const double &**. Likewise for **UpdatePolicyX** which must have a method **update_impl** (no return type is required).

```
template <template <class> class DataPolicy
  , template <class> class UpdatePolicy>
class  Neuron
  : public DataPolicy<Neuron<DataPolicy,
     UpdatePolicy> >
  , public UpdatePolicy<Neuron<DataPolicy,
     UpdatePolicy> >
{
  friend class DataPolicy<Neuron>;
  friend class UpdatePolicy<Neuron>;

public:
  Neuron(DataPolicy::VarTypeInit a)
    : DataPolicy<Neuron>(a)
    , UpdatePolicy<Neuron>()
  {}
  const double & a() {
     return DataPolicy<Neuron>::a_impl();}
  void update()
     {UpdatePolicy<Neuron>::update_impl();}
};
```

Listing 8

Now, we can simply define **Neuron1** as a combination of **Data1** and **UpdateMech1**:

```
typedef Neuron<Data1, UpdateMech1> Neuron1;
```

The policies can communicate together, via the casting back to **TypeImpl**. Indeed, thanks to the friend declarations in **Neuron**, each policy can access to the members of **Neuron**, which inherits the protected members of all the policies. Note also that the public function members of the policies and of **Neuron** will be public for **TypeImpl**. What is private within a policy cannot be accessed by other policies, to keep safe variables (e.g. via protected accessors). An alternative option is to use protected derivation instead of the public one here in the design of **Neuron**, in order to prevent public members of the policies from being accessible from **TypeImpl**.

Compatibility between classes

Now, we define two new policies **Data2** and **UpdateMech2** (respectively in the same family as **Data1** and **UpdateMech1**), but the data **a_** is no longer constant in **Data2**, and the update method of **UpdateMech2** changes its value. Furthermore, **Data2** has a **learning** method which modifies its variable **a_** according to the variable **state_** (see Listing 9).

```
template <class TypeImpl>
class Data2
{
protected:
  typedef double VarTypeInit;
  Data2(VarTypeInit a) : a_(a) {}
  double & a_impl() {return a_;}
  void learning()
  {
    if (static_cast<TypeImpl &>
        (*this).state_impl()>10) --a_;
  }
private:
  double a_;
};

template <class TypeImpl>
class UpdateMech2
{
public:
  const double & state() const {return state_;}
protected:
  UpdateMech2() : state_(0.) {}
  void update_impl()
  {
    state_ +=
       ++static_cast<TypeImpl &>(*this).a_impl();
  }
private:
  double state_;
};
```

Listing 9

These two policies are 'compatible' in the sense that the update policy modifies the data **a_**, which is not constant as a return type of **a_impl** in **Data2** and the type **Neuron2** defined by:

```
typedef Neuron<Data2, UpdateMech2> Neuron2;
Neuron2 nrn2;
nrn2.update();
```

is thus 'legal'. So is the combination between **Data2** and **UpdateMech1**.

However, the class **Neuron3** defined here:

```
typedef Neuron<Data1, UpdateMech2> Neuron3;
Neuron3 nrn3;
nrn3.update();
```

is ill-defined because the implemented **update_impl** method of **UpdateMech2** tries to modify the data **a_** of **Data1** which is constant, and the method cannot be called on an object of this type. Such an incompatibility is detected at compile time (the member function templates described achieve same functionalities). Depending on the compiler, the incompatibility is detected when defining the type **Neuron<Data1, UpdateMech2>** (g++) or when calling the method **update** (VC8).

Managing the bridged objects via dedicated interfaces

The term bridged objects here refers to instantiated types using a policy bridge. If we want to design an interface for all the neurons to access **a_impl** (with **const double &** as common return type) and **update_impl**, we can add another derivation to **Neuron** with pure virtual methods. For example for **update_impl**:

```
struct InterfaceBase
{
  virtual void update() = 0;
};
```

which provides the common method update that has to be defined in a derived class. We can either do it in **Neuron** (or in a derived class if needed). Defining it in **Neuron** saves some code:

```
template <...> class Neuron : public InterfaceBase,
   ...;
void Neuron<...>::update() {
  UpdatePolicy<Neuron>::update_impl();}
```

but then the order of the policies in the list of the template arguments is then fixed. On the contrary, in a dedicated derivation (instead of a **typedef** like **Neuron1** above), the order of the template argument list can be changed at will:

```
class Neuron4
   : public Neuron<Data1, UpdateMech1>
   , public InterfaceBase;
```

We can no longer use the policy bridge to build neurons with distinct algorithms

and we can also imagine combining more than two policies to define neurons. The latter option would keep the design more generic (see the generic version `PolicyBridgeN` of `Neuron`), with no member function but the constructor. Yet, the same code would be duplicated in all the derived classes. Note that such an interface also allows us to store neurons in standard containers.

Now, to implement a special interface for the 'learning' neurons, we just have to derive the suitable policy (only `Data2` here, but we can imagine more) from another abstract base class with a pure virtual method called `learning`:

```
struct Interface1
{
  Interface1() {list_learning_nrn.push_back(this);}
  virtual void learning() = 0;
  std::list<Interface1 * const>
     list_learning_neurons;
};
template <class TypeImpl>
   class Data2 : public Interface1 {...};
```

Thus, learning neurons can be handled and updated specifically (for their learning mechanisms only) separately from the rest of the neurons (useful when the distinct processes happen at distinct times for example):

```
 for(std::list<Interface1 * const>::iterator i =
       Interface1::list_learning_neurons.begin();
     i != Interface1::list_learning_neurons.end();
     ++i)
   (*i)->learning();
```

This polymorphic feature only uses the derivation from an abstract base class and the cost is the one of a virtual function call at run time. Note that any bridged class (whatever the bridge like `Neuron`) can actually share the same interface, and thus can be handled together as shown here. An alternative solution could be to use the variant pattern and suitable visitors [Boost], [Tiny].

Further considerations

Comparison with the use of member function templates

So far, apart from the dedicated interfaces, this design based on policies has functionality comparable with member function templates. Another difference is that the body of `Neuron` does not have to be changed for a new mechanism with a method that requires internal communication (like when adding `Data2::learning`). We indeed do not need to change `Neuron` with policy, whereas we would have to define in the body of `Neuron` a call for `Data2::learning<TypeImpl>` when using member function templates.

Note that the computation speed is equivalent for the two options, since all the internal mechanisms are based on function calls (no virtual function, only the use of an interface involves a virtual function call).

More genericity with the combination of policies and meta-functions

Further refinement can be obtained using meta-functions (that implement polymorphism at compile-time). So far, we only considered policies with one sole template argument, which is eventually the type of the bridged object (`TypeImpl`). Say we want to parametrise some of the policies with more template parameters. For example, the `learning` in data class `Data2` would depend on an algorithm embedded in a class, thus we need to add an extra template parameter for `Data2` (and not `Data1`):

```
template<class TypeImpl, class Algo> class Data2;
```

We can no longer use the policy bridge to build neurons with distinct algorithms because the template signature of `Data2` is different now (one template parameter in the original design of the policy bridge `Neuron`). Meta-functions can help here:

```
 template<class Algo>
 struct GetData2
 {
   template <class TypeImpl>
   struct apply
   {
     typedef Data1<TypeImpl, Algo> Type;
   };
};
```

With this design, `TypeImpl` will be provided by the bridge, and all the other template parameters can be set via `GetData2` (with suitable

```
template <class GetData, class GetUpdateMech>
class Neuron
   : public GetData::template
       apply<Neuron<GetData, GetUpdateMech>
>::Type
  , public GetUpdateMech::template
      apply<Neuron<GetData, GetUpdateMech> >::Type
{
  typedef typename GetData::
     template apply<Neuron>::Type InstData;
  typedef typename GetUpdateMech
     ::template apply<Neuron>::Type
    InstUpdateMech ;

  friend InstData;
  friend InstUpdateMech;

protected:
  Neuron(InstData::VarTypeInit
        var_init)
    : InstData(var_init)
    , InstUpdateMech()
  {}
};
```

Listing 10

defaulting if needed). The policy bridge then becomes as shown in Listing 10.

Note that the instantiated policy **GetData::template apply<Neuron>::Type** was renamed into **InstData** using **typedef** (likewise for **InstUpdateMech**) in order to simplify the code. A neuron type can thus be simply created:

```
typedef Neuron<GetData2<Algo1>,
   GetUpdateMech2> Neuron5;
```

Policy-based visitors

Non-template classes with member function templates can be used with usual visitors because they have a plain type. We adapt here the processing using basic visitors [Alexandrescu01] (with an abstract base class **VisitorBase**) to be able to visit the policies which a bridged object is made of:

```
class VisitorBase
{
  virtual ~VisitorBase() {}
};
template <class T>
class Visitor
{
  virtual void visit(T &) = 0;
};
```

In particular, such a visitor must have the same behaviour for all the types **T = Policy1<...>**, for a given **Policy1**. We thus need a tag to identify each policy, which is common to all the instantiated types related to the same policy, and a solution is to use **Policy<void>** because it will never be used in any bridged object. A particular visitor for **Data1** and **Data2** would be implemented:

```
class DataVisitor
  : public VisitorBase
  , public Visitor<Data1<void> >
  , public Visitor<Data2<void, void> >
{
  void visit(Data1<void> & d) {...}
  void visit(Data2<void, void> & d) {...}
};
```

```
template <class TypeImpl> class Data1;

template <>
class Data1<void>
{
  friend class DataVisitor;
protected:
  Data1(double a) : a_(a) {}
  const double & a_impl() const {return a_;}
  void apply_vis_impl(VisitorBase & vis)
  {
    Visitor<Data1<void> > * pv =
      dynamic_cast<Visitor<Data1<void> > *>(&
vis);
    if (pv) pv->visit(*this);
  }
private:
  const double a_;
};

template <class TypeImpl>
class Data1
  : public Data1<void>
{
protected:
  typedef double TypeInit;
  Data1(TypeInit a = 0.) : Data1<void>(a) {}
};
```

Listing 11

Now, a method **apply_vis(VisitorBase & vis)** has to be defined at the interface level (pure virtual method), and defined at the same level as **update** to call the method in the policy (here in **Neuron1**, likewise for **Neuron2**):

```
void Neuron1::apply_vis(VisitorBase & vis)
{
  InstData::apply_impl(vis);
}
```

Finally, the 'visitable part' of the policy **Data1** (the data that the visitor wants to access, **a_** here) has to be moved in the specialised instantiation **Data1<void>** from which derive all the **Data1<TypeImpl>**. The function **apply_impl** has also to be defined in its body (here with a solution using **dynamic_cast** to check the compatibility of the visitor), as shown in Listing 11.

This way, the argument of the call of **visit** is the parent object of type **Data1<void>** instead of the object itself. The cost of the passage of the visitor is thus a **dynamic_cast** plus a virtual function call. In the case the visitor is not compatible, nothing is done here, but an error could be thrown or returned by the method **apply_impl**. Likewise with **Data2<void, void>** for the visitable part of **Data2**.

Policy Bridge design for an arbitrary number of policies

Recap of the code

In the end, the code can be abstracted to bridge N policies (the dedicated methods such as **update**, etc. are 'decentralised' in derived classes), in Listing 12 with N = 15.

This design requires each policy to define a type **TypeInit**, which is set to a 'fake void' type **NullType** when no initialisation variable is required (same **NullType** as used for **TypeList** in [Alexandrescu01]). The overriding of the pure virtual functions defined in the interface are left to the implementation of the instantiated types here. A class **Neuron1** is derived from a suitable instantiation as shown in Listing 13.

A dedicated version of **PolicyBridge2** could be written to create neurons with the common interface hard-wired: **InterfaceBase** should be put in the template argument list and the methods **update**, **a** and **apply** can be centralised in the **PolicyBridge2bis** code (to save some code lines), shown in Listing 14.

Automation of the code generation

The code of **PolicyBridgeN** can be generated for all values of N in a define range using preprocessor macros such as **REPEAT** (cf. **boost::preprocessor** [Boost], the TTL library [Tiny]), i.e. to create the class templates **PolicyBridge1** up to **PolicyBridgeN**, where **N** is an arbitrary limit.

See http://www.bionicear.org/people/gilsonm/prog.html for details on this implementation.

Conclusion

The policy bridge pattern aims to build classes that inherit 'properties' or functionalities from a certain number of policies (variable and function members from the protected 'space'). Each policy can call members from other ones and the compatibility between the policies is checked at compile time. This approach is modular and flexible, and keeps the design of policies related to distinct functionalities somehow 'orthogonal'.

A common interface can be designed in order to provide a main 'gate' (to all the common functionalities that have to be public, and the cost is the one of a virtual-function call), to store them in standard containers, etc. Moreover, specific interfaces can similarly be design for particular functionalities belonging to a restrained number of mechanisms, allowing us to pilot the bridged objects according to what they are actually made of. Usual visitors can be adapted and applied to these objects in order to interact with some of the policies involved in the design. Meta-functions

```
template <class GetPolicy0
  , ...
  , class GetPolicy14>
class PolicyBridge15
   : public GetPolicy0::template
       apply<PolicyBridge15<GetPolicy0, ...,
          GetPolicy14> >::Type
  , ...
  , public GetPolicy14::template
       apply<PolicyBridge15<GetPolicy0, ...,
          GetPolicy14> >::Type
{
protected:
  typedef GetPolicy0::template
apply<PolicyBridge15>::Type
    InstPolicy0;
  ...
  typedef GetPolicy14::
     template apply<PolicyBridge15>::
     Type InstPolicy14;

  friend InstPolicy0;
  ...
  friend InstPolicy14;

  typedef typename InstPolicy0::TypeInit
TypeInit0;
  ...
  typedef typename InstPolicy14::
     TypeInit TypeInit14;

  PolicyBridge15(TypeInit0 var_init0
      , ...
      , TypeInit14 var_init14)
    : InstPolicy0(var_init0)
    , ...
    , InstPolicy14(var_init14)
  {}
};
```

Listing 12

```
class Neuron1
  : public PolicyBridge2<GetData1, GetUpdateMech1>
  , public InterfaceBase
{
public:
  Neuron1(TypeInit0 a)
    : PolicyBridge2<GetData1, GetUpdateMech1>(
       a, NullType()) {}
   void update()
  {
     GetUpdateMech1::apply<PolicyBridge2<GetData1,
        GetUpdateMech1>
      >::Type::update_impl();
  }
   const double & a() { ... }
   void & apply_vis(VisitorBase & vis) { ... }
};
```

Listing 13

```
template <...>
PolicyBridge2bis<InterfaceBase, GetData,
GetUpdateMech>
{
  void update()
  {
    GetUpdateMech::template <...>::
       Type::update_impl();
  }
  ...
};
```

Listing 14

can help and bring further refinement to use policies with more than one template parameter.

Such design proved to be useful and efficient in a simulation program where objects of various types interact: for instance neurons with distinct intern mechanisms, as well as diverse synapses to connect them; these objects are created at the beginning of the program execution and they evolve and interact altogether during the simulation. Requirements such as ‘fast’ object creation or deletion have not been considered here so far, ‘optimisation’ was sought for only for the function calls (call via the common interface class). Visitors were used for adequate object construction and linking objects from distinct kind (in particular between synapses and neurons to check compatibility when creating a synaptic connection), but not used during the execution: the interface then provides a faster way to access the functionalities of the objects.

Code is available online at http://www.bionicear.org/people/gilsonm/index.html to illustrate how such a design can be used. ■

Acknowledgements

The author thanks Nicolas di Cesare for earlier discussion about design pattern, and Alan Griffiths, Phil Bass and Roger Orr for very helpful comments that brought significant improvements of the text during the review process. MG is funded by a scholarship from the NICTA Victorian Research Lab (www.nicta.com.au).

References

[Alexandrescu01]A. Alexandrescu. *Modern C++ design: generic programming and design patterns applied*, Addison-Wesley, Boston, MA: 001. Includes bibliographical references and index.

[Vandevoorde/Josuttis03]David Vandevoorde, Nicolai M. Josuttis *C++ templates : the complete guide*, Addison Wesley, 2003.

[Boost]Boost c++ libraries. http://www.boost.org/

[Tiny]Tiny Template Library. http://tinytl.sourceforge.net/

Live and Learn with Retrospectives

How can a team learn from experience? Rachel Davies presents a powerful technique for this.

Software development is not a solitary pursuit it requires collaboration with other developers and other departments. Most organisations establish a software lifecycle that lays down how these interactions are supposed to happen. The reality in many teams is that their process does not fit their needs or is simply not followed consistently. It's easy to grumble when this happens and it can be frustrating if you have ideas for improvements but are not sure how to get them off the ground. This article offers a tool that may help your team get to grips with process improvement based on the day-to-day experience of the team. Retrospectives are a tool that a team can use for positive change to shift from following a process to driving their process.

Retrospectives are meetings that get the whole team involved in the review of past events and brainstorming ideas for working more effectively going forward. Actions for applying lessons learned are developed for the team by the team. This article aims to explain what you need to do to facilitate a retrospective for your team.

Background

The term 'Retrospective' was coined by Norman Kerth author of *Project Retrospectives: a handbook for team reviews* [Kerth]. His book describes how to facilitate three-day off-site meetings at the end of a project to mine lessons learned. Such retrospectives are a type of post-implementation review – sometimes called *post-mortems*! It seems a pity to wait until the end of a project to start uncovering lessons learned. In 2001, Extreme Programming teams adapted retrospectives to fit within an iterative development cycle [Collins/Miller01]. Retrospectives also got added to another agile process, Scrum [Schwaber04] and nowadays it's the norm for teams applying agile development to hold many short "heartbeat" retrospectives during the life of the project so that they can gather and apply lessons learned about their development process **during** the project rather than waiting until the end.

Learning from experience

Experience without reflection on that experience is just data. Taking a step back to reflect on our experience is how we learn and make changes in our daily lives. Take a simple example, if I hit heavy delays driving to work then it sets me thinking about alternative routes and even other means of getting to work. After some experimentation, I settle into a new routine.

No one wants to be doomed to repeating the same actions when they are not really working (some definition of madness here). Although a retrospective starts with looking back over events, the reason for doing this is to change the way we will act in the future – retrospectives are about creating change not navel-gazing. Sometimes we need to rethink our approach rather than trying to speed up our existing process.

Retrospectives also improve team communication. There's an old adage "a problem shared is a problem halved". Retelling our experiences to friends and colleagues is something we all do as part of everyday life. In a team effort, none of us know the full story of events. The whole story can only be understood by collating individual experiences. By exploring how the same events were perceived from different perspectives, the team can come to understand each other better and adjust to the needs of the people in their team.

Defusing the time-bomb

Let's move onto how to run an effective retrospective. Where a team has been under pressure or faced serious difficulties tempers may be running high and relationships on the team may have gone sour. Expecting magic to happen just by virtue of bringing the team together in a room to discuss recent events is unrealistic. Like any productive meeting, a retrospective needs a clear agenda and a facilitator to keep the meeting running smoothly. Without these in place, conversations are likely to be full of criticism and attributing blame. Simply getting people into a room to vent their frustrations is unlikely to resolve any problems and may even exacerbate them. Retrospectives use a specific structure designed to defuse disagreements and to shift the focus to learning from the experience. The basic technique is to slow the conversation down – to explore different perspectives before jumping to conclusions.

The prime directive

By reviewing past events without judging what happened, it becomes easier to move into asking what could we do better next time? The key is to adopt a systems thinking perspective. To help maintain the assumption that problems arise from forces created within the system rather than destructive individuals Norm Kerth declared a Prime Directive for retrospectives that he proposed is a fundamental ground-rule for all retrospectives.

This purpose of this prime directive is often misunderstood. Clearly, there are times when people messed up – maybe they don't know any better or maybe they really are lazy or bloody-minded. However, in a retrospective the focus is solely on process improvements and we use this Prime Directive to help us suspend belief. Poor performance by individuals is best dealt with managers or HR department and should be firmly set outside the scope of retrospectives.

Rachel Davies Rachel is an independent agile coach based in the UK, a frequent presenter at industry conferences and a director of the Agile Alliance. She has been working in the software industry for nearly 20 years. She can be reached via her website: www.agilexp.com

Prime Directive

Regardless of what we discover, we must understand and truly believe that everyone did the best job he or she could, given what was known at the time, his or her skills and abilities, the resources available, and the

> **if in the heat of the moment this rule is flouted then you can try use a 'talking stick' so only one person is talking at a time**

Getting started with ground rules

To run an effective retrospective someone needs to facilitate the meeting. It's the facilitator's job to create an atmosphere in which team members feel comfortable talking.

Setting ground-rules and a goal for the retrospective helps it to run smoothly. There are some obvious ground-rules that would apply to most productive meetings – for example, setting mobile phones to silent mode. So what special ground-rules would we need to add for a retrospective? It's important for everyone to be heard so an important ground-rule is 'No interruptions' – if in the heat of the moment this rule is flouted then you can try use a 'talking stick' so only one person is talking at a time – the person holding the talking stick token (the token does not have to be a stick – Norm uses a mug and teams I have worked with have used a fluffy toy which is easier to throw across a room than a mug).

Once the ground-rules for the meeting are established then they should be written up on flipchart paper and posted on the wall where everyone can see them. If people start to forget the ground-rules then it is the facilitator's job to remind everyone. For example, if someone answers a phone call in the meeting room then gently usher them out so that their conversation does not disrupt the retrospective.

Safety check

Another important ground rule is that participation in exercises during a retrospective is optional. Some people can feel uncomfortable or exposed in group discussions and it's important not to exacerbate this if you want them to contribute at all. When a team do their first few retrospectives, it's a useful to run a 'Safety Check' to get a sense of who feels comfortable talking. To do this run an anonymous ballot, ask each person to indicate how likely they are to talk in the retrospective by writing a number on slips of paper using a scale 1 to 5 (where 1 indicates 'No way' and 5 'No problem') – the facilitator collects these slips of paper in, tallies the votes and posts them on a flipchart in the room. The purpose of doing this is for the participants to recognise that there are different confidence levels in the room and for the facilitator to assess what format to use for discussions. Where confidence of individuals is low, it can be effective to ask people to work in small groups and to include more exercises where people can post written comments anonymously.

Action replay

Sportsmen use the action replay to analyse their actions and look for performance improvements. The equivalent in retrospectives is the Timeline.

Start by creating a space for the team to post events in sequence that happened during the period they are reflecting over; moving from left to right – from past to present. Each team member adds to the timeline using coloured sticky notes (or index cards). The facilitator establishes a key for the coloured cards. For example, pink – negative event, yellow – neutral event and green – positive event. The use of colour helps to show patterns in the series of events. This part of the meeting usually goes quickly as team members work in parallel to build a shared picture.

The exercise of creating a timeline serves several purposes – helping the team to remember what happened, providing an opportunity for everyone on the team to post items for discussion and trying to base conversations on actual events rather than general hunches. The timeline of event is a transient artefact that helps to remind the team what happened but it is not normally kept as an output of the retrospective.

Identifying lessons learned

Once a shared view of events has been built, the team can start delving for lessons-learned. The team is invited to walk the timeline from beginning to end with the purpose of identifying 'What worked well that we want to remember?' and 'What to do differently next time?'

The facilitator reads each note on the timeline and invites comments from the team. The team work to identify lessons learned both good and bad. It's important to remind the team at this stage that the idea is to identify areas to focus on rather than specific solutions as that comes in Action Planning.

As a facilitator, try to scribe a summary of the conversation on a flipchart (or other visible space) but try to check with the team that what you have written accurately represents the point being made. Writing down points as they are originally expressed helps show that a person's concerns have been listened to.

In my experience, developers are prone to talking at an abstract level – making general claims that are unsubstantiated. As a facilitator, it's important to dig deeper and check assumptions and inferences by asking for specific examples that support the claims being made.

Action planning

Typically, more issues are identified than can be acted on immediately. The team will need to prioritise issues raised before starting action planning. The team needs to be realistic rather than wishful thinking mode. For an end of iteration retrospective, between three and five actions would be sensible.

Before setting any new actions the team should review if there are outstanding actions from their previous retrospective. If so then it's worth exploring why and whether the action needs to be recast. Sometimes people are too ambitious in framing an action and need to decrease the scope to something they can realistically achieve. For each action, try to separate out the long-term goal from the next step (which may be a baby-step). The team may even decide to test the water by setting up a process improvement as an experiment where the team take on a new way of working and then review its effectiveness at the next retrospective. Also it's important to differentiate between short-term fixes and attempting to address the root cause. Teams may need both types of action – a book, which provides a nice model for differentiating between types of action, is Edward DeBono's *Six Action Shoes* [DeBono93].

The power of regular retrospectives and regular exercise is that they prevent big problems from happening

Each action needs an owner responsible for delivery plus it can be a good idea to identify a separate person to act as a buddy and work with that person to make sure the action gets completed before next retrospective. Some actions may be outside the direct sphere of influence of the team and require support from management – the team may need to sell the problem! Your first action in this case, is to gather evidence that will help the team convince their boss action is required.

Wrapping-up

Before closing the retrospective, the facilitator needs to be clear what will happen to the outputs of the meeting. The team can display the actions as wallpaper in the team's work area. Or the team may choose to use a digital camera to record notes from flipcharts/whiteboards so the photos can be upload a shared file space or transcribed onto a team wiki. Before making outputs visible to the wider organisation the facilitator should need to check with the team that they are comfortable with this.

Perfecting retrospectives

To run a retrospective it helps to hone your facilitation skills – a retrospective needs preparation and follow through. The facilitator should work through the timings in advance and vary the exercises every now and again. A good source of new exercises is the book *Agile Retrospectives* [Derby/Larsen06]. A rough guide to timings is a team need 30 minutes retrospective time per week under review so using this formula allow 2 hours for a monthly retrospective and a whole day for a retrospective of a several months work.

In addition, to planning the timings and format, the facilitator also needs to review: Who should come? Where to hold the meeting? When to hold the meeting? When a team first starts with retrospectives they will find that they come up with plenty of actions that are internal to the team. Once the team has its own house in order then they usually turn to interactions with other teams and it's worth expanding the invitation list to include people who can bring a wider perspective on these. As a team lead or manager it's hard to maintain a neutral perspective on events. If you work alongside other teams that use retrospectives then it may be possible to take turns to facilitate them for each other. As standard practice at the end of my retrospectives, I gather a 'return on time invested' rating from participants and this might be used a tool for assessing whether a new facilitator is doing a good job if you are trying to build a team of facilitators in an organisation.

Finding a suitable meeting space can make a big difference. It may help to pick a meeting room away from your normal work area so that it's harder for people to get dragged back to work partway through the retrospective. Where possible try to avoid boardroom layout – sitting around a large table immediately places a big barrier between team members – and instead look for somewhere that you can set up a semi-circle of chairs. You also need to check the room has at least a couple of metres of clear wall space or whiteboards. I have learned that when an offsite location is booked for a retrospective it's important to check that there will be space to stick paper up on the wall. I have sometimes been booked to facilitate retrospectives in boardrooms with flock wallpaper, bookcases and antique paintings so we used the doors and up-ended tables to create temporary walls.

As for timing, when working on an iterative planning cycle, you need to hold the retrospective before planning the next efforts. However, running retrospective and planning as back-to-back meetings will be exhausting for everyone so try to separate them out either side of lunch or even on separate days.

Final words

I am sometimes asked, by people wanting to understand more about retrospectives, 'Can you tell me a story that demonstrates a powerful outcome that resulted from a retrospective?'. I have come to realize that this question is similar to 'Can you tell me about a disease that was cured by taking regular exercise?'.

I have worked with teams where running regular heartbeat retrospectives made a big difference in the long term but because the changes were gradual and slow they don't make great headlines. For example, one team I worked with had an issue of how to handle operational requests that came in during their planned product development iterations. It took us a few months before we established a scheme that worked for everyone but without retrospectives it might have taken a lot longer.

The power of regular retrospectives and regular exercise is that they prevent big problems from happening so there should be no war stories or miraculous transformations! ■

References

[Kerth] *Project Retrospectives: A Handbook for Team Reviews* by Norman L. Kerth. Dorset House. ISBN: 0-932633-44-7

[Collins/Miller01] 'Adaptation: XP Style' XP2001 conference paper by Chris Collins & Roy Miller, RoleModel Software

[Schwaber04] *Agile Project Management with Scrum* by Ken Schwaber. Microsoft Press, 2004. ISBN: 978-0735619937

[DeBono93] *Six Action Shoes* by Edward DeBono HarperCollins 1993. ISBN: 978-0006379546

[Derby/Larsen06] *Agile Retrospectives: Making Good Teams Great* by Esther Derby and Diana Larsen. Pragmatic Programmers 2006. ISBN: 0-9776166-4-9

Continuous Integration with CruiseControl.Net

Is CC any good? How could it be better? Did it make a real difference where it was installed? Should we all give it a go?

What is continuous integration?

Continuous integration is vital to the software development process if you have multiple build configurations and/or multiple people working on the same code base. WikiPedia [WikiPedia] describes continuous integration as '…a software engineering term describing a process that completely rebuilds and tests an application frequently.' Generally speaking, continuous integration is the act of automatically building a project or projects and running associated tests; typically after a checkin or multiple checkins to a source control system.

Why should you use continuous integration?

My Dad takes a huge interest in my career and always wants to know what I'm doing. He knows almost nothing about software engineering, but he does know *a lot* about cars, so I often use the production of a car as an analogy to software engineering.

Imagine a car production factory where one department designs and builds the engine, another department designs and builds the gear box and a third department designs and builds the transmission (prop shaft, differential, etc.). The engine has to connect to the gearbox and the gearbox to the transmission. These departments work to a greater or lesser degree in isolation, just like teams or individuals working on different libraries or areas of a common code base on some projects.

A deadline has been set in the factory for all the parts of the car to be ready and the car will be assembled and shipped the next day. During the time to the deadline the gearbox is modified to have four engine mountings, as a flaw in the original design is identified, instead of the three the specification dictates and the ratio of the differential is changed as the sales department has promised the customer the car will have a higher top speed.

The deadline has arrived and the first attempt to assemble the engine, gearbox and transmission is made. The first problem is that the gearbox cannot be bolted onto the engine correctly as there are insufficient mountings on the engine. However this can be fixed, but will take an extra two weeks while the engine block is recast and the necessary mountings added.

Two weeks later the engine, gearbox and transmission are all assembled, bolted into the car and it's out on the test track. The car is flat out down the straight and it is 10 miles per hour slower than sales department promised it would be as the engine designers did not know about the change in differential ratio and the maximum torque occurs at the wrong number of revs. So the car goes back to the factory have the valve timings adjusted which takes another two weeks.

When presented like this it is clear that there is a problem with the way development has been managed. But all too often this the way that software development is done – specs are written and software developed only to be put together under the pressure of the final deadline. Not surprisingly the software is delivered late, over budget and spoils reputations. We've all been there.

The problems could have been avoided or at least identified in time to be addressed, by scheduling regular integrations between the commencement of production and the deadline. Exactly the same applies to software engineering. All elements of the system should be built together and tested regularly to make sure that it builds and that it performs as expected. The ideal time is every time a checkin is made. Integration problems are then picked up as soon as they are created and not the day before the release and the ideal way to do this is using an automated system such as CruiseControl.

CruiseControl.Net

CruiseControl, written in Java, is one of the better known continuous integration systems it is designed to monitor a source control system, wait for checkins, do builds and run tests. CruiseControl.Net [CruiseControl.Net] is, obviously, a .Net implementation of CruiseControl, designed to run on Windows although it can be used with Mono[Mono].

I found the simple start-up documentation for CruiseControl.Net sadly lacking, so in this article I am going to go through a simple CruiseControl.Net configuration step-by-step using my Aeryn [Aeryn] C++ testing framework. Aeryn is an ideal example as it has both Makefiles for building in Unix-like environments and a set of Microsoft Visual C++ build files. It also has a complete test suite which is run as part of the build.

Download and install

You can download CruiseControl.Net from the CruiseControl.Net website. It comes in several different formats including source and a Windows MSI installer. Download and install the Windows MSI and select the defaults. This will install CruiseControl.Net as a service and setup a virtual directory that so it can be used with Microsoft's Internet Information Service [IIS] to give detailed information about the builds (I'll cover this in Part 2). Also download and install the CCTray Windows MSI. CCTray is a handy utility for monitoring builds I'll discuss later.

CruiseControl.Net can run as both a command line program and a Windows service. It is useful to start off with the command line version and then move to the Windows service once all the configuration bugs have been ironed out.

Project block

CruiseControl.Net uses an XML configuration file called `ccnet.config`, which is located in the CruiseControl.Net `server` directory (the default CruiseControl.Net install directory is: `C:\Program Files\CruiseControl.NET`). The configuration must be wrapped in a **`<cruisecontrol>`** block and contain at least one **`<project>`** block:

Paul Grenyer has been a member of the ACCU since 2000 when he started his professional career. As well as founding the ACCU Mentored Developers and serving on the committee, Paul has written a number of articles for CVu and Overload. Paul now contracts at an investment bank in Canary Wharf.

```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\>cd C:\Program Files\CruiseControl.NET\server

C:\Program Files\CruiseControl.NET\server>ccnet
CruiseControl.NET Server 1.2.1.7 -- .NET Continuous Integration Server
Copyright (C) 2003-2006 ThoughtWorks Inc.  All Rights Reserved.
.NET Runtime Version: 2.0.50727.42      Image Runtime Version: v1.1.4322
OS Version: Microsoft Windows NT 5.1.2600 Service Pack 2        Server locale: en-GB

[CCNet Server:DEBUG] The trace level is currently set to debug.  This will cause CCNet to log at the most
verbose level, which is useful for setting up or debugging the server.  Once your server is running
smoothly, we recommend changing this setting in C:\Program
Files\CruiseControl.NET\server\ccnet.exe.config to a lower level.
[CCNet Server:INFO] Reading configuration file "C:\Program Files\CruiseControl.NET\server\ccnet.config"
[CCNet Server:INFO] Registered channel: tcp
[CCNet Server:INFO] CruiseManager: Listening on url: tcp://192.168.0.100:21234/CruiseManager.rem
[CCNet Server:INFO] Starting CruiseControl.NET Server
[Aeryn:INFO] Starting integrator for project: Aeryn
[Aeryn:INFO] No modifications detected.
```

Figure 1

```
<cruisecontrol>
  <project name="Aeryn" ></project>
</cruisecontrol>
```

The above is the minimal project block. In the above example the project is simply given the name Aeryn. It will be added as we step through the configuration. To run CruiseControl.Net from the command line, open a command prompt and change to the `server` directory, type `ccnet` and hit return. The output is similar to that shown in Figure 1.

This starts CruiseControl.Net, but it has nothing to do so it just sits and waits. Now is a good point at which to configure CCTray. Bring up the CCTray window by double clicking on the CCTray icon (usually a green, red or orange circle with CC in the centre) in the system tray. First register the CruiseControl.Net server:

1. Select the file menu and settings.
2. Then select the Add button from the Build Projects tab.
3. Click the Add Server button from the project dialog.
4. Select the 'Connect directly using .Net remoting' radio button.
5. Enter **localhost** to connect to a server on the local machine or the IP address or host name for a server on a remote machine and click Ok.
6. Select the project (in this case Aeryn) from the projects list box and click Ok.
7. Click Ok on the CruiseControl.Net Tray Settings dialog.

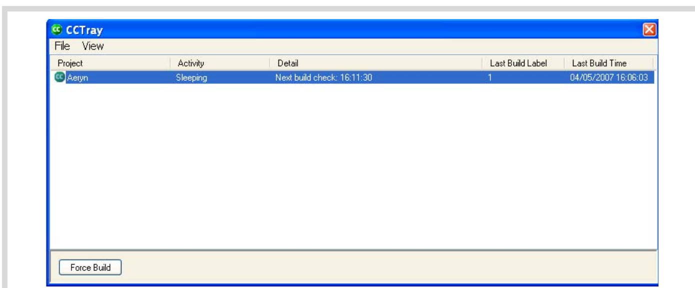


Figure 2

CCTray will connect to the CruiseControl.Net server and you should see the CCTray main window, looking something like Figure 2.

CCTray is designed not only to run on the same machine as CruiseControl.Net, but on any number of client machines as well.

Source control block

CruiseControl.Net can be configured to monitor a number of source control systems for changes. Theses include subversion, CVS, Perforce, ClearCase and Visual Source Safe. The CruiseControl.Net documentation includes a complete list. One of the strengths of CruiseControl.Net is that it is easy to add support, via a plugin, for other source control systems. I plan to write about creating CruiseControl.Net plugins in future articles.

Aeryn uses Subversion [SVN] and CruiseControl.Net. A subversion client must be installed to use it (TortoiseSVN doesn't appear to have the right executable). The minimum parameters needed are the (trunk) URL of the repository and the working directory (a path to check the code out to). However, this assumes that `svn.exe` (subversion client executable) is also in the working directory, so it is necessary to specify the path to it. The full working directory must also exist.

```
<project name="Aeryn">
  <sourcecontrol type="svn">
    <trunkUrl>http://aeryn.tigris.org/svn/aeryn/
       trunk/</trunkUrl>
    <workingDirectory>c:\temp\ccnet\aeryn
       </workingDirectory>
    <executable>C:\Program Files\Subversion\bin\
       svn.exe</executable>
  </sourcecontrol>
</project>
```

```
[CCNet Server:DEBUG] The trace level is currently set to debug.  This will cause CCNet to log at the most
verbose level,  which is useful for setting up or debugging the server.  Once your server is running
smoothly, we recommend changing this setting in C:\Program Files\CruiseControl.NET\server\ccnet.exe.
config to a lower level.
[CCNet Server:INFO] Reading configuration file "C:\Program Files\CruiseControl.NET\server\ccnet.config"
[CCNet Server:INFO] Registered channel: tcp
[CCNet Server:INFO] CruiseManager: Listening on url: tcp://192.168.0.100:21234/CruiseManager.rem
[CCNet Server:INFO] Starting CruiseControl.NET Server
[Aeryn:INFO] Starting integrator for project: Aeryn
[Aeryn:INFO] No modifications detected.
```

Figure 3

```
[Aeryn:DEBUG] Starting process [C:\Program Files\Subversion\bin\svn.exe] in working directory
[c:\temp\ccnet\aeryn] with arguments [log http://aeryn.tigris.org/svn/aeryn/trunk/ -r "{2007-05-
04T16:06:42Z}:{2007-05-04T17:26:17Z}" --verbose --xml --non-interactive --no-auth-cache]
...
[Aeryn:INFO] No modifications detected.
[Aeryn:INFO] Building: Paul Grenyer triggered a build (ForceBuild)
[Aeryn:DEBUG] Starting process [C:\Program Files\Subversion\bin\svn.exe] in working directory
[c:\temp\ccnet\aeryn] with arguments [checkout http://aeryn.tigris.org/svn/aeryn/trunk/
c:\temp\ccnet\aeryn --non-interactive --no-auth-cache]
[Aeryn:DEBUG] A    C:\temp\ccnet\aeryn\corelib
[Aeryn:DEBUG] A    C:\temp\ccnet\aeryn\corelib\corelib.vcproj
[Aeryn:DEBUG] A    C:\temp\ccnet\aeryn\corelib\Makefile
[Aeryn:DEBUG] A    C:\temp\ccnet\aeryn\Doxyfile
[Aeryn:DEBUG] A    C:\temp\ccnet\aeryn\include
...
[Aeryn:DEBUG] A    C:\temp\ccnet\aeryn\examples\lift\TestClient\main.cpp
[Aeryn:DEBUG] U   C:\temp\ccnet\aeryn
[Aeryn:DEBUG] Checked out revision 157.
[Aeryn:INFO] Integration complete: Success - 04/05/2007 18:26:50
```

Figure 4

CruiseControl.Net monitors `ccnet.config`, so simply making the above changes and saving the file should be all that needs to be done. Alternatively the server can be started again from the command line, giving an output similar to Figure 3.

There is some extra debug information, such as the last checkin message, omitted from Figure 3. The final message is **`No modifications detected`**. This means that CruiseControl.Net has identified that there have been no recent changes to the repository. Therefore it has not checked anything out and it has not attempted to build anything.

One way to test that it checks code out correctly would be to commit a change to the repository, however this is unnecessary. CCTray can be used to force the checkout. Select the project from the CCTray list box and click **`Force Build`**. You will get an output similar to Figure 4.

The SVN source control block, unlike some of the other source control blocks, supports username and password parameters. This allows code to be checked out on machines where the current user, such as the system account if CruiseControl.Net is running as a service, does not have the necessary permissions.

```
<sourcecontrol type="svn">
  <trunkUrl>http://aeryn.tigris.org/svn/aeryn/
     trunk/</trunkUrl>
  <workingDirectory>c:\temp\ccnet\aeryn
     </workingDirectory>
  <executable>C:\Program Files\Subversion\bin\
     svn.exe</executable>
  <username>fprefect<username>
  <password>towel<password>
</sourcecontrol>
```

The drawback is that the username and password are stored in `ccnet.config` in unencrypted human readable format. However, CruiseControl.Net only needs to be able to check code out, it doesn't need to check it back in, so if the repository you are using supports anonymous checkouts this is less of a disadvantage.

```
<cruisecontrol>
  <project name="Aeryn">
    <workingDirectory>c:\temp\ccnet\aeryn
       </workingDirectory>
    ...
    <tasks>
      <devenv>
        <solutionfile>aeryn2.sln</solutionfile>
        <configuration>Debug</configuration>
        <executable>C:\Program Files\Microsoft
           Visual Studio … .NET 2003\Common7\IDE\
           devenv.com</executable>
        <buildtype>Rebuild</buildtype>
        <buildTimeoutSeconds>300
           </buildTimeoutSeconds>
      </devenv>
      <devenv>
        <solutionfile>aeryn2.sln</solutionfile>
        <configuration>Release</configuration>
        <executable>C:\Program Files\Microsoft
           Visual Studio … .NET 2003\Common7\IDE\
           devenv.com</executable>
        <buildtype>Rebuild</buildtype>
        <buildTimeoutSeconds>300
           </buildTimeoutSeconds>
      </devenv>
    </tasks>
    ...
  </project>
</cruisecontrol>
```

Listing 1

Devenv task block (Visual Studio 7.x Task)

One of the two basic build systems supported by Aeryn is Microsoft Visual Studio solutions. CruiseControl.Net has two special task blocks for visual studio solutions: **`<devenv>`** and **`<msbuild>`**. **`<devenv>`** is used to build version 7.x solutions and **`<msbuild>`** to build version 8 solutions using Microsoft's MSBuild [MSBuild]. Aeryn uses visual studio 7.1 solutions and therefore requires **`<devenv>`**.

The minimum parameters needed are the solution file to build and the configuration to build (e.g. **`debug`** or **`release`**). However this assumes that Visual Studio 7.x is installed at a specific location, but Visual Studio 7.x can be installed to any path and the default path varies across versions, so it is best to specify the path to the `devenv.com` executable for the version being used.

```
<sourcecontrol type="svn">
    ...
</sourcecontrol>
<tasks>
  <devenv>
    <solutionfile>C:\temp\ccnet\aeryn\aeryn2.sln
       </solutionfile>
    <configuration>Debug</configuration>
    <executable>C:\Program Files\Microsoft Visual
       Studio .NET 2003\Common7\IDE\devenv.com
       </executable>
  </devenv>
</tasks>
```

There are a number of other useful **`<devenv>`** parameters, two of which are: **`<buildtype>`** and **`<buildTimeoutSeconds>`**. The build types are **`Build`**, **`Clean`** and **`Rebuild`** and have their normal visual studio

meanings. The default is **Rebuild**. Build timeout is the number of seconds that CruiseControl.Net will wait before assuming the build has hung and should be killed. The default is 600 (10mins):

```
<devenv>
  <solutionfile>C:\temp\ccnet\aeryn\aeryn2.sln
     </solutionfile>
  <configuration>Debug</configuration>
  <executable>C:\Program Files\Microsoft Visual
     Studio .NET 2003\Common7\IDE\devenv.com
     </executable>
  <buildtype>Rebuild</buildtype>
  <buildTimeoutSeconds>300</buildTimeoutSeconds>
</devenv>
```

Aeryn has both a debug and a release configuration and the building of both should be tested. That requires two **<devenv>** blocks and two fully hard coded solution paths. This introduces a possible maintenance headache if the working directory is moved. Solution paths can be relative if a working directory is specified in the project block, as shown in Listing 1.

Again, the changes to `ccnet.config` should be automatically picked up by the server when it is saved or the server can be restarted from the command line. Using CCTray to force the build will cause both configurations to build.

Exec (make) task block

The other build system supported by Aeryn is make on both Windows and Linux. Obviously CruiseControl.Net can only run the Windows version. CruiseControl.Net doesn't have a specific make task block, so a generic executable block must be used instead.

I think that a task block supporting make is a fundamental omission from CruiseControl.Net. As creating new task blocks is very easy; I have written a **make** task block and am currently trying to get it incorporated into CruiseControl.Net. If I am unsuccessful I will be making it available as a plugin.

The parameters are the path to the executable, the arguments to pass to the executable, the number of seconds to wait before assuming that the process has hung and should be killed, and the working directory. The working directory is only needed if it is different from the working directory specified by the project block.

```
<exec>
  <executable>C:\MinGW\bin\mingw32-make.exe"
     </executable>
  <buildArgs>-f Makefile rebuild</buildArgs>
  <buildTimeoutSeconds>300</buildTimeoutSeconds>
</exec>
```

This disadvantage of using an **<exec>** block to run make is that it does not give any **INFO** level log output indicating that the task is running or whether it was successful, as the **<devenv>** and **<msbuild>** blocks do. CruiseControl.Net also generates a **DEBUG** level error as the output from calling make is not in XML format. This is not a serious problem as when CruiseControl.Net is running in production debug logging should be turned off. The make task block I have written solves both these issues.

Saving the changes to **ccnet.config** and using CCTray to force the build should cause the make configurations to build along with the visual studio configurations.

Publisher block – email

CruiseControl.Net uses publisher blocks to notify developers of the status of the build. The most useful publisher block is email. The email block can be used to send status emails to groups of email addresses.

For example the developer responsible for maintaining the CruiseControl.Net server may want an email every time a build takes place. However, the rest of the developers on the team probably only want to receive an email when the status of the build changes (e.g. from fixed to broken or vice-versa) or while the build is broken. To achieve this, two groups can be setup, a "developers" group and a "buildmaster" group, each group is configured individually.

Emails can be triggered by three different notification events:

- Always — An email is sent every time a build takes place.
- Change — An email is sent when the status of the build changes, either from fixed to broken or from broken to fixed.
- Failed — Sends an email whenever a build fails.

An SMTP server must also be specified along with the relevant username and password when needed. The **<email>** block has the same security issues as the source control block in terms of the username and password being in human readable format. However, most of the time a build server will be fixed within a particular network and access to the SMTP server on a corporate network or through the ISP will not require a username or password.

If CruiseControl.Net is being run on a roaming computer such as a laptop then this becomes more of an issue. I use Google mail for my every day email and the Google SMTP server uses a non-standard port and requires a secure connection. This is not supported by the email block or, it appears, the underlying .Net **SMTP** class. I am sure that both the username and password issue and the port and security issue could be overcome by writing a custom email block based on the existing one, however that is outside the scope of this article.

See Listing 2. Assuming the SMTP details and email addresses are correct, emails will be sent at the end of each build.

When setting up email notifications from continuous integration on multi-developer projects it is important to be aware of how the members of the team feel about potentially receiving a lot of extra email and having the fact that their code changes have broken the build highlighted to the team.

During the setting up of the server it is sensible to restrict emails to the person doing the setup. I found that people became irritated with only a small increase in email to begin with. However as the builds became more successful and the appropriate email rules implemented (see above), this became less of an issue.

To get people to accept that they have broken the build and agree to fix it, I found that it was important to get buy-in for continuous integration. This is an ongoing task. The management, however, is on side and pushing quite hard. I am sure that as soon as the next release is built smoothly everyone will be more enthusiastic about continuous integration and maintaining working builds.

Running CruiseControl.Net as a Service

CruiseControl.Net can be run as a Windows Service. This has the advantage that whenever the dedicated build server is rebooted or someone

```
<project name="Aeryn"> ...
  <publishers>
    <email from=paul.grenyer@gmail.com  mailhost="mailhost.zen.co.uk"
       includeDetails="TRUE">
      <users>
        <user name="Paul Grenyer" group="buildmaster"…
           …address="paul.grenyer@gmail.com"/>
        <user name="Aeryn Developers" group="developers"…
           …address="continuousintegration@aeryn.tigris.org"/>
      </users>
      <groups>
        <group name="developers" notification="change"/>
        <group name="buildmaster" notification="always"/>
      </groups>
    </email>
  </publishers>
</project>
```

Listing 2

logs in or logs out, the CruiseControl.Net server keeps running. Running CruiseControl.Net as a service uses exactly the same `ccnet.config` file.

During the development of the configuration it useful to have lots of debug information. Once the configuration is complete and working; this extra debug information is no longer useful and should be turned off. To adjust the logging level, edit the **`<level value>`** tag in the `ccservice.exe.config` file in the CruiseControl.Net `server` directory. The available levels are **`DEBUG`**, **`INFO`**, **`WARN`**, **`ERROR`**, **`OFF`**. The default is **`Debug`**. Changing the setting to **`INFO`** reduces a lot of unnecessary noise:

```
<level value="INFO" />
```

Changes to `ccservice.exe.config` are not picked up by CruiseControl.Net until it is restarted.

CruiseControl.Net is installed as a Windows service as part of the standard setup. Before it can be started it must be configured to run as a user that has access to the source control system (unless the username and password have been put into source control block) and the compilers and applications that are used in the configuration.

1. Open the **Services** dialog (Control Panel->Administrative Tools->Services) and double click on CruiseControl.Net Server.
2. In the **General** tab set **Start Type** to **Automatic** so that CruiseControl.Net starts when the build server starts.
3. In the **Log On** tab select the **This Account** radio button and enter the username and password of a user who has rights to the necessary source control and application.
4. Click **Ok** and use the **Services** dialog to start CruiseControl.Net.

CruiseControl.Net writes a log file called `ccnet.log` to the `Service` directory. It can be useful to monitor this with a tool such as tail [Tail]. When CruiseControl.Net is running as a service a build can be forced from CCTray in the same way as when it was running from the command line. The complete `ccnet.config` file is shown in Listing 3.

```
<cruisecontrol>
  <project name="Aeryn">
    <workingDirectory>c:\temp\ccnet\aeryn</workingDirectory>
    <sourcecontrol type="svn">
      <trunkUrl>http://aeryn.tigris.org/svn/aeryn/trunk/</trunkUrl>
      <workingDirectory>c:\temp\ccnet\aeryn</workingDirectory>
      <executable>C:\Program Files\Subversion\bin\svn.exe
         </executable>
    </sourcecontrol>
    <tasks>
      <devenv>
        <solutionfile>aeryn2.sln</solutionfile>
        <configuration>Debug</configuration>
        <executable>C:\Program Files\Microsoft Visual Studio…
          ….NET 2003\Common7\IDE\devenv.com</executable>
        <buildtype>Rebuild</buildtype>
        <buildTimeoutSeconds>300</buildTimeoutSeconds>
      </devenv>
      <devenv>
        <solutionfile>aeryn2.sln</solutionfile>
        <configuration>Release</configuration>
        <executable>C:\Program Files\Microsoft Visual Studio…
          ….NET 2003\Common7\IDE\devenv.com</executable>
        <buildtype>Rebuild</buildtype>
        <buildTimeoutSeconds>300</buildTimeoutSeconds>
      </devenv>
      <exec>
        <executable>C:\MinGW\bin\mingw32-make.exe</executable>
        <buildArgs>-f Makefile rebuild</buildArgs>
        <buildTimeoutSeconds>300</buildTimeoutSeconds>
      </exec>
    </tasks>
    <publishers>
      <email from=paul.grenyer@gmail.com…
        …mailhost="mailhost.zen.co.uk" includeDetails="TRUE">
        <users>
          <user name="Paul Grenyer" group="buildmaster"…
            …address="paul.grenyer@gmail.com"/>
          <user name="Aeryn Developers" group="developers"…
            …address="continuousintegration@aeryn.tigris.org"/>
        </users>
        <groups>
          <group name="developers" notification="change"/>
          <group name="buildmaster" notification="always"/>
        </groups>
      </email>
    </publishers>
  </project>
</cruisecontrol>
```

Listing 3

Final test

As a final test modify a source file in such a way as to break the build. Commit the file to the source control system and see that it triggers the CruiseControl.Net build and that the build fails. Then undo the modification, commit it again and see that the build succeeds. `ccnet.config` should also be committed to the source control system and checking it in will also trigger a build.

Part 2

In this article I have demonstrated how easy it is to setup continuous integration with CruiseControl.Net. I found the documentation lacking and I hope this article has started to rectify that situation. I have highlighted some of CruiseControl.Net's shortcomings, not only its lack of documentation, but that it is missing at least one fundamentally important task block and that the source control and email blocks need to have additional features. I intend to address these issues with plugins in future articles.

I think it is clear that, because continuous integration highlights integration issues early, it is something we should all be doing on multi-developer projects or projects with multiple build systems. This is what I have found in the relatively short period of time I have been using continuous integration.

In part two, I am going to look at setting up a webserver to allow more detailed information to be obtained about the status of the CruiseControl.Net server and its projects and builds. ■

Acknowledgments

Thank you to Jez Higgins, Peter Hammond, Roger Orr, Paul Thomas and Alan Griffiths for reviews and suggestions.

References

[WikiPedia] http://en.wikipedia.org/wiki/Continuous_Integration
[CruiseControl.Net] http://ccnet.thoughtworks.com/
[Aeryn] http://www.aeryn.co.uk
[Mono] http://www.mono-project.com
[IIS] http://www.microsoft.com/windowsserver2003/iis/default.mspx
[SVN] http://subversion.tigris.org/
[MSBuild] http://msdn2.microsoft.com/en-us/library/wea2sca5(VS.80).aspx
[Tail] http://tailforwin32.sourceforge.net/

Working with GNU Export Maps

Taking control over the symbols exported from shared libraries built with the GNU toolchain.

Introduction

Recently I've been preparing version 2 of an internal shared library for my company and I needed to make sure that both the old version of the library and the new one could be loaded into a process at the same time, without causing problems. I knew from previous work that the answer was probably GNU export maps, but my memory was distinctly rusty on the details. Having been to the discussion on writing for the ACCU at the conference, it occurred to me that other people might also find some notes on the topic useful.

Exporting C++ from a shared library

When exporting symbols from a shared library, the GNU ELF shared library linker behaves in a significantly different way to the Microsoft Windows linker. On Windows, nothing is exported from a DLL unless the programmer explicitly requests it. The GNU ELF linker, on the other hand, exports everything by default.

The GNU ELF default undoubtedly makes initial C++ application development simpler; hands up everyone who has at some point struggled to export a class from a DLL, because it uses an STL container... There's a cost to that initial simplicity though. A C++ shared library will typically contain a large number of symbols. When an application is linked against that library, the compiler and linker generate a reference for each of those symbols. When the library is loaded at run time, each of those references has to be bound to the corresponding symbol in the shared library.

Let's take a look at a trivial example (Listing 1).

If we build `spaceship.cpp` into a shared library and `testflight.cpp` into an executable linked against that library, we can examine what happens at runtime, using the **`LD_DEBUG`** environment variable.

```
> g++ -shared -fPIC spaceship.cpp -o
libspaceship.so.1 -Wl,-soname=libspaceship.so.1
> ln -s libspaceship.so.1 libspaceship.so
> g++ testflight.cpp -L. -lspaceship -o testflight
> export LD_DEBUG=symbols
> export LD_LIBRARY_PATH=.
> ./testflight
```

This produces a lot of output. Digging through it, we see some things that we are expecting to be resolved to our library, like those shown in Figure 1.

But we also see a lot more that we might not have expected, like those in Figure 2.

Ian Wakeling Ian first started programming on a ZX81 and hasn't managed to stop yet. He has been paid for indulging the compulsion for the last 14 years, mainly in C++, on both Unix and Windows. He can be contacted at ian.wakeling@ntlworld.com

In total, there are twenty one symbols that get resolved to our library. That's quite a lot of fix-ups for such a small amount of code. It's worse than it immediately looks, as well, because each lookup is done by doing a string compare against each possible function in each library, until a match is found, so the number of symbols exported from our library affects not just how many symbols have to be fixed up by the executable, but how many string matches have to be done for each fix-up. Imagine how that scales up for a real C++ library.

```
// spaceship.h
#include <string>
#include <vector>

namespace scifi
{
class Spaceship
{
public:
  Spaceship( std::string const& name );
  ~Spaceship();
  void stabliseIonFluxers();
  void initiateHyperwarp();
private:
  Spaceship( Spaceship const& );
  Spaceship& operator=( Spaceship const& );
private:
  typedef unsigned int             FluxLevel;
  typedef std::vector<FluxLevel>  FluxLevels;
private:
  void doSomethingInternal();
  FluxLevel checkFluxLevel( size_t ionFluxerIdx );
private:
  std::string m_name;
  FluxLevels  m_fluxLevels;
};
}

// spaceship.cpp omitted for brevity

// testflight.cpp
#include "spaceship.h"
int main( int, char** )
{
  scifi::Spaceship*  ship = new scifi::Spaceship(
     "Beagle" );
  ship->stabiliseIonFluxers();
  ship->initiateHyperwarp();
  delete ship;
  return 0;
}
```

Listing 1

all that implementation detail will still be exported and available for perusal by anyone who cares to run freely available tools…over the shared library

```
5975: symbol=_ZN9SpaceshipC1ERKSs;  lookup in file=./testflight
5975: symbol=_ZN9SpaceshipC1ERKSs;  lookup in file=./libspaceship.so.1
5975: symbol=_ZN9Spaceship19stabiliseIonFluxersEv;  lookup in file=./testflight
5975: symbol=_ZN9Spaceship19stabiliseIonFluxersEv;  lookup in file=./libspaceship.so.1
```

Figure 1

```
5975: symbol=_ZNSt6vectorIjSaIjEEC1IiEET_S3_RKS0_;  lookup in file=./testflight
5975: symbol=_ZNSt6vectorIjSaIjEEC1IiEET_S3_RKS0_;  lookup in file=./libspaceship.so.1
5975: symbol=_ZNSt18_Vector_alloc_baseIjSaIjELb1EE11_M_allocateEj;  lookup in file=./testflight
5975: symbol=_ZNSt18_Vector_alloc_baseIjSaIjELb1EE11_M_allocateEj;  lookup in file=./libspaceship.so.1
```

Figure 2

Notice how even though we didn't deliberately export them from our library, there are quite a lot of symbols from STL instantiations being looked up there; in fact they swamp the symbols we actually intended to export.

We can use **nm** to look at what's being exported from our library:

```
> nm -g -D -C --defined-only libspaceship.so.1
```

As with the output generated by **LD_DEBUG**, we see some symbols that relate directly to our class, as shown in Figure 3, and we also see many other symbols relating to the STL classes we used in the implementation, like those shown in Figure 4.

In amongst the noise of the weakly defined STL template instantiations being exported from our library, notice how our private member functions are also exported.

Remember that we did not build with debug information. Also don't be fooled into thinking that it's because the header file listed them; remember that there's nothing special about header files in C++; the compiler and linker don't even know they exist, so even if we use idioms like Cheshire Cat or abstract base classes, all that implementation detail will still be exported and available for perusal by anyone who cares to run freely available tools like **nm** over the shared library.

```
0000132a T scifi::Spaceship::checkFluxLevel(unsigned int)
0000131e T scifi::Spaceship::initiateHyperwarp()
00001324 T scifi::Spaceship::doSomethingInternal()
00001318 T scifi::Spaceship::stabiliseIonFluxers()
00001134 T scifi::Spaceship::Spaceship(std::string const&)
00001270 T scifi::Spaceship::~Spaceship()
```

Figure 3

For some projects, this could represent an unacceptable IP leakage.

If those problems don't concern you, there is another issue you might like to consider.

Let's imagine that we have successfully deployed version 1 of our **spaceship** library. It's being used in a few places and is perhaps referenced by a few other shared libraries. Consider what happens if we now want to do a version 2, which isn't compatible. Obviously, we'll build it into `libspaceship.so.2`, with the **SONAME** set appropriately, so we're versioned and everything is OK, right?

Not quite. When the dynamic linker is resolving symbols, it simply searches the list of modules, in order. There is no information in the symbol to say which library it ought to be loaded from. So let's imagine that one part of our application is linked against `libspaceship.so.2`, but another part hasn't been updated yet and still links against `libspaceship.so.1`. If `libspaceship.so.1` gets loaded first, then whenever the **Spaceship** constructor is searched for, the one in `libspaceship.so.1` will always be found. If we then try to use a facility that we added to `libspaceship.so.2`, disaster will ensue.

Fortunately, there is a mechanism which can solve both problems. What we need to do is take control over which symbols are exported from our library. The GNU tool-chain offers a few ways of doing this. One involves decorating the

```
00001330 W std::allocator<unsigned int>::allocator()
00001336 W std::allocator<unsigned int>::~allocator()
0000154a W std::_Vector_base<unsigned int, std::allocator<unsigned int>
>::_Vector_base(std::allocator<unsigned int> const&)
0000147c W std::_Vector_base<unsigned int, std::allocator<unsigned int> >::~_Vector_base() 000014e6 W
std::__simple_alloc<unsigned int, std::__default_alloc_template<true, 0> >::deallocate(unsigned int*,
unsigned int)
```

Figure 4

code with `__attribute__ ((visibility("xxx"))` tags; another, introduced with GCC 4.0, uses `#pragma GCC visibility`, but I'm going to focus on GNU Export Maps, sometimes called Version Scripts. This is partly because I don't like adding large amounts of tool-chain specific decoration to my code and partly because, at present, as far as I know, only export maps can help with versioning symbols.

An export map is simply a text file listing which symbols should be exported and which should not. A really simple example to export one 'C' function called `foo` from a shared library would look like this:

```
{
  global:
    foo;
  local:
    *;
};
```

Unfortunately, the situation for C++ is, inevitably, slightly more complex... you've guessed, of course: name mangling! Export maps are used by the linker, by which time the compiler has mangled the names. The good news is that the GNU linker understands the GNU compiler's C++ name mangling, we just have to tell it that the symbols are C++.

So for our spaceship, we might write:

```
{
  global:
    extern "C++" {
      *scifi::Spaceship;
      scifi::Spaceship::Spaceship*;
      scifi::Spaceship::?Spaceship*;
      scifi::Spaceship::stabliseIonFluxers*;
      scifi::Spaceship::initiateHyperwarp*
    };
  local
    *;
};
```

A few points need explaining here.

The first entry exports the typeinfo for the class. In this example, it's not strictly necessary, but I have included it to show how. See the sidebar entitled 'Exporting TypeInfo' for an explanation of why you might need to do so.

Tilde (~) is not a valid character in an export map, so in the export line for the destructor, we replace it with a single character wildcard.

Next, notice how within the extern C++ block, every entry ends with a semi-colon except the last. This is not a typo! The syntax is defined that way.

Lastly, the wildcards on the end of the function names are because the full function name includes its signature and we don't want to have to write it out here.

Let's build our example using the example export map and see what happens. This is done by passing an extra option to the linker:

```
> g++ -shared spaceship.cpp -o libspaceship.so.1 -
Wl,-soname=libspaceship.so.1 -Wl,--version-
script=spaceship.expmap
> g++ testflight.cpp -L. -lspaceship -o testflight
```

First the output of `nm`, to show that we are now only exporting what we actually want to (see Figure 5).

All of the implementation details of our class are now safely hidden away as they should be and only our public interface is visible outside the library.

Obviously, the first thing we must do is run the test harness to check that we haven't broken anything by restricting the exports. It runs without any problems, so we can use `LD_DEBUG` again to see what difference it has made to the runtime behaviour. Filtering the output to show only those symbols that were resolved to the spaceship library, this time we get Figure 6.

This looks more like we'd want this time: the only things being resolved to the library are the functions that make up the library's public interface. What's more, if you compare the raw output of the two runs, you'll notice that there are fewer lookups being performed in total, because we no longer have the weak symbols to be resolved when our library is loaded.

Symbol versioning

At its simplest, this requires a simple addition to the export map:

```
SPACESHIP_1.0 {
  global:
    extern "C++" {
      *scifi::Spaceship;
      scifi::Spaceship::Spaceship*;
      scifi::Spaceship::~Spaceship*;
      scifi::Spaceship::stabliseIonFluxers*;
      scifi::Spaceship::initiateHyperwarp*
    };
  local
    *;
};
```

In order to see the effect this has, we need to use `objdump`, rather than `nm`, because `nm` does not display symbol versioning information. First, if we look at the symbols exported from `libspaceship.so.1`, we can see that they are all now marked with a version. `objdump` doesn't have a filtering option equivalent to nm's `--defined-only`, so I have picked out just the relevant lines from its output in Figure 7.

Notice how each export is now marked with the version string we gave. Also, there is a single extra absolute symbol which states that this shared library provides this version of the ABI.

Now let's look at the imports in the test executable. Again, I'm going to pick out just the entries (Figure 8) that relate to `libspaceship`. If you do this for yourself, you'll see a lot more entries for `glibc` and `libstdc++`.

Not only does the executable state, as usual, that it needs `libspaceship`, but it now states that it needs a version of `libspaceship` that provides the right version of the ABI. In addition and more importantly, each symbol being imported from our library is now marked with the required version.

```
> nm -g -D -C --defined-only libspaceship.so.1
00000b4e T scifi::Spaceship::initiateHyperwarp()
00000b48 T scifi::Spaceship::stabiliseIonFluxers()
00000a02 T scifi::Spaceship::Spaceship(std::string const&)
00000964 T scifi::Spaceship::Spaceship(std::string const&)
00000af4 T scifi::Spaceship::~Spaceship()
00000aa0 T scifi::Spaceship::~Spaceship()
```

Figure 5

```
13421: symbol=_ZN5scifi9SpaceshipC1ERKSs;  lookup in file=./testflight
13421: symbol=_ZN5scifi9SpaceshipC1ERKSs;  lookup in file=./libspaceship.so.1
13421: symbol=_ZN5scifi9Spaceship19stabiliseIonFluxersEv;  lookup in file=./testflight
13421: symbol=_ZN5scifi9Spaceship19stabiliseIonFluxersEv;  lookup in file=./libspaceship.so.1
13421: symbol=_ZN5scifi9Spaceship17initiateHyperwarpEv;  lookup in file=./testflight
13421: symbol=_ZN5scifi9Spaceship17initiateHyperwarpEv;  lookup in file=./libspaceship.so.1
13421: symbol=_ZN5scifi9SpaceshipD1Ev;  lookup in file=./testflight
13421: symbol=_ZN5scifi9SpaceshipD1Ev;  lookup in file=./libspaceship.so.1
```

Figure 6

Exporting Typeinfo

It is important to export **typeinfo** for your classes if you want any language features that rely on **typeinfo**, like **dynamic_cast** or exceptions, to work across the shared library interface. This is because when the GCC runtime compares two **typeinfo**s, it does so by pointer. If a shared library does not export the **typeinfo** for a class it defines, then the executable will contain its own copy generated from the relevant header file. Thus when when an instance of a class is created inside the library, it will have the library's copy of the **typeinfo** associated with it, but when a client executable performs a **dynamic_cast** or a **catch(TheType)**, it will be looking for the executable's copy of the **typeinfo** and the two will not match, leading to unpleasant surprises and extended debugging sessions...

There are some more sophisticated things that can be done with symbol versioning, such as marking which minor version of an interface symbols were introduced in, by having more than one section in the export map. See the reference at the end for more information on this.

Making it easier

There's only one slight fly in the ointment. For a trivial example, that export map looks fine, but maintaining it for a real library could quickly become painful.

One answer, that works for some circumstances, is to think carefully about how much you actually need to export from your shared library. If you are programming to interfaces expressed as abstract base classes, then you probably also have factory functions to create instances of implementation classes, which return a pointer to the interface. In that case, it often turns out the the only thing that needs to be exported from the shared library is that factory function. On one project that I work on, we have a number of shared libraries that are loaded dynamically at run time that have this property. Because the libraries are loaded dynamically and the factory function is found at runtime using **dlsym()**, the factory function can have the same name in every such component and so we can generate the export map at build time and don't have to maintain them at all.

That is not always a sensible or workable approach though; if you are writing a C++ class library, then you need to export the classes.

By deciding to write an export map, we have, in effect, created the same situation that we have on Windows: we are starting with an empty 'global' section in our export map, so that nothing is exported and we're now trying to get a list of what to export. On Windows, this is often done by decorating the code with some special directives for the Microsoft tool-chain. (Note that these directives occur in a different place in the source code to the GNU **__attribute__** directive.) These are often hidden away in a macro, because different directives are needed when building the library and when linking against it:

```
#if defined( _WIN32 )
#  if defined(SPACESHIP_EXPORT_INTERFACE)
#    define SPACESHIP_API __declspec(dllexport)
#  else
#    define SPACESHIP_API __declspec(dllimport)
#  endif
#else
#  define SPACESHIP_API
#endif
class SPACESHIP_API Spaceship
{
  // etc
};
void SPACESHIP_API myGlobalFunction();
```

If all the classes that are to be exported are marked this way, then it ought to be possible to write a tool that will generate the export map for us.

An approach that I've been experimenting with is to use regular expression matching to find interesting declarations (namespaces, classes / structures and functions) in header files. Tracking instances of opening and closing block braces allows the generation of scoped names in the export map. Of course, it's not possible (as far as I'm aware) to write a regular expression that will correctly identify function declarations in the general case, but we can spot global functions that are to be exported by the presence of the **SPACESHIP_API** tag and if we avoid writing any serious implementation inside our exportable class declarations, then we can spot function declarations within the class declaration body. Looking out for **public** / **protected** / **private** tags allows us to avoid exporting implementation functions.

Anyone fancy a summer project? ■

References

- *How To Write Shared Libraries*: Ulrich Drepper, 2005
- Linux man and info pages, nm, objdump, ld: various


```
> objdump -T -C libspaceship.so.1
libspaceship.so.1:     file format elf32-i386
DYNAMIC SYMBOL TABLE:
00000a92 g    DF .text  0000009d  SPACESHIP_1.0 scifi::Spaceship::Spaceship(std::string const&)
00000bde g    DF .text  00000005  SPACESHIP_1.0 scifi::Spaceship::initiateHyperwarp()
00000b84 g    DF .text  00000054  SPACESHIP_1.0 scifi::Spaceship::~Spaceship()
00000000 g    DO *ABS*  00000000  SPACESHIP_1.0 SPACESHIP_1.0
00000b30 g    DF .text  00000054  SPACESHIP_1.0 scifi::Spaceship::~Spaceship()
000009f4 g    DF .text  0000009d  SPACESHIP_1.0 scifi::Spaceship::Spaceship(std::string const&)
```

Figure 7

```
> objdump -x testflight
Dynamic Section:   NEEDED      libspaceship.so.1
Version References:
required from libspaceship.so.1:     0x04a15a30 0x00 03 SPACESHIP_1.0
SYMBOL TABLE:
00000000       F *UND*  00000005          _ZN5scifi9Spaceship17initiateHyperwarpEv@@SPACESHIP_1.0
00000000       F *UND*  00000054          _ZN5scifi9SpaceshipD1Ev@@SPACESHIP_1.0
00000000       F *UND*  0000009d          _ZN5scifi9SpaceshipC1ERKSs@@SPACESHIP_1.0
00000000       F *UND*  00000005          _ZN5scifi9Spaceship19stabiliseIonFluxersEv@@SPACESHIP_1.0
```

Figure 8

auto_value: Transfer Semantics for Value Types

std::auto_ptr has a reputation for causing problems due to its surprising copy/assignment semantics. Richard Harris tries to separate the good ideas from the bad.

The problem of eliminating unnecessary copies is one that many programmers have addressed at one time or another. This article proposes an alternative to one of the most common techniques, copy-on-write. We'll begin with a discussion on smart pointers, and why we need more than one type of them. We'll then look at the relationship between smart pointers and the performance characteristics of some simple string implementations, including one that supports copy-on-write. I'll suggest that a different choice of smart pointer better captures our intent, and show that what we were trying to do doesn't achieve half as much as we thought it would.

Finally, I hope to show that whilst the problem we set out to solve turns out to be a bit of a non-issue, this technique has a side effect that can be exploited to dramatically improve the performance of some complex operations.

`article::article()`

I'd like to begin by recalling Jackson's Rules of Optimization.

- Rule 1: Don't do it.
- Rule 2 (for experts only): Don't do it yet.

This is probably a little presumptuous of me, but I'd like to add something:

- Harris's Addendum: Nyah, nyah. I can't hear you.

I'll admit it's not very mature, but I think it accurately reflects how we all *really* feel. No matter how much a programmer preaches, like Knuth, that premature optimisation is the root of all evil, I firmly believe that deep down they cannot help but recoil at inefficient code.

Don't believe me? Well, ask yourself which of the following function signatures you'd favour:

```
void f(std::vector<std::string> strings);
void f(const std::vector<std::string> &strings);
```

Thought so.

But you mustn't feel bad about it, the desire to write optimal code is a *good* thing. And I can prove it.

In their seminal 2004 paper, 'Universal Limits on Computation', Krauss and Starkman demonstrated that the universe will only be able to process another 1.35×10^{120} bits during its lifetime. Count them. Just 1.35×10^{120}. If Moore's Law continues to hold true, we'll run out of bits in 600 years.

So we can stop feeling guilty about our oft-criticised drive to optimise, because wasted CPU cycles are accelerating the heat death of the universe.

Will nobody think of the children?

Act responsibly.

Optimise.

Richard Harris Richard has been a professional programmer since 1996. He has a background in Artificial Intelligence and numerical computing and is currently employed writing software for financial regulation.

OK, that's a little disingenuous. Knuth actually said that in 97% of cases we should forget about small efficiencies. I suspect that we'd all agree that using `const` references by default for value types generally falls into the 3% that we shouldn't ignore. There is, after all, a world of difference between choosing the more efficient of two comparably complex statements and significantly increasing the complexity of your code in the name of a relatively small efficiency gain.

There is a grey area though. Sometimes it really is worth increasing the complexity of your code for relatively small gains. Especially when the particular source of inefficiency occurs regularly within your code base and you can hide that complexity behind a nice tightly defined class interface.

This article is going to take a look at those most profligate of wastrels, temporaries.

But since the direct route rarely has the most interesting views, we're going to set off with a discussion on smart pointers.

auto_ptr

Let's start by taking a look at the definition of `auto_ptr` (see Listing 1). It's a little bit more complicated than you'd expect isn't it?

```
template<typename X>
class auto_ptr
{
public:
typedef X element_type;
  explicit auto_ptr(X *p = 0) throw();
  auto_ptr(auto_ptr &p) throw();
  template<class Y> auto_ptr(auto_ptr<Y> &p)
     throw();
  auto_ptr(auto_ptr_ref<X> p) throw();
  ~auto_ptr() throw();
  auto_ptr & operator=(auto_ptr &p) throw();
  template<class Y> auto_ptr &
     operator=(auto_ptr<Y> &p) throw();
  auto_ptr & operator=(auto_ptr_ref<X> p)
     throw();
  template<class Y> operator auto_ptr_ref<Y>()
     throw();
  template<class Y> operator auto_ptr<Y>()
     throw();
  X & operator*() const throw();
  X * operator->() const throw();
  X * get() const throw();
  X * release() throw();
  void reset(X *p = 0) throw();
private:
  X *x_;
};
```

Listing 1

like HAL from Clark's 2001: A Space Odyssey, it has been driven ever so slightly barking mad from being given two competing responsibilities

This is because, like HAL from Clark's 2001: A Space Odyssey, it has been driven ever so slightly barking mad from being given two competing responsibilities. The first of these is to tie the lifetime of an object to the scope in which it's used.

For example:

```
void
f()
{
  const auto_ptr<T> t(new T);
  //...
} //object is destroyed here
```

The destructor of the `auto_ptr` deletes the object it references, ensuring that it is properly destroyed no matter how we leave the scope.

The second responsibility is to safely transfer objects from one location to another.

For example:

```
auto_ptr<T>
f()
{
  return auto_ptr<T>(new T);
}

void
g(auto_ptr<T> t)
{
  //...
} //object is destroyed here
void
h()
{
  auto_ptr<T> t;
  t = f(); //ownership is transferred from f here
  g(t);    //ownership is transferred to g here
}
```

The two roles are distinguished by the `const`ness of the `auto_ptr` interface. The `const` member functions of `auto_ptr` manage lifetime control, whereas the non-`const` member functions manage ownership transfer.

For example, by making the variable `t` in the function `h` in the previous example `const`, we can ensure that the compiler will tell us if we accidentally try to transfer its ownership elsewhere:

```
void
h()
{
  const auto_ptr<T> t;
  t = f(); //oops
  g(t);    //oops
}
```

And herein lies the problem. We want `const auto_ptrs` to jealously guard their contents, so we make the arguments to the transfer constructor and assignment operator *non*-`const` references. But, unnamed temporaries, such as function return values, can only be bound to `const` references, making it difficult to transfer ownership of an object out of one function and into another.

For example:

```
void
h()
{
  g(f()); //now I'm confused
}
```

This is where the mysterious `auto_ptr_ref` class comes to our rescue. You'll note that `auto_ptr` has a non-`const` conversion to `auto_ptr_ref` and that there's a conversion constructor that takes an `auto_ptr_ref`. So a non-`const` unnamed temporary can be converted to an `auto_ptr_ref`, which will in turn transfer ownership to an `auto_ptr` via the conversion constructor.

Neat, huh?

Well, perhaps. But we could almost certainly do better by giving each of `auto_ptr`'s personalities its own body. We'll do this by introducing a new type, `scoped_ptr`, to manage object lifetimes and stripping those responsibilities from `auto_ptr`.

The definition of `scoped_ptr` is as shown in Listing 2.

```
template<typename X>
class scoped_ptr
{
public:
  typedef X element_type;

  explicit scoped_ptr(X *p = 0) throw();
  explicit scoped_ptr(
     const auto_ptr<X> &p) throw();
  ~scoped_ptr() throw();

  X & operator*() const throw();
  X * operator->() const throw();
  X * get() const throw();

  auto_ptr<X> release() throw();

private:
  scoped_ptr(const scoped_ptr &); // not
                                  // implemented
  scoped_ptr & operator=(const scoped_ptr &);
                                 //not implemented
  X * x_;
};
```

Listing 2

So we lie to the compiler. We tell it that "no, really, this function is const" and use mutability to change the object pointer anyway.

Those in the know will see the similarity with the boost **scoped_ptr** (www.boost.org). This is only natural since I pretty much just swiped it from there.

As with the original **auto_ptr**, the constructors take ownership of the objects passed to them and the destructor destroys them. The principal change is that it is no longer legal to copy or assign to **scoped_ptr**s (the unimplemented private **copy** constructor and assignment operator are there to suppress the automatically generated ones).

We can continue to use **scoped_ptr** to tie object lifetime to scope:

```
void
f()
{
  scoped_ptr<T> t(new T);
  //...
} //object is destroyed here
```

But we can no longer use it to transfer ownership:

```
scoped_ptr<T> //oops, no copy constructor
f()
{
  return scoped_ptr<T>(new T);
}

void
g(scoped_ptr<T> t) //oops, no copy constructor
{
  //...
}

void
h()
{
  scoped_ptr<T> t;
  t = scoped_ptr<T>(new T); //oops, no assignment
                            // operator
}
```

Now let's have a look at how giving up responsibility for lifetime control changes **auto_ptr** (Listing 3).

OK, so I lied a little.

We haven't so much lost the ability to control object lifetimes with **auto_ptr** as made it a little less attractive. The constructors still take ownership of the objects passed to them and the destructor still destroys them, but holding on to them is difficult.

This is because the object pointer is now *mutable*, allowing it to be changed even through **const** member functions. Usually mutable is reserved for members that can change whilst the object maintains the appearance of **const**ness (caches, for example), an idea typically described as logical **const**ness. Here, to be honest, it's a bit of a hack. We need to tell the compiler to abandon all notions of **const**ness for **auto_ptr**s and unfortunately we can't do that (unnamed temporaries rear their problematic heads again). So we lie to the compiler. We tell it that "no, really, this function is **const**" and use mutability to change the object pointer anyway.

```
template<typename X>
class auto_ptr
{
public:
  typedef X element_type;

  explicit auto_ptr(X *p = 0) throw();
  auto_ptr(const auto_ptr &p) throw();
  template<class Y> auto_ptr(
     const auto_ptr<Y> &p) throw();
  ~auto_ptr() throw();

  const auto_ptr & operator=(
     const auto_ptr &p)    const throw();
  template<class Y>
    const auto_ptr & operator=(
       const auto_ptr<Y> &p) const throw();

  X & operator*() const throw();
  X * operator->() const throw();

  X * release() const throw();

private:
  mutable X *x_;
};
```

Listing 3

We can still use **auto_ptr** to transfer object ownership from one place to another:

```
auto_ptr<T>
f()
{
  return auto_ptr<T>(new T);
}
void
g(auto_ptr<T> t)
{
  //...
} //object is destroyed here
void
h()
{
  auto_ptr<T> t;
  t = f(); //ownership is transferred from f here
  g(t);    //ownership is transferred to g here
}
```

If you would like to bring a little joy into someone's life, ask a Java programmer about garbage collection

But we might run into problems if we try to use it to control object lifetime:

```
T
h()
{
  const auto_ptr<T> t;
  g(t);      //ownership is transferred to g here
  return *t; //oops
}
```

It's precisely because this new **auto_ptr** is so slippery, that I've added a release method to boost's **scoped_ptr**. This enables us to use the **scoped_ptr** to control the lifetime of the object within a function and **auto_ptr** to control its transfer during function return.

For example:

```
auto_ptr<T>
f()
{
  scoped_ptr<T> t;
  //...
  return t.release();
}

void
g()
{
  scoped_ptr<T> t(f());
}
```

I shall keep the name **auto_ptr** for this new ownership transfer pointer despite many reasonable arguments against doing so. Firstly, I believe that **auto_ptr** has strong associations with transfer semantice for most of us. Secondly, and far more importantly, it has led to a much snappier title for this article than the alternative.

Henceforth, therefore, when we refer to **auto_ptr**, we will mean this new version, having the sole responsibility of ownership transfer.

shared_ptr

Another approach to managing object lifetimes is to allow multiple references to share ownership of an object. This is achieved by keeping the object alive for as long as something is referring to it.

There are two common techniques used to do this, the correct approach and the easy approach. Guess which one we're going to look at.

That's right. Reference counting.

Reference counting works by keeping a count of the number of active references to an object and deleting it once this count drops to zero. Each time a new reference is taken, the count is incremented and each time a reference is dropped, the count is decremented. This is generally achieved by requiring the referencing entity to explicitly register its interest or disinterest in the object.

The chief advantage of using reference counting to implement shared ownership semantics is that it's relatively simple compared to the alternative.

The chief disadvantage occurs when an object directly or indirectly holds a reference to itself, such as when two objects hold references to each other. In this situation, the reference count will not fall to zero unless one of the objects explicitly drops the reference to the other. In practice, it can be extremely difficult to manually remove mutual references since the ownership relationships can be arbitrarily complex. If you would like to bring a little joy into someone's life, ask a Java programmer about garbage collection.

We can automate much of the book-keeping required for reference counting by creating a class to manage the process for us. In another shameless display of plagiarism I'm going to call this class **shared_ptr** (see Listing 4).

```
template<typename X>
class shared_ptr
{
public:
  typedef X element_type;

  shared_ptr() throw();
  template<class Y> explicit shared_ptr(Y * p);
  shared_ptr(const shared_ptr &p) throw();
  template<class Y> shared_ptr(
     const shared_ptr<Y> &p) throw();
  template<class Y> explicit shared_ptr(
     const auto_ptr<Y> &p);
  ~shared_ptr() throw();

  shared_ptr & operator=(
     const shared_ptr &p) throw();
  template<class Y>
    shared_ptr & operator=(
       const shared_ptr<Y> &p) throw();
  template<class Y>
    shared_ptr & operator=(
       const auto_ptr<Y> &p) throw();

  X & operator*() const throw();
  X * operator->() const throw();
  X * get() const throw();

  void reset(X *p = 0) throw();
  bool unique() const throw();
  long use_count() const throw();

private:
  X *x_;
  size_t *refs_;
};
```

Listing 4

The reference count is pointed to by the member variable **refs_** and is incremented whenever a **shared_ptr** is copied (either through assignment or construction) and decremented whenever a **shared_ptr** is redirected (either through assignment or reset) or destroyed.

For example:

```
void
f()
{
  shared_ptr<T> t(new T); //*refs_==1

  {
    shared_ptr<T> u(t); //*refs_==2
    //...
  } //*refs_==1

  //...
} //*refs_==0, object is destroyed here
```

Reference counting blurs the distinction between object lifetime control and transfer by allowing many entities to simultaneously "own" an object.

```
shared_ptr<T>
f()
{
  return shared_ptr<T>(new T);
}

void
g(shared_ptr<T> t) //++*refs_
{
  //...
} //--*refs_

void
h()
{
  shared_ptr<T> t;
  t = f(); //ownership is transferred from f here
  g(t);    //ownership is shared with g here
}
```

The sequence of events in the above example runs as shown in Figure 1.

As you can see, at the end of **h**, the reference count is zero and the object is consequently destroyed.

Since the object has more than one owner we must exercise caution when using it or, more specifically, when changing its state.

For example:

```
void
f(shared_ptr<T> t)
{
  //...
}

void
g()
{
  shared_ptr<T> t(new T);
  shared_ptr<const T> u(t);
  f(t);    //ownership is shared with f here
           //state of u is uncertain here
  }
```

Of course, this is just pointer aliasing in a spiffy new suit and as such shouldn't come as much of a surprise. I mean, *nobody* makes that mistake these days, do they?

Well, almost nobody.

Well, certainly not standard library vendors.

```
call h
shared_ptr()

call f
new T
shared_ptr(T *)                        //*refs_=1
shared_ptr(const shared_ptr &)         //++*refs_
~shared_ptr                            //--*refs_
exit f

shared_ptr::operator=(const shared_ptr &)
                                       //++*refs_
~shared_ptr                            //--*refs_

call g
shared_ptr(const shared_ptr &)         //++*refs_
~shared_ptr                            //--*refs_
exit g

exit h
~shared_ptr                            //--*refs_
```

Figure 1

Well, probably not standard library vendors.

Well, probably not very often.

string

The last time I saw this problem was in a vendor supplied **std::string**. Well, not this problem exactly, but it was related to incorrect use of reference counted objects. I shan't name names, but it was a company you all know and many of you respect. When I finally tracked down what was causing my program to crash I was stunned.

Now you may be wondering how these sorts of problems could possibly relate to **string**, after all it's a value type not an object type. The reason is that this particular **string** used a common optimisation technique known as copy-on-write, or COW for short, and this technique relies upon reference counting.

Next time, we'll take a look at **string** and the implications of the COW optimisation. ■

#include

Krauss and Starkman. *Universal Limits on Computation* (arXiv:astro-ph/0404510 v2, 2004).

Clark, *2001: A Space Odyssey* (Hutchinson, 1968).

Acknowledgements

With thanks to Kevlin Henney for his review of this article and Astrid Osborn, Keith Garbutt and Niclas Sandstrom for proof reading it.