The magazine of the ACCU

www.accu.org

{cvu}

Volume 19 Issue 2 April 2007 £3

Features

{cvu}
Volume 19 Issue 2
April 2007
ISSN 1354-3164
www.accu.org

accu

Conference Time

Well, we are only a few months into the year, and I don't know about you lot but my New Year resolutions aren't going too well. For some reason even just eating less is a lot harder than it sounds.

As you all should be aware the ACCU spring conference is almost upon us. And if you don't live in Europe you might be reading this after it's happened already. I have to admit I love the conferences. I didn't always. The only reason I went to my first ACCU conference was to see Bjarne Stroustrup, as I didn't think there would be much chance of me seeing him back in New Zealand. The next year I went because I'd catch up with some colleagues who I had worked with. I didn't get much out of that conference. Then I didn't go for a few years. I just looked at the program and thought 'Nah, that doesn't sound that interesting. Maybe next year.' Then one year I had an epiphany. There is a saying that goes something like 'you become like those you hang around', and there were many people I respected and wanted to be more like who go to these ACCU conferences. So I decided that I'd go with the intent of interpersonal networking, and enjoying some interesting talks. It is the talking and association that goes on in between the sessions, at breakfast, at dinner, and at the bar, that you'll often get the most worth out of the conference. Yeah sure, some of the talks are interesting, and you'll learn a thing or two, but there is the immeasurable benefit that you get out of just 'hanging around' people who you aspire to be like.

On a slight tangent, for those of you, like me, living on the other side of the planet to Oxford, you'll be aware that the cost of getting to the conference is significantly more than the cost of entry. If we can get some more members close to us, there is no reason not to aim for a southern hemisphere ACCU spring conference. If there is enough interest I'm sure we can work something out, and maybe we will see Bjarne over here after all.

Go to the conference, but hang around between the sessions, and in the evening. Find out where people are going for dinner and ask to go along.

TIM PENHEY,
EDITOR

The official magazine of 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.

ACCU exists for programmers at all levels of experience, from students and trainees to experienced developers. As well as publishing magazines, we run a respected annual developers' conference, and provide targeted mentored developer projects.

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

To find out more about ACCU's activities, or to join the organisation and subscribe to this magazine, go to www.accu.org.

Membership costs are very low as this is a non-profit organisation.

DIALOGUE

REGULARS

FEATURES

COPY DATES

C Vu 19.3: 1st May 2007
C Vu 19.4: 1st July 2007

IN OVERLOAD

Begin your reading with 'Software Product Line Engineering with Feature Models' by Danilo Beuche and Mark Dalgarno.

ADVERTISE WITH US

COPYRIGHTS AND TRADE MARKS

A Custom Event Layer for the ACE Reactor Framework

Matthew Wilson and Garth Lancaster describe a mechanism for seamless insertion into event-based hierarchies.

Introduction

The Adaptive Communications Environment (ACE) [1] provides an operating-system adaptation layer for I/O, timers, signals and synchronisation based events, and offers several substantive frameworks for the development of high-performing networking programs in C++. The ACE Reactor framework operates on a call-back model [2], whereby an event handler – an instance of a class derived from **`ACE_Event_Handler`** – is registered with the reactor – an instance (usually a singleton) of a class derived from **`ACE_Reactor`** – and receives events about which it is interested via its overridden handler methods. Figure 1 shows the basic relationship between the main actors in the Reactor framework, and the methods primarily involved in the event handling.

One area in which the functionality offered by the framework is lacking is in support for rich 'application' events. For example, in a recent middleware project, our design called for abstractions of the various system events – both communications and logical – in the form of 'notifications', which are able to carry context information (see 'Rich events' section).

A limited form of such event mechanisms may be layered over the Reactor framework's timer support, and this can work well in simple cases. But using reactor timers leaves a lot to be desired in two important ways:

1. Supporting classes of events that may have multiple concurrent instances, e.g. an *IncomingUpstreamMessage* event, requires additional functionality to be layered over the existing timer event support.
2. The caller-supplied argument supported by ACE – of type **`void const*`** – facilitates the association of an object (pointer) with a timer event instance. This must be used to associate any identifying information with an event. But it will also be used by bona fide timers of a given reactor instance. It therefore forces an undue coupling between base and derived classes in event-handler hierarchies, since there has to be some mechanism to ensure that an arbitrary **`void*`** value used by the parent is not the same as an arbitrary **`void*`** used by the child. This is fragile at best.

In this article we describe the ACESTL **`custom_event_handler`** class, with which you can extend the ACE reactor framework with support for rich, multi-instance application events without affecting the normal reactor/timer event mechanism.

ACESTL's custom_event_handler

The **`custom_event_handler`** class – part of ACESTL, a sub-project of the STLSoft libraries [3] – is an abstract class that derives from **`ACE_Event_Handler`**, and defines the additional overrideable method **`handle_custom_event()`**. To use **`custom_event_handler`**, you derive your handler class from it rather than from **`ACE_Event_Handler`** directly, and provide an implementation of **`handle_custom_event()`**; all other handling can be carried out in exactly the same way it would in a regular event-handler scenario. Figure 2 illustrates the relationships between the various types; **`callback_hook`** and **`event_info`** are internal structures that we'll explain later.

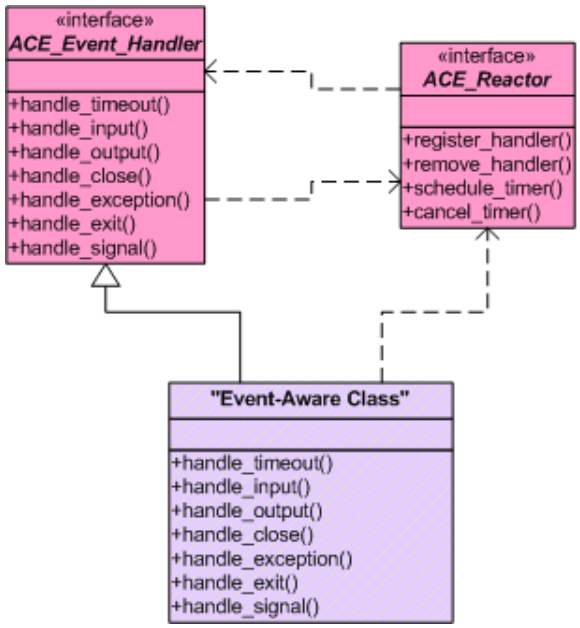


Figure 1

The public interface of **`custom_event_handler`** is shown in Listing 1. The constructor, **`protected`** because it is an abstract class, follows that of **`ACE_Event_Handler`** in taking an optional reactor instance and an optional priority for the event handler. This allows it to be plugged into an existing architecture with no other code changes. The seven public methods are for managing custom events, and will be explained shortly.

Custom events are scheduled for immediate or delayed actuation by the two **`schedule_custom_event()`** overloads. Each takes an event code **`(long)`** and an optional argument **`(void*)`**. The three-parameter overload also takes a delay parameter **`(ACE_Time_Value)`** that is the time offset from the time of call, rather than an absolute time. They each return a non-0 value (of the opaque type **`event_id`**) to indicate success, and which also identifies the particular event instance. The **`code`** and **`arg`** parameters, along with the returned event id are associated internally (via a call to **`ACE_Reactor::schedule_timer()`**) with a timer instance in the **`custom_event_handler`**'s reactor. Immediate events are processed subject to the reactor instance's next dispatch epoch.

MATTHEW WILSON

Matthew is a development consultant for Synesis Sofware and creator of the STLSoft libraries. He is the author of *Imperfect C++* and *Extended STL, volume 1* (both published by Addison-Wesley) and is working on his next book, *Breaking Up the Monolith*. He can be contacted via http://imperfectplusplus.com/

GARTH LANCASTER

Garth is the EDI/Automation Support Manager for MBF Australia Pty Ltd, and bashful owner of a bewildering spectrum of skills in software development, integration and management. He can be contacted via garth.lancaster@optusnet.com.au

Listing 1

```
class custom_event_handler
    : public ACE_Event_Handler
{
// Types
private:
  struct event_id_ {};

public:
  typedef ACE_Event_Handler
     parent_class_type;
  typedef custom_event_handler  class_type;
  typedef event_id_             *event_id;
  typedef void (*cancelled_event_code_fn)(
                void *param, long code,
                event_id id, void *arg);
// Construction
protected:
  explicit custom_event_handler(
    ACE_Reactor *reactor  =
       ACE_Reactor::instance(),
    int priority  =
       ACE_Event_Handler::LO_PRIORITY);
```

Listing 1 (cont'd)

```
public:
  ~custom_event_handler();
// Operations
public:
  event_id schedule_custom_event(long code,
     ACE_Time_Value const &delay, void *arg = 0);
  event_id schedule_custom_event(long code,
     void *arg = 0);
  int cancel_custom_event(event_id event,
     void **parg = NULL);
  int cancel_custom_events(long code);
  int cancel_custom_events(long code,
     cancelled_event_code_fn pfn, void *param);
// Attributes
public:
  int has_custom_event(long code) const;
  int has_custom_event(event_id event) const;
// Overrides
private:
  // The callback function, to be implemented
  // by derived classes
  virtual int handle_custom_event(
     ACE_Time_Value const  &current_time
   , long                  event_code
   , void                  *arg) = 0;
. . .
};
```

Individual events may be cancelled by calling **cancel_custom_event()** and passing the **event_id** value returned from **schedule_custom_event()**; you can optionally pass the address of a variable to receive the **arg** associated with the event. Alternatively, all the events of a particular event code may be cancelled by calling **cancel_custom_events()** and passing in the event code. The second overload takes a callback function and caller-supplied parameter (to be passed back in each invocation of the callback function) that allows the caller to retrieve the **arg** parameters for all corresponding event instances before they are cancelled (See 'Cleanup Sink Member Function Template'). (There are two overloads, rather than using default parameters, because the callback function and parameter are a logical pair; passing a function but defaulting the parameter would, in most cases, be a programming error.) The cancel methods indicate failure according to the standard ACE idiom by returning –1. (All the methods follow the ACE convention of returning -1 rather than throwing exceptions.) Cancelling events that do not exist (or have already been actuated) is benign.

A **custom_event_handler** instance is able to give information regarding its pending events, via the **has_custom_event()** overloads, which take either the id of an event instance (**event_id**) or the event code of a group of events (**long**). They both return the number of pending events that match the given parameter; since event ids are unique within a given **custom_event_handler** instance, the **event_id** overload returns either 0 or 1.

The remaining part of the interface is the **handle_custom_event()** pure virtual method. Derived classes must override this method, which is then called when a custom event is actuated, passing the original event code and argument, and the current time. It is declared **private** to prevent any unwanted calls from outside; this does not in any way affect the ability of derived classes to override it, though they may elect to declare their overrides **protected** if they want to chain them in a rich hierarchy.

Implementation

The first idea was simply to have each **custom_event_handler** instance register timers for the events, with itself as the handler instance. However, there are a couple of flaws in this logic, because derived classes may still need to register their own timers. First, it would require the

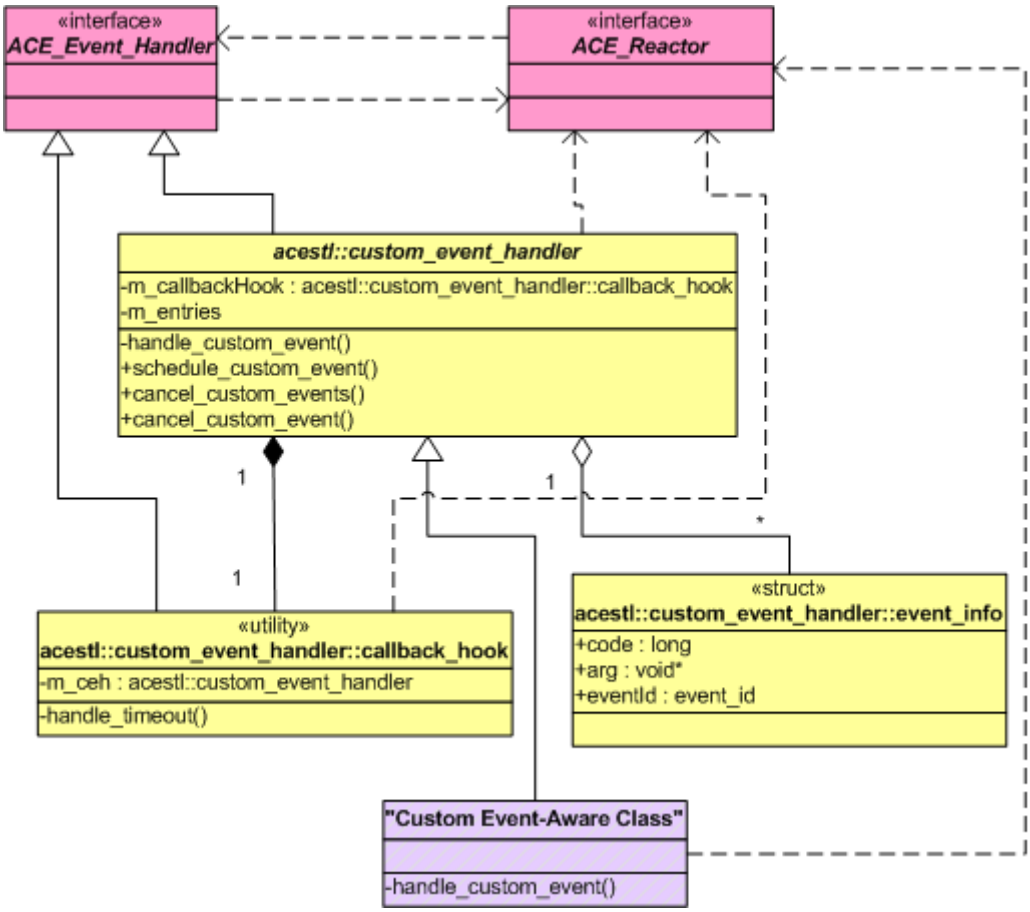


Figure 2

Cleanup sink member function template

Having to define a static / non-member function specifically to translate the **void*** parameter back into a class instance pointer can be onerous, particularly when it has to repeated for every type that wishes to receive the notifications:

```
class CancelHandler
{
public:
  void report(long code, event_id id, void *arg)
  // This method would have to be provided in
  // every cancel handler
  static void report_proc(void *param, long code,
     event_id id, void *arg)
  {
    static_cast<CancelHandler*>(
       param)->report(code, id, arg);
  }
};
  . . .
CancelHandler  cc;
mn->cancel_custom_events(101,
   &CancelHandler::report_proc, &cc);
```

Listing 2

```
class custom_event_handler
{
  . . .
// Implementation
private:
  class callback_hook
    : public ACE_Event_Handler
  {
  public: // Construction
    callback_hook(custom_event_handler *ceh,
       ACE_Reactor *reactor, int priority)
       : ACE_Event_Handler(reactor, priority)
       , m_ceh(ceh)
    {
      ACESTL_MESSAGE_ASSERT(
         "reactor may not be null",
         NULL != reactor);
    }
    ~callback_hook()
    {
      reactor()->remove_handler(this
         , ACE_Event_Handler::ALL_EVENTS_MASK |
         ACE_Event_Handler::DONT_CALL);
    }
  private: // Overrides
    virtual int handle_timeout(
       ACE_Time_Value const    &current_time
       ,   const void              *arg)
    {
      return m_ceh->handle_callback_timeout(
         current_time, const_cast<void*>(arg));
    }
  private: // Members
    custom_event_handler *const m_ceh;
    // Back ptr to owner
  private: // Copy proscription
    callback_hook(callback_hook const &);
    callback_hook &operator =(
       callback_hook const &);
  };
// Members
private:
  . . .
  callback_hook   m_callbackHook;
};
```

Cleanup sink template (cont'd)

This functionality is abstracted into the form of a member function template of **custom_event_handler**, which overloads **cancel_custom_events()**, as follows:

```
class custom_event_handler
{
  . . .
public:
    int cancel_custom_events(long code
          , cancelled_event_code_fn pfn
          , void *param);
  template<typename C>
  struct cancel_adapter
  {
    typedef cancel_adapter<C> class_type;
    cancel_adapter(C *obj,
       void (C::*pfn)(long, event_id, void *))
       : m_obj(obj), m_pfn(pfn)
    {}
    static void proc( void *param, long code
                   , event_id id, void *arg)
    {
      class_type *pThis =
         static_cast<class_type*>(param);
      ((pThis->m_obj)->*(pThis->m_pfn))(code,
         id, arg);
    }
  private:
    C *const m_obj;
    void (C::*m_pfn)(long, event_id, void *);
  };
  template<typename C>
  int cancel_custom_events(long code, C *obj
     , void (C::*pfn)(long, event_id, void *))
  {
    cancel_adapter<C> adapter(obj, pfn);
    return this->cancel_custom_events(code
       , &cancel_adapter<C>::proc, &adapter);
  }
```

Granted this is a bit to swallow at first glance, but it facilitates the following simplification for any and all client types.

```
class CancelHandler
{
public:
  void report(long code, event_id id, void *arg);
};
  . . .
CancelHandler  cc;
mn->cancel_custom_events(101, &cc,
    &CancelHandler::report);
```

authors of derived classes to pass on unrecognised timer events to the parent classes, something that is at once too easy to forget (or to lose in maintenance) and also not part of established idiom in ACE event handler classes [1].

Second, there is the possibility of clashes in the interpretation of the **handle_timeout()** method. Because the custom event handler must handle multiple event instances for a given event code, it maintains internal associations between code and instances, as we'll see shortly. It passes the address of information structures to the ACE timer infrastructure as the **arg** parameter in **ACE_Reactor::schedule_timer()**, and then (re-) interprets this on behalf of derived classes into the *real* **arg** parameter passed to **custom_event_handler::schedule_custom_event()**. Hence, with the original design it would have been possible that, for a given **custom_event_handler** instance, an **arg** given to a bona fide **schedule_timer()** call coincided with the address of the

Listing 3

```
class custom_event_handler
    : public ACE_Event_Handler
{
  . . .
// Implementation
private:
  int handle_callback_timeout(
     ACE_Time_Value const  &current_time
     , void                *arg);
  . . .
// Members
private:
  struct event_info
  {
    long      code;     // event code
    void      *arg;     // custom event argument
    event_id  eventId;  // event id
  };
  typedef stlsoft::shared_ptr<event_info>
     info_ptr;
  typedef std::map<event_id, info_ptr>
     event_map_type;
  typedef std::map<long, event_map_type>
     event_code_map_type;
  callback_hook       m_callbackHook;
  event_code_map_type m_entries;
};
```

internal information structure, which could result in premature firing of the custom event and tardy delivery of the scheduled timer, or vice versa. It's plausible that this might cause worse issues, i.e. crashes.

The answer to these significant drawbacks, therefore, is to insulate the relationship between the reactor and the **custom_event_handler** instance in a separate handler, in the form of a **private** member variable of the nested class **callback_hook**, which also derives from **ACE_Event_Handler** (as shown in Listing 2).

The **callback_hook** instance is associated with its containing **custom_event_handler** instance in the latter's constructor, as in:

```
custom_event_handler::custom_event_handler(
   ACE_Reactor *reactor, int priority)
   : parent_class_type(reactor, priority)
   , m_callbackHook(this, reactor, priority)
{}
```

Listing 4

```
int
custom_event_handler::handle_callback_timeout(
   ACE_Time_Value const &current_time, void *arg)
{
  event_info const *entry  =
     static_cast<event_info const*>(arg);
  event_code_map_type::iterator itc =
     m_entries.find(entry->code);
  ACESTL_ASSERT(m_entries.end() != itc);
  event_map_type &event_map  = (*itc).second;
  event_map_type::iterator ite =
     event_map.find(entry->id);
  ACESTL_ASSERT(event_map.end() != ite);
  // Keep entry alive, until call completes
  info_ptr  ep = (*ite).second;
  event_map.erase(ite);   // erase event instance
  if(event_map.empty())
  {
    m_entries.erase(itc); // erase event code
  }
  return this->handle_custom_event(current_time,
     entry->code, entry->arg);
}
```

The events are managed inside **custom_event_handler** in the form of a map of maps (listing 3)

The inner map, of type **event_map_type**, associates an event Id with a (smart) pointer to an **event_info** structure. The outer map, of type **event_code_map_type**, associates an event code with an **event_map_type** instance. Thus, event codes map to collections of events, within each of which event Ids map to event information.

Hence, the caller-supplied event argument **arg** is maintained within the **event_info** data structure managed by **custom_event_handler**. The actual **arg** passed to **schedule_timer()** is the address of the **event_info** instance for the given event. When the callback from the ACE reactor is received by **m_callbackHook** (see Listing 2), it is passed to **custom_event_handler**'s private **handle_callback_timeout()** method (Listing 4). The **arg** is translated back into an **event_info** pointer that is used to lookup the event information, before delivering the event in a call back to the derived class via **handle_custom_event()**.

Examples

Listing 5 is an example custom event handler class that prints even information to standard output. Listing 6 is a simple test program that demonstrates scheduling and cancelling events, including cancelling individual instances and cancelling all instances of a given code.

When executed, this program produces the following output:

```
Cancelled: param: 00000000; code: 101; id: 4; arg: 1002
handle custom event: code: 100; arg: 1000; t: 0
handle custom event: code: 102; arg: 1003; t: 0
handle custom event: code: 100; arg: 1001; t: 1000
handle custom event: code: 102; arg: 1005; t: 2109
```

Rich events

The recent networking middleware projects we mentioned earlier required a rich event model, whereby multiple instances of each event type, optionally carrying additional information, could be sent to recipients throughout the system. These events, known as Notifications, were generated in response to I/O events – e.g. incoming bytes generating an *InputUpstreamByteQueue* notification – as well as used to manipulate the logical state of the system – e.g. the supervisory component sending an *ActivateChannel* notification to (re)start channels. This was readily

Listing 5

```
class MyNotification
    : public
{
public:
  MyNotification(ACE_Reactor *reactor)
    : acestl::custom_event_handler(reactor)
    , m_start(ACE_OS::gettimeofday())
  {}
private:
  virtual int handle_custom_event(
     ACE_Time_Value const  &current_time
     , long event_code, void *arg)
  {
    const ACE_Time_Value delta =
       current_time - m_start;
    std::cout << "handle custom event: code: "
              << event_code << "; arg: "
              << "; t: " << (delta.sec() * 1000
              + delta.usec() / 1000)
              << arg << std::endl;
    return 0;
  }
private:
  const ACE_Time_Value  m_start;
};
```

Listing 6

```
static void cancel_proc(void *param, long code,
   event_id id, void *arg)
{
  std::cout << "Cancelled: param: " << param
          << "; code: " << code << "; id: "
          << reinterpret_cast<unsigned long>(id)
          << "; arg: "
          << reinterpret_cast<unsigned long>(arg)
          << std::endl;
}

int main()
{
  event_id id1, id2, id3, id4, id5, id6;
  MyNotification  *mn = new MyNotification(
     ACE_Reactor::instance());
  assert(0 == mn->has_custom_event(100));
  assert(0 == mn->has_custom_event(101));
  assert(0 == mn->has_custom_event(102));

  id1 = mn->schedule_custom_event(
     100, (void*)1000);
  id2 = mn->schedule_custom_event(
     100, ACE_Time_Value(1), (void*)1001);
  assert(0 != mn->has_custom_event(100));
  assert(0 == mn->has_custom_event(101));
  assert(0 == mn->has_custom_event(102));

  id3 = mn->schedule_custom_event(
     101, (void*)1002);
  assert(0 != mn->has_custom_event(100));
  assert(0 != mn->has_custom_event(101));
  assert(0 == mn->has_custom_event(102));

  id4 = mn->schedule_custom_event(
     102, (void*)1003);
  id5 = mn->schedule_custom_event(
     102, ACE_Time_Value(2), (void*)1004);
  id6 = mn->schedule_custom_event(
     102, ACE_Time_Value(2, 100000),
     void*)1005);
  assert(0 != mn->has_custom_event(100));
  assert(0 != mn->has_custom_event(101));
  assert(0 != mn->has_custom_event(102));

  mn->cancel_custom_event(id5);
  mn->cancel_custom_events(
     101, cancel_proc, NULL);
  assert(0 != mn->has_custom_event(100));
  assert(0 == mn->has_custom_event(101));
  assert(0 != mn->has_custom_event(102));

  ACE_Reactor
     ::instance()->run_reactor_event_loop();
}
```

achieved using the event support of **acestl::custom_event_handler.**

Each recipient derives from the abstract class **NotificationHandler** (Listing 7), which itself derives from **acestl::custom_event_handler**. **NotificationHandler** insulates derived classes from the **void*** event arguments of **custom_event_handler**, instead manipulating notification instances via the **INotification** reference-counted interface (Listing 8). **INotification** defines methods for retrieving notification-specific information from an instance, and also supports the notification chaining. Because the notification interface is reference-counted, we were able to cache common stateless notifications – known as 'Stock Notifications' –

Listing 7

```
class NotificationHandler
  : public acestl::custom_event_handler
{
public:
  . . .
  void PostNotification(
     INotification *notification);
  void PostNotification(NotificationId id);
  void PostNotification(Time const &delay,
     INotification *notification);
  void PostNotification(Time const &delay,
     NotificationId id);
  bool CancelNotifications(NotificationId id);
  int CancelAllNotifications();
  bool HasNotification(NotificationId id);
private:
  virtual int HandleNotification(
     NotificationId  id
     , INotification         *notification
     , NotificationHandler   *sender) = 0;
  virtual int handle_custom_event(
     ACE_Time_Value const  &current_time
     , long event_code, void *arg);
  . . .
```

which meant that there was very little memory allocation and object creation associated with the notification layer, even in very heavily loaded servers.

This abstraction (see Figure 3) afforded us the ability to implement our servers in an entirely event-based manner. For example, in response to *InputUpstreamByteQueue* notification, the channel would invoke the message parser on the contents of the input upstream byte queue. If a complete and correctly formed message payload was contained therein, an instance of the system protocol message would be created and enqueued, and an *IncomingUpstreamMessageQueue* notification would be generated and dispatched by the channel to itself. Conversely, if the channel contained invalid data, a *ChannelErrorState* notification would be dispatched to the server supervisor, which would log the situation and issue *ResetChannel* and *ActivateChannel* notifications to tear down and then re-establish the channel's connections to its network peers.

Being based on **custom_event_handler**, the amount of code in this notification layer is pleasingly small, and, once developed for our initial requirements, was readily utilised to rapidly develop other server components in our system as our requirements evolved (as requirements are wont to do).

Summary

In summary, ACESTL's **custom_event_handler**:

- Provides a simple event interface, suitable for expansion by application-specific custom event functionality.

Listing 8

```
class INotification
  : public IRefCounter
{
public:
  virtual NotificationId  id() const = 0;
  virtual RC LookupValue(char const *name, long
&value) const = 0;
  virtual RC LookupValue(char const *name, void
*&value) const = 0;
  virtual RC LookupValue(char const *name,
string_t &value) const = 0;
  virtual RC LookupValue(char const *name,
INotification *&value) const = 0;
};
```

- Can be plugged into any existing `ACE_Event_Handler` hierarchies with no disruption whatsoever to existing functionality, and requires only the overriding of one new (pure) virtual function.
- Enables rich-object base notification frameworks to be built with ease.
- Avoids fragile interruption of application-oriented virtual function overloads.
- Behaves like a good ACE citizen, respecting conventions for return codes, exceptions, memory, etc.

`custom_event_handler` is part of ACESTL, the STLSoft sub-project for extending the Adaptive Communications Environment. ■

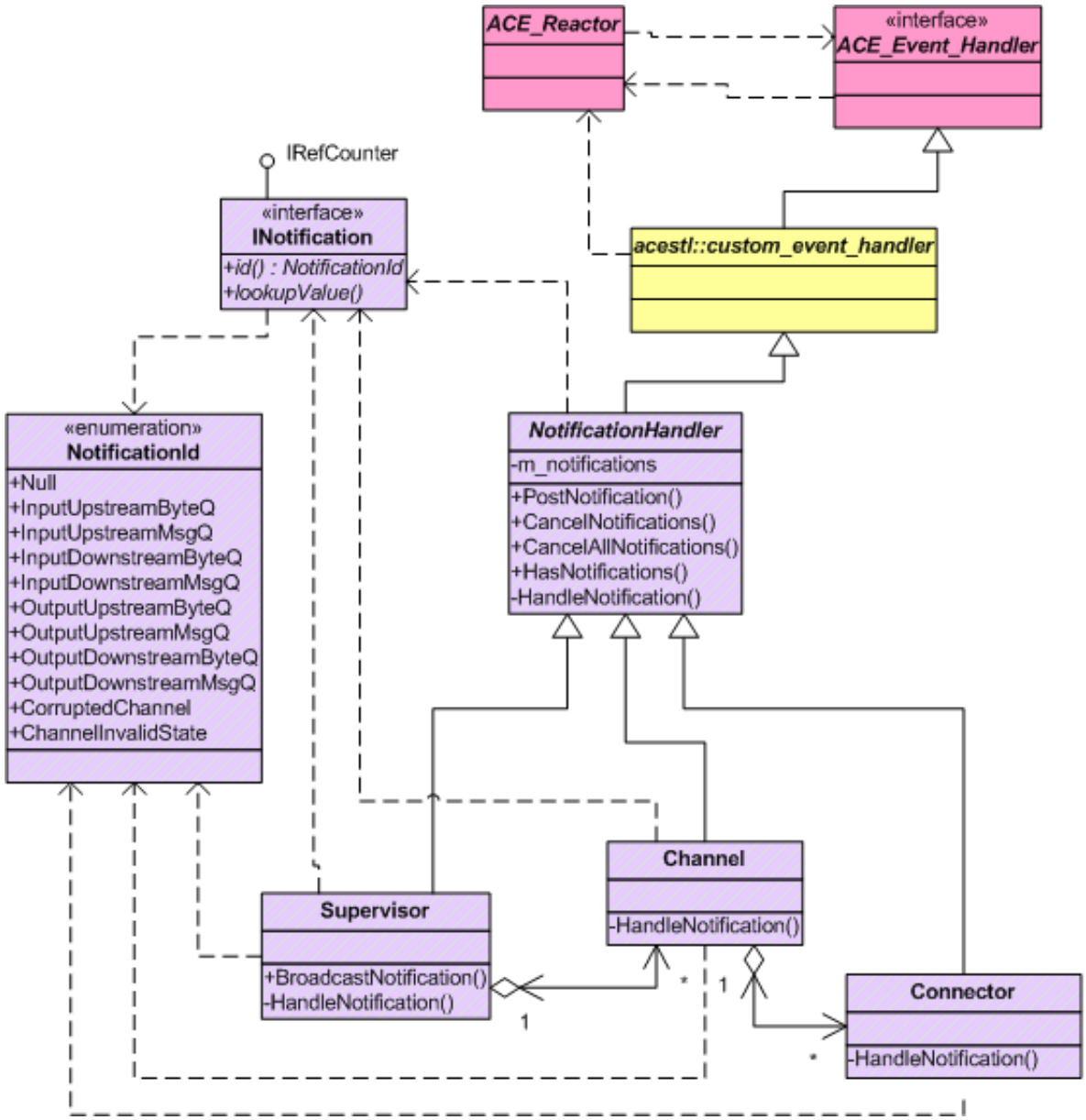
Acknowledgements

Thanks to Bjorn Karlsson, Kevlin Henney, Nevin Liber, Pablo Aguilar and Walter Bright for reviewing the article, and helping us keep to the point.

Notes and References

1. The Adaptive Communication Environment project is located at http://www.cs.wustl.edu/~schmidt/ACE.html.
2. *C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks*, Douglas C. Schmidt & Stephen D. Huston, Addison-Wesley, 2002
3. STLSoft is an open-source organisation whose focus is the development of robust, lightweight, simple-to-use, cross-platform STL-compatible software, and is located at http://stlsoft.org/.

Figure 3



Playing by the Rules

Pete Goodliffe describes the rules that help him to play the programming game.

Incarcerated as I am in a software factory, I have to spend my days with other programmers bashing out lines of carefully honed C++ code. To make this bearable we've developed a system to help us get the job done.

Well, you've got to have a system.

Sure, we have methods and methodologies. But sometimes that's not enough. We program in C++, using object oriented techniques, 'modern' C++ idioms, and the kind of flair and style that would put Huggie Bear to shame if he now worked in the software business. We employ agile programming, (*we're XP, daaaaaahling* – it sounds so very fashionable). We're test driven, pair programmed, and continuously integrated up to our eyeballs. All well and good.

Those are all basically just a set of rules – rules that we use to play the programming game together: to describe how our team interacts, who does what, and how to tell when we've won. (Just don't try to describe the C++ *off-side rule* to a non-programmer.) These rules actually *define* the programming game we play.

But sometimes all of these rules, good as they are, aren't enough. Sometimes the poor programmers need *more* rules. Really, we do. We need rules that we've *made ourselves*. Rules that we can take ownership of. Rules that define the culture and style of development in our particular team. These needn't be large unwieldy draconian edicts. Just something simple you can give a new team member so that they can immediately play with you. These are rules that describe something more than mere methods and processes; they are rules that describe a coding culture – how to be a good sportsman.

Sound sane? Well, we think so. Our team's Tao of development is summed up in three short complementary statements. These statements are now enshrined in our folklore, have been printed out in large, friendly letters, and emblazon our communal work area. They reign over all we do; whenever we face a choice, a tricky decision, or a heated discussion, they help to guide us to the right answer.

Are you ready to receive our wisdom? Brace yourself. Our three earth-shattering rules for writing good code are:

1. Keep it simple
2. Use your brain
3. Nothing is set in stone

That's it. Aren't they great?

I could end the article here and leave the exegesis as an exercise for the reader. But that would be cruel. I like to be succinct, but not cruel. So let's take a short stroll through the land surrounding these rules, see what they mean to my team, and whether we can learn something from them to help us write good code in the Real World.

1. Keep it simple

That sounds like an innocuous first rule, doesn't it? And it sounds quite easy to follow, too. Ah! How appearances can be deceptive...

There are two kinds of simplicity: the *wrong* sort and the *right* sort. Here 'simplicity' specifically does *not* mean: write code the easiest way you can, cut corners, ignore the complicated stuff (and hope it goes away), and generally be a programming simpleton. Oh, if only it meant that. Too many programmers out there in the Real World *do* write 'simple' code like this. Some of them don't even realise that they're doing it; they just don't think enough about the code they're writing (and how things might go wrong in it) to notice. Simplicity is never an excuse for incorrect code.

Invalid 'simplifying' assumptions are easy to make, and (whilst they can lead to less complexity in your head) inevitably lead to bugs. We must always write robust, correct code. Each function must do exactly what's it supposed to. To the letter. Nothing less. And nothing more... *that's* simplicity. But I'm getting ahead of myself.

Instead of this wrong simple-minded 'simplicity' we strive to write the *simplest* code possible. This is very different from disengaging your brain are writing stupid code (see rule 2). It is actually a very brain intensive pursuit – ironically, it's hard to write something simple. It's far easier to let your code grow into that spaghetti-like ball of complexity [1].

If simplicity is our goal, what does it look like? Perhaps it's easiest to describe in terms of its counterpart: simple code is not *complicated*. Complication comes in a number of guises. For example, unnecessary tight coupling between code modules, too much code, verbose code, complicated control flow, too much indirection [2], too many threads (more than one?) running around the same code, and more, and more...

Unfortunately, we need to care about far more than just simplicity at the *code* level. We must continually strive for simplicity at many levels, across the whole software development endeavour. For example, strive for:

- Simplicity in **design**

  Think again about your component coupling, check the number of components in your design, and the roles of those components. The number of components and the nature of their plumbing should be *appropriate*. Big problems with many parts may require a large number of components. Don't be afraid of this. Break things up if you need to – if they need to be broken. But don't split things into a million components when it's not necessary.

  Simple designs appear elegant and cohesive. Because of their simplicity, they can be quickly and clearly described, and easily understood. Simple designs are easy to visualise.

- Simplicity in lines of **code**

  We started to scope this out already, but there's an awful lot to consider if you want to achieve true code simplicity. Sadly, a lot of

PETE GOODLIFFE

Pete Goodliffe is a programmer who never stays at the same place in the software food chain. He has a passion for curry and doesn't wear shoes. Pete can be contacted at pete@cthree.org

it is subjective – down to a personal preferences and your sense of aesthetics.

- Simplicity in a **bug fix**

  Ensure that you always fix a bug *correctly*; in the simplest way, not with a mammoth workaround, or a superficial patch for a symptom that leaves the 'fix site' in a mess and the root problem unaddressed. This is really a special case of...

- Simplicity in **maintenance**

  First, consider whether your code is simple enough to be easily maintained. Is it readable, comprehensible, modifiable – *simple*? Then consider when you maintain a section of code whether you are putting in the extra effort to ensure that your modification leaves the code at least *as* maintainable, if not more so. Are you tacking more stuff onto an existing tottering pile of flaky software, or are you refining the internal code structure to create something better (and, hopefully, *more* simple)?

- Simplicity in **process** and **documentation**

  Are you developing in a simple, lightweight, manner without unnecessary layers of bureaucracy? Or do you have countless forms, procedures, meetings, infeasibly large and incomprehensible Gantt charts, sign-offs, and other unnecessary software voodoo dances?

  Do you create documentation for all code that you write? Or is the code simple enough to be its own clear documentation? Do you have to write specifications for all work? Design proposals? Literate code comments? Are any of these forming a duplication of information? Duplication is the enemy of simplicity.

Simplicity is necessarily allied with *sufficiency*. You should work in the simplest way possible and write the simplest code possible. But no more. If you over-simplify, you will not be solving the actual problem. Our 'simple' solutions *must* be 'sufficient' or they are *not* solutions. But how do you know what the right level of simplicity is?

I've only just scratched the surface of this topic, and simplicity is already looking like pretty hard work. Phew. But be prepared: we'll revisit this theme in various guises as we look at rules 2 and 3.

2. Use your brain

Like many ancient texts, scholars could examine, re-examine, interpret, re-interpret and try to intuit what this rule means and still not have agreed upon an answer by the second coming. The problem is that different programmers' brains work in different ways. What seems obvious to one person may not to another. That's life.

But, given this rule's blatant ambiguity and openness to misinterpretation, it is still really *very* important. It's far too easy to program without engaging your brain. Really. It's easy to just follow your fingers as they bash out lines of code. It's easy to get stuck in a rut trying to solve (what you think is) the immediate problem, without really considering the bigger picture, thinking about the code that's around you, or even whether what you're typing is the right thing. Let's admit it, we've all done it – programming on auto-pilot.

So this rules tells us to *stop* and think about what we're doing. Whilst that might seem obvious, there are many subtle ways to program without brain usage. For example, it's easy to favour one design approach over another just because you understand it best, and then plough onwards when it's not appropriate.

Use your brain is an empowering rule, too. You are *allowed* to use your brain. If we both don't know an answer, then your brain is as good as mine – *you* can work it out. No one programmer *owns* any piece of code; someone else might have started a component, but you can extend it and make design decisions about it as well as anyone else (because, of course, it's nice simple code – isn't it?)

Now sometimes following this rule involves an uncomfortable phenomenon known as *common sense*. In general, programmers seem curiously devoid of common sense. This is where forcible use of the brain is useful. We must be constantly alert to check that we're doing the 'right thing'.

This rule helps when:

- You are performing code design. You must ensure that you thoughtfully pick the most appropriate approach. What should I do when faced with several decisions? Don't blindly pick one; thoughtfully chose the best alternative. And don't make that single decision, forget it, and then plough onwards. Keep using your brain in the aftermath; whilst you work through its ramifications. Check that your decision still makes sense. If it doesn't then use your brain, admit that you got it wrong, and back it out.
- You don't know an answer – don't expect someone else to think for you. Use your own brain. If you really don't know where to start with a problem, though, don't waste time flapping about. Use your brain. The right thing to do then *is* to ask others which direction you should be headed in. How do you know when to ask, and when to work it out for yourself? Use your brain!
- You are about to check some code into source control. Stop! Have you actually remembered to build it after that last modification? *Does* it compile? Did you remember to test it? Does it work? Do all the unit tests pass? Did you add new unit tests? Use your brain: don't check in code that doesn't work. Don't break things. Don't make things worse.
- When you want to do the simplest thing, but you don't know what the simplest thing is. Sometimes it's not always obvious. But use your brain, work it out. And *then* use your brain – it's a good idea to run your plan past someone. Do they think it makes sense, too?
- When you need to establish some new development processes and practices, should you copy what you've heard works for other people, or adopt the latest fashionable methodology? No! Use your brain. Adopt the things that will work for you. Select processes with just the right amount of ceremony. Avoid procedural duplication wherever you can.

3. Nothing is set in stone

There is a strange fiction prevalent in programming circles: once you've written some code then it is sacred. It should not be changed. Ever. Somewhere along the development line, perhaps at the first check-in, or perhaps just after a product release, the code is embalmed. It moves league. It is promoted. No longer riff-raff, it's digital royalty. The once questionable design is suddenly considered above (or at least beyond) question and becomes unchangeable. The internal code structure is no longer to be messed around with. All of its interfaces to the outside world are hallowed and mustn't be changed.

Why *do* programmers think like this? Fear. There is a very real apprehension of changing code that you don't know entirely. If you don't understand it from the inside-out, forwards and backwards, if you're not entirely sure what you're doing, if you don't understand every possible consequence of a change, then you could break the program in strange ways or alter odd corner case behaviour and introduce very subtle bugs into the product. You don't want to do that, do you? So *don't* tinker with the code. Better safe than sorry.

That is pure nonsense. Code should *never* stay still. No code is sacred. No code is ever perfect. How could it be? The world is constantly changing around it. Requirements are always in a state of flux. Product version 2.4

is so radically different from version 1.6 that it's entirely possible the internal code structure *should* be totally different. And we're always finding new bugs in our old code. If the code ever becomes a straight jacket then you are fighting the software, *not* developing it. Plenty of people know exactly what this feels like. And it's not nice.

Of course, it is perfectly correct to fear breaking the code. Only a fool would happily make changes without actually knowing what they're doing. So how do we reconcile these two stances? Although it sounds like a nightmare – who could possibly work with code that is constantly changing – there are a few key principles we adopt that really do make this work. Some are practical code writing concerns, some are procedural issues, and some are cultural. All are important:

- First of all, we all *want* to improve our code. We all agree that it is important. We all agree that it is necessary if we intend to build the best products possible. And we all recognise that it is a continual process.
- *All* of the code that we write is pair programmed, or peer reviewed. We don't let any code into the source repository unless it's been past two sets of eyes, and so we can be confident that it is the best code that we can currently write.
- *Pay attention now! Here's the really important bit: All* code is *fully* unit tested. These unit tests are not an optional extra. They are an integral part of our build process. Running them is not optional. It is not even an extra step after building; you can't actually create a final executable in our build environment unless all the unit tests pass first [3]. Harsh, but fair.
- These unit tests are thorough, and correct. We know this because they've been pair programmed or code reviewed. The unit tests are absolutely *essential* to our development. They are our safety harness. We have been able to make quite considerable refactors safe in the knowledge that we haven't broken anything, because the tests tell us if we do.
- We go to great lengths to write code that's easy to change. It's easy to change because we always write *simple* code. Because we *used our brains* to make the code clearly structured, and painless to maintain. We write code that is actually *designed* to be changed. We write code that doesn't pretend to be perfect as it first arrives.
- We adopt a common uniform coding style. We don't have an enormous 'code style guide' that says where every space and bracket should go, but we do have a complete naming convention (covering capitalisation, namespacing and the like) and a small set of clear guidelines for laying out code. That makes our code very easy to read and very easy to alter.
- We don't write code that contains subtle side effects, or is brittle in the face of change. As soon as we see code that is like this, we will refactor it mercilessly until it isn't.
- Fixing wrong, dangerous, bad, duplicated, or distasteful, code is positively encouraged. In fact, it is expected. We don't want to leave weak spots lying around for too long. If you find code that is too scary to change, then it *must* be changed!
- Refactoring is encouraged. If you have a job that requires a fundamental code change to get done properly then do it properly, do the refactor.
- No one 'owns' an area of code. Anyone is allowed to make changes in any area.
- It is not a crime to write the wrong thing. If someone changes your code it is not a sign that you are weak, or that another programmer is better than you. You'll probably tinker with their work tomorrow. That's just the way it works.
- No one's opinion is considered more important than anyone else's. Everyone has a valid contribution to make in any part of the codebase.
- We reduce coupling and make all units of code as simple as possible, with very clearly defined interfaces. This way, all the code makes sense in isolation. Reduced coupling means that rebuilds do not take forever, so the development cycle is short and feedback is quick.
- This is backed up with *continuous integration* – a server continually checks out and builds the code. If – heaven forbid – anything bad slips through our process and breaks the build, we will find out about it quickly. If there is a breakage then it's everyone's responsibility to fix it. That's our final safety net.
- To work on our codebase you are expected to have nerves of steel and not mind the ground changing underneath you. The code changes quickly, get used to it.

Naturally, we pick our battles. We can't possibly change all of the code all of the time, and add more code to it at the same time. There is a certain amount of *technical debt* that we live with until we get a chance to make appropriate changes. But we factor that into our system. Debt becomes work items that we plan onto our development roadmap, rather than forget and leave to fester.

So you can see how we design change into the code; we *expect* nothing to be set in stone. We do this is by adopting a certain coding style, a certain development process, and a certain mindset. And it works. The code is constantly changing. And each time we change it, it gets better – closer to how it needs to be *right now*. But who knows, in a few months it might look very different.

So there it is: the last of our three rules. Well, it's the last one for the moment. Nothing is set in stone.

Conclusion

What do these three rules prove? Nothing. But it's good to have a system. And it works for us. Sure, this isn't radical stuff, and it isn't going to enter the cannon of divinely inspired software development wisdom. We haven't yet set the programming community alight. But we have written a lot of good code, quickly.

These three rules describe *our* particular approach to writing software, and as such a short summary they are actually very powerful. You can see how our team organises code and collaboration around them. With them we've defined our unique coding culture. They are ideals that we own, collectively. We mutually enforce them. And they help us to write better software. Together.

You can't knock that.

Consider now how your team can take a greater collective ownership of the code you create, and of the process you employ to create it. How can you define your own, more healthy, programming culture? ■

Endnotes

1. But, of course, spaghetti code is far harder to maintain or fix. So is this really *easier*?
2. Remember the famous programmers' maxim: *every problem can be solved by adding an extra level of indirection*? Many complex problems can be subtly masked – and even caused by – unnecessary levels of indirection hiding the problem. Beware! Especially if reason for the existence of a level of indirection is not entirely clear.
3. *But*, you say, *you could just comment out the tests to get an executable.* And you can. But you can't check in the code like that. Try slipping that one past a peer, or a code review. Our process encourages – in fact it actually enforces – well-tested malleable code. And we like it like that.

A Guided Tour of the POCO C++ Libraries

Günter Obiltschnig explains the origins of POCO.

History and introduction

It all started in the early summer of 2004. Having just quit my job, I prepared myself to do some consulting work and also thought about writing a book. I spent the previous six years doing C++ development on a number of different platforms (Windows, various Unixes and OpenVMS – giving the state of C++ compilers on the various platforms back then, this was quite an adventure), so I wanted to write down my experiences in developing portable C++ applications. The best way to do this, I thought, was to guide the reader of my book through the creation of a C++ class library for cross-platform development. I had already written such a class library in my previous job, so I had a long list of things that I wanted to do 'right' should I ever get a second chance to design and write a cross-platform class library. Well, the book gave me the rare second chance, so, having enough time at my disposal, I started to work on my class library – after all, first you need something to write about.

The first parts I developed were very basic – macros for identifying the underlying operating system and C++ compiler, fixed-size integer types and dealing with different byte orders. Next I delved into I/O streams and wrote some template classes that made implementing my own stream classes easier. A bunch of stream classes were next, including streams for base64 encoding/decoding and for data compression/decompression using zlib. Every class library for cross-platform development needs multithreading abstractions, so these came next. By that time it became clear to me that the resulting class library would be much more than just a small abstraction layer for teaching how to write portable code. I was then (and still am) a big fan of C++, but it was with some envy that I looked at Java and C# developers and the comprehensive class libraries they had at their disposal. I wanted something similar for C++. Sure, there were other class libraries around, like Boost and ACE. However, ACE, to me, seemed somewhat stuck in the 90's, and Boost had lots of stuff in it that I did not have a use for and none of the stuff I desperately wanted – networking and XML, for example. Something had to be done. It was around October 2004, when POCO – the C++ Portable Components, were thus born. From the beginning I planned to make the libraries available under an open source licence. The first public release of POCO (0.91.1) was posted to Sourceforge on February 21, 2005. With the release, I also made some announcements at various internet forums and mailing lists, and people actually started downloading my library. Sure, there were not that many downloads at first, but still enough to keep me motivated. I also received some encouraging feedback from users, and some even offered to help.

Fast forward to early 2007. POCO is now at release 1.3, containing well over 550 classes, a community website for POCO users and developers is available [1], and POCO is being used in various commercial and open source projects. So, let's have a look at what's inside the POCO libraries.

POCO consists of four core libraries, and a number of add-on libraries. The core libraries are Foundation, XML, Util and Net. Two of the add-on libraries are NetSSL, providing SSL support for the network classes in the Net library, and Data, a library for uniformly accessing different SQL databases. Other libraries and applications are currently in development, for example there is a C++ servlet container based on the Java Servlet specification available in the POCO source code repository on Sourceforge. POCO also has its own documentation generation tool, named PocoDoc, also available from the repository.

POCO aims to be for network-centric, cross-platform C++ software development what Apple's Cocoa is for Mac development, or Ruby on Rails is for Web development – a powerful, yet easy and fun to use platform to build your applications upon. POCO is built strictly using standard ANSI/ISO C++, including the standard library. The contributors attempt to find a good balance between using advanced C++ features and keeping the classes comprehensible and the code clean, consistent and easy to maintain.

The Foundation library

The Foundation library makes up the heart of POCO. It contains the underlying platform abstraction layer, as well as frequently used utility classes and functions. The Foundation library contains types for fixed-size integers, functions for converting integers between byte orders, an **`Any`** class (based on **`boost::Any`**), utilities for error handling and debugging, including various exception classes and support for assertions. Also available are a number of classes for memory management, including reference counting based smart pointers, as well as classes for buffer management and memory pools. For string handling, POCO contains a number of functions that among other things, trim strings, perform case insensitive comparisons and case conversions. Basic support for Unicode text is also available in the form of classes that convert text between different character encodings, including UTF-8 and UTF-16. Support for formatting and parsing numbers is there, including a typesafe variant of **`sprintf`**. Regular expressions based on the well-known PCRE library [2] are provided as well.

POCO gives you classes for handling dates and times in various variants. For accessing the file system, POCO has **`File`** and **`Path`** classes, as well as a **`DirectoryIterator`** class. In many applications, some parts of the application need to tell other parts that something has happened. In POCO, **`NotificationCenter`**, **`NotificationQueue`** and events (similar to C# events) make this easy. Listing 1 shows how POCO events can be used. In this example, class **`Source`** has a public event named **`theEvent`**, having an argument of type **`int`**. Subscribers can subscribe by calling **`operator +=`** and unsubscribe by calling **`operator -=`**, passing a pointer to an object and a pointer to a member function. The event can be fired by calling **`operator ()`**, as its done in **`Source::fireEvent()`**.

I have already mentioned the stream classes available in POCO. These are augmented by **`BinaryReader`** and **`BinaryWriter`** for writing binary data to streams, automatically and transparently handling byte order issues.

In complex multithreaded applications, the only way to find problems or bugs is by writing extensive logging information. POCO provides a powerful and extensible logging framework that supports filtering, routing to different channels, and formatting of log messages. Log messages can be written to the console, a file, the Windows Event Log, the Unix syslog daemon, or to the network. If the channels provided by POCO are not sufficient, it is easy to extend the logging framework with new classes.

For loading (and unloading) shared libraries at runtime, POCO has a low-level **`SharedLibrary`** class. Based on it is the **`ClassLoader`** class template and supporting framework, allowing dynamic loading and unloading of C++ classes at runtime, similar to what's available to Java and .NET developers. The class loader framework also makes it a breeze to implement plug-in support for applications in a platform-independent way.

GÜNTER OBILTSCHNIG

Günter is the founder and managing director of Applied Informatics and spends his rare spare time listening to music, reading and running. His blog can be found at obiltschnig.com.

Finally, POCO Foundation contains multithreading abstractions at different levels. Starting with a **`Thread`** class and the usual synchronization primitives (**`Mutex`**, **`ScopedLock`**, **`Event`**, **`Semaphore`**, **`RWLock`**), a **`ThreadPool`** class and support for thread-local storage, also high level abstractions like active objects are available. Simply speaking, an active object is an object that has methods executing in their own thread. This makes asynchronous member function calls possible – call a member function, while the function executes, do a bunch of other things, and, eventually, obtain the function's return value. Listing 2 shows how this is done in POCO. The **`ActiveAdder`** class in Listing 2 defines an active method **`add()`**, implemented by the **`addImpl()`** member function. Invoking the active method in **`main()`** yields an **`ActiveResult`** (also known as a future), that eventually receives the function's return value.

The XML library

The POCO XML library provides support for reading, processing and writing XML. Following one of POCO's guiding principles – don't try to reinvent things that already work – POCO's XML library supports the industry-standard SAX (version 2) [3] and DOM [4] interfaces, familiar to many developers with XML experience. SAX, the Simple API for XML, defines an event-based interface for reading XML. A SAX-based XML parser reads through the XML document and notifies the application whenever it encounters an element, character data, or other XML artifact. A SAX parser does not need to load the complete XML document into memory, so it can be used to parse huge XML files efficiently. In contrast, DOM (Document Object Model) gives the application complete access to an XML document, using a tree-style object hierarchy. For this to work, the DOM parser provided by POCO has to load the entire document into memory. To reduce the memory footprint of the DOM document, the POCO DOM implementation uses string pooling, storing frequently occuring strings such as element and attribute names only once.

The XML library is based on the Expat open source XML parser library [5]. Built on top of Expat are the SAX interfaces, and built on top of the SAX interfaces is the DOM implementation.

For strings, the XML library uses **`std::string`**, with characters encoded in UTF-8. This makes interfacing the XML library to other parts of the application easy. Support for XPath and XSLT will be available in a future release.

The Util library

The Util library has a somewhat misleading name, as it basically contains a framework for creating command-line and server applications. Included is support for handling command line arguments (validation, binding to configuration properties, etc.) and managing configuration information. Different configuration file formats are supported – Windows-style INI files, Java-style property files, XML files and the Windows registry.

For server applications, the framework provides transparent support for Windows services and Unix daemons. Every server application can be registered and run as a Windows service, with no extra code required. Of course, all server applications can still be executed from the command line, which makes testing and debugging easier.

The Net library

POCO's Net library makes it easy to write network-based applications. No matter whether your application simply needs to send data over a plain TCP socket, or whether your application needs a full-fledged built-in HTTP server, you will find something useful in the Net library.

At the lowest level, the Net library contains socket classes, supporting TCP stream and server sockets, UDP sockets, multicast sockets, ICMP and raw sockets. If your application needs secure sockets, these are available in the NetSSL library, implemented using OpenSSL [6]. Based on the socket classes are two frameworks for building TCP servers – one for multithreaded servers (one thread per connection, taken from a thread pool), one for servers based on the Acceptor-Reactor pattern [7]. The multithreaded TCPServer class and its supporting framework are also the

Listing 1

```
#include "Poco/BasicEvent.h"
#include "Poco/Delegate.h"
#include <iostream>
using Poco::BasicEvent;
using Poco::Delegate;
class Source
{
public:
    BasicEvent<int> theEvent;
    void fireEvent(int n)
    {
        theEvent(this, n);
    }
};
class Target
{
public:
    void onEvent(const void* pSender, int& arg)
    {
        std::cout << "onEvent: "
                  << arg << std::endl;
    }
};
int main(int argc, char** argv)
{
    Source source;
    Target target;
    source.theEvent += Delegate<Target, int>(
        &target, &Target::onEvent);
    source.fireEvent(42);
    source.theEvent -= Delegate<Target, int>(
        &target, &Target::onEvent);
    return 0;
}
```

Listing 2

```
#include "Poco/ActiveMethod.h"
#include "Poco/ActiveResult.h"
#include <utility>
using Poco::ActiveMethod;
using Poco::ActiveResult;
class ActiveAdder
{
public:
    ActiveObject(): add(this,
                        &ActiveAdder::addImpl)
    {
    }
    ActiveMethod<int, std::pair<int, int>,
       ActiveAdder> add;
private:
    int addImpl(const std::pair<int, int>& args)
    {
        return args.first + args.second;
    }
};
int main(int argc, char** argv)
{
    ActiveAdder adder;

    ActiveResult<int> sum =
       adder.add(std::make_pair(1, 2));
    // do other things
    sum.wait();
    std::cout << sum.data() << std::endl;
    return 0;
}
```

foundation for POCO's HTTP server implementation. On the client side, the Net library provides classes for talking to HTTP servers, for sending and receiving files using the FTP protocol, for sending mail messages (including attachments) using SMTP and for receiving mail from a POP3 server.

Putting it all together

Listing 3 shows the implementation of a simple HTTP server using the POCO libraries. The server returns a HTML document showing the current date and time. The application framework is used to build a server application that can run as a Windows service, or Unix daemon process. Of course, the same executable can also directly be started from the shell. For use with the HTTP server framework, a `TimeRequestHandler` class is defined that servers incoming requests by returning an HTML document containing the current date and time. Also, for each incoming request, a message is logged using the logging framework. Together with the `TimeRequestHandler` class, a corresponding factory class, `TimeRequestHandlerFactory` is needed; an instance of the factory is passed to the HTTP server object. The `HTTPTimeServer` application class defines a command line argument help by overriding the `defineOptions()` member function of `ServerApplication`. It also reads in the default application configuration file (in `initialize()`) and obtains the value of some configuration properties in `main()`, before starting the HTTP server.

Conclusions and outlook

In the two and a half years since its birth, POCO has become a mature citizen in the C++ library world. The project is actively maintained, and the number of users and contributors is growing steadily. POCO is used in a wide range of applications, from simple command line tools to steel mill automation and building management systems. The POCO libraries are licensed under the Boost licence, which makes them free for both open source and commercial use. Support for POCO is available from the POCO Community website, containing forums, a weblog and a wiki, as well as from the poco-develop mailing list. Commercial support is also available. POCO runs on all major platforms (Windows, various Unixes including Mac OS X, Linux and some embedded platforms) and requires a reasonably modern C++ compiler (Microsoft Visual C++ 2003 and GCC 3.4 are fine).

Whenever you have to write a network-centric application in C++, you should definitely consider POCO for the task. To get you started quickly, POCO comes with a lot of sample code. A growing number of tutorials and how-tos on the project website augment the reference documentation. If you want to contribute to POCO, you are more than welcome. The simplest way to contribute is by sending us feedback, feature requests or bug reports. If you want to contribute code, we are always looking for people willing to take over development tasks. And if you have an idea for a new library or application based on POCO, we'll be happy to host your project on the POCO website.

And if you want to know what happened to my book – well, coding is much more fun than writing about coding, so the book still sits half-finished somewhere on my hard disk... ■

Acknowledgements

I would like to thank Aleksandar Fabijanic for being one of the first users, and the first contributor to the POCO project, soon after the project's debut on Sourceforge.

Listing 3

```
#include "Poco/Net/HTTPServer.h"
#include "Poco/Net/HTTPRequestHandler.h"
#include "Poco/Net/HTTPRequestHandlerFactory.h"
#include "Poco/Net/HTTPServerParams.h"
#include "Poco/Net/HTTPServerRequest.h"
#include "Poco/Net/HTTPServerResponse.h"
#include "Poco/Net/HTTPServerParams.h"
#include "Poco/Net/ServerSocket.h"
#include "Poco/Timestamp.h"
#include "Poco/DateTimeFormatter.h"
#include "Poco/DateTimeFormat.h"
#include "Poco/Exception.h"
#include "Poco/ThreadPool.h"
#include "Poco/Util/ServerApplication.h"
#include "Poco/Util/Option.h"
#include "Poco/Util/OptionSet.h"
#include "Poco/Util/HelpFormatter.h"
#include <iostream>

using Poco::Net::ServerSocket;
using Poco::Net::HTTPRequestHandler;
using Poco::Net::HTTPRequestHandlerFactory;
using Poco::Net::HTTPServer;
using Poco::Net::HTTPServerRequest;
using Poco::Net::HTTPServerResponse;
using Poco::Net::HTTPServerParams;
using Poco::Timestamp;
using Poco::DateTimeFormatter;
using Poco::DateTimeFormat;
using Poco::ThreadPool;
using Poco::Util::ServerApplication;
using Poco::Util::Application;
using Poco::Util::Option;
using Poco::Util::OptionSet;
using Poco::Util::OptionCallback
using Poco::Util::HelpFormatter;

class TimeRequestHandler:
   public HTTPRequestHandler
{
public:
  TimeRequestHandler(
     const std::string& format): _format(format)
  {
  }

  void handleRequest(HTTPServerRequest& request,
                     HTTPServerResponse& response)
  {
    Application& app = Application::instance();
    app.logger().information("Request from "
       + request.clientAddress().toString());
    Timestamp now;
    std::string dt(DateTimeFormatter::format(
       now, _format));
    response.setChunkedTransferEncoding(true);
    response.setContentType("text/html");
    std::ostream& ostr = response.send();
    ostr << "<html><head><title>HTTPTimeServer
       powered by "
       "C++ Portable Components</title>";
    ostr << "<meta http-equiv=\"refresh\"
       content=\"1\"></head>";
    ostr << "<body><p style=\"text-align: center; "
       "font-size: 48px;\">";
    ostr << dt;
    ostr << "</p></body></html>";
  }
```

Listing 3 (cont'd)

```
private:
  std::string _format;
};

class TimeRequestHandlerFactory
   : public HTTPRequestHandlerFactory
{

public:
  TimeRequestHandlerFactory(
     const std::string& format): _format(format)
    {
    }

  HTTPRequestHandler* createRequestHandler(
     const HTTPServerRequest& request)
  {
    if (request.getURI() == "/")
       return new TimeRequestHandler(_format);
    else
      return 0;
    }

private:
  std::string _format;
};

class HTTPTimeServer
   : public Poco::Util::ServerApplication
{

public:
  HTTPTimeServer(): _helpRequested(false)
  {
  }
  ~HTTPTimeServer()
  {
  }

protected:
  void initialize(Application& self)
  {
    loadConfiguration();
    ServerApplication::initialize(self);
  }
  void uninitialize()
  {
    ServerApplication::uninitialize();
  }
  void defineOptions(OptionSet& options)
  {
    ServerApplication::defineOptions(options);
    options.addOption(
    Option("help", "h",
       "display argument help information")
       .required(false)
       .repeatable(false)
       .callback(
        OptionCallback<HTTPTimeServer>(
        this, &SampleApp::handleHelp)));
  }
  void handleHelp(const std::string& name,
                  const std::string& value)
  {
    HelpFormatter helpFormatter(options());
    helpFormatter.setCommand(commandName());
    helpFormatter.setUsage("OPTIONS");
    helpFormatter.setHeader(
       "A web server that serves the current date
        and time.");
    helpFormatter.format(std::cout);
    stopOptionsProcessing();
    _helpRequested = true;
  }

  int main(const std::vector<std::string>& args)
  {
    if (!_helpRequested)
    {
      unsigned short port = (unsigned short)
      config().getInt("HTTPTimeServer.port",
9980);
      std::string format(
         config().getString(
         "HTTPTimeServer.format",
         DateTimeFormat::SORTABLE_FORMAT));

      ServerSocket svs(port);
      HTTPServer srv(
         new TimeRequestHandlerFactory(format),
         svs, new HTTPServerParams);
      srv.start();
      waitForTerminationRequest();
      srv.stop();
    }
    return Application::EXIT_OK;
  }

private:
  bool _helpRequested;
};

int main(int argc, char** argv)
{
  HTTPTimeServer app;
  return app.run(argc, argv);
}
```

Listing 3 (cont'd)

References

1. The POCO Community Website, http://www.pocoproject.org
2. The PCRE – Perl Compatible Regular Expressions Library, http://www.pcre.org
3. The SAX Project, http://www.saxproject.org
4. The W3C Document Object Model, http://www.w3.org/DOM/
5. The Expat XML Parser, http://www.libexpat.org
6. The OpenSSL Project, http://www.openssl.org
7. Reactor – An Object Behavioral Pattern for Demultiplexing and Dispatching Handles for Synchronous Events, Douglas C. Schmidt, http://www.cs.wustl.edu/~schmidt/PDF/reactor-siemens.pdf

The Trouble with Version Numbers

Thomas Guest untangles the version numbering puzzle.

Introduction

Software version numbers should be straightforward to implement. Their sequencing is hardly subtle: version 1.0 is the first production quality release; version 1.1 improves on it, version 1.2 is a little better; and so on until we get to version 2.0, which delivers more substantial changes. Then comes 2.1, then 2.2 ...

As anyone who has tried to implement such a scheme will realise, it can be a surprising source of problems, and although these problems have been tackled by many projects in many organisations there seems to be no consensus on how to reach a solution. To give an example: deriving the version number from the version control system is tempting, but ultimately turns out to be unsatisfactory.

This article discusses the problems in more depth and presents a simple solution. It goes on to argue that perhaps the emphasis on version numbers is itself the problem, and one which requires a more radical solution.

A flawed versioning recipe

Here's a sensible but flawed recipe for creating a versioned software release. To make the discussion easier to follow, let's suppose it's version 2.0 we want to release. Let's also suppose that a file named `VERSION` is the sole point of version information for the software [1].

0. At some suitable point, branch the software to isolate the release from noise on the main development trunk. Edit `VERSION` to read '2.0.0 RELEASE BRANCH' and check this change in.
1. Create a build from the release branch.
2. Test this build.
3. If the tests pass, edit `VERSION` to read 2.0, check the change in, and go to step 5.
4. The tests haven't passed, so we need to fix all the bugs we've found and return to step 1 (or even, in extreme circumstances, to step 0).
5. Tag the release branch. Checkout this tagged version of the code and create our final release build from it.

The problems

There are several problems with this procedure.

I haven't explicitly stated how the builds are being created and how they're being tested, but reading between the lines suggests that it's a little ad hoc. It could well be that one of the developers generates the builds from a personal working copy, runs a few sanity checks, then throws the code across to the test team for a more thorough thrashing.

Such an approach exposes us to an unacceptable level of human error. Instead, we need a machine to ensure that our builds are clean and reproducible. Before worrying about how we version the software, we must ensure we have a build server to automatically generate builds for us, and to run as many tests as a machine can on these builds, collating and publishing the results. Inevitably, there will still be a need for manual testing; but this build server should become the single source of builds for the manual testers.

THOMAS GUEST

Thomas is an enthusiastic and experienced programmer. He has developed software for everything from embedded devices to clustered servers. His website is http://www.wordaligned.org Contact him at thomas.guest@gmail.com

Even when this build server is in place and doing its job, the procedure described suffers two major problems.

1. There is still too much manual intervention: somebody has to remember to edit the `VERSION` file, and that somebody had better get it right. Typically, as release dates close in, the pressure increases, and editing fingers become less steady.
2. The final tagged build isn't the build which actually passed all the manual tests. We rebuilt it! What's worse, we *changed the code* before rebuilding – we changed the `VERSION` file, and we checked the code out of the repository in a different way.

Keyword expansion

We might turn to our version control system to help us overcome the manual intervention problem. Rather than edit a file by hand, shouldn't the version control system provide the version number directly? This is a seductive argument but I'm going to suggest it's wrong. Before explaining why I think it's wrong, let's show how you can indeed derive a version number directly from the metadata stored in the code repository.

To provide a concrete example let's suppose we're using Subversion for version control. Here's a typical Subversion repository layout:

```
.
`-- trunk
|   |-- file1
|   `-- file2
|-- branches
|   |-- 1.0
|   |   |-- file1
|   |   `-- file2
|   `-- 2.0
|       |-- file1
|       `-- file2
|-- tags
    |-- 1.0
    |   |-- file1
    |   `-- file2
    `-- 2.0
        |-- file1
        `-- file2
```

The trunk area is where most development goes on. When we want to branch the code before making a release, we copy the trunk into the branches area; and when we finally freeze the release, we tag it by copying it into the tags area. To check out release 2.0 of the software we'd issue the command:

```
svn checkout svn://svnserver/tags/2.0
```

As you can see, the repository URL embeds the desired version string, 2.0. If we want to get the `VERSION` file to reflect the URL it was checked out from, we must enable keyword expansion and set its contents to read:

```
$URL: $
```

When we update this file on the trunk, the magic `$URL:` $` keyword expands to read something like:

```
$URL: svn://svnserver/trunk/VERSION $
```

When we copy this file to our 2.0 branch and update, we'll see:

```
$URL: svn://svnserver/branches/2.0/VERSION $
```

and in the tagged release area we get:

```
$URL: svn://svnserver/tags/2.0/VERSION $
```

With some simple text parsing we can extract this information. Here's a minimal Python program which parses the repository URL it came from in order to display version information.

```
#!/usr/bin/env python
def version():
    " Return the software version. "
    url = "$URL$"
    import re
    search = re.compile(
       "svn://svnserver/tags/([^/]*)").search
    match = search(url)
    return match.group(1) if match else
"Development"

print "Version:", version()
```

If this program has been checked out from a base URL `svn://svnserver/tags/2.0`, running it yields the output:

```
Version: 2.0
```

Running it checked out from the trunk, we'll see:

```
Version: Development.
```

Note, incidentally, that the CVS keyword designed for this purpose is **$Name$** – this keyword won't even expand unless we checked out a tagged version of the code.

A misuse of keyword expansion

Look at what's happening here: we tag the software to ensure we can recover exactly what went into a build; but by enabling keyword expansion, the code we check out *differs* depending on the repository URL we use to access it. By tangling the software with version control meta-data we're changing the very thing we want to stabilise.

What gets tested anyway?

It may appear that some judicious use of keyword expansion will help us automate the software version generation, but as we can see, it does so at the expense of amplifying the second problem – which I argue us the more serious.

Let's return to this second problem, then. We've created a chicken and egg situation. We don't want to award the software its final version number until we've tested it; but the version number is part of the software, and we can't test the final version of the software until we've set its version number. Which should come first?

We may convince ourselves that we're making a fuss over nothing important. How big a change is it to change the software version and nothing else? A few text strings, perhaps; the contents of a dialog box. Maybe it has an effect on the licence sub-system. Oh, and the documentation too. Surely nothing much can go wrong with these simple changes and a quick set of sanity checks should confirm they have been correctly applied? If we're really worried, we could always re-run the full set of tests.

These arguments don't convince me. When we get close to a release, impatience and carelessness can set in. It would be foolish to think the testers wouldn't baulk at repeating the full set of system tests for no good reason. And it would be equally foolish to assume the version change has had no side-effects.

Build numbers and version numbers

Let's question the assumption that applying the version number should be the *last* step in the release procedure. Why is this?

Essentially it's because we don't want to have more than one version 2.0: if a customer contacts us to report a problem with version 2.0 of our software, then we'd better be able to identify exactly what it is they're running. Indeed, if the system testers report a problem with version 2.0 of the software, we'd equally like to know which build they're talking about. If we fixed the software version at 2.0 and then continued to work on the code until it reached release quality, how would we ever identify the real version 2.0?

Note, though, that if we were prepared to allow our build server to generate a unique build number for every build it produces, and to use this as our version number, we'd have no problems. If build 1729 was the build which passed the tests, then our release could simply be identified as version 1729 – with no change required. Unfortunately version numbers aren't just numbers; conventions dictate how their fields are interpreted, and a version of 1729 flouts these conventions.

The best of both

The way to break out of this apparent conflict is simple. We must give up on the idea of tying the version number to the version control system: it just doesn't work. Instead, we can adapt the build number idea to automate our versioning as follows.

1. Again, a single file – `VERSION`, let's say – must provide the single source of version information. This file is version controlled, but doesn't use keyword expansion [2]. Instead, it will (generally) only be modified automatically by the build system.
2. At the *start* of the countdown to release 2.0, this file reads 2.0.0. Here, the leading 2.0 is the major and minor part of the version number, and the trailing 0 is the build number.
3. From this point onwards, each time the build server produces a build it uses up a build number. As a post-build step, the server edits the `VERSION` file to increment the build number by 1, then checks this change in, ensuring that no subsequent build can have the same version number.

We've created a chicken and egg situation . . . Which should come first?

We now have the build server generating a series of release candidates, each with its own version number, 2.0.0, 2.0.1, 2.0.2, ... When a release candidate meets the required quality level it can be promoted to being a formal release. This promotion is essentially a book-keeping operation: the software may be tagged, the deliverables may be archived; but whatever happens, there should be no further change to the code, and what gets shipped will be *identical* to what was tested.

Refinements

You'll notice that the full version of the final 2.0 release is unlikely to be 2.0.0. If it took us another 29 builds to fix all the defects, we will ship 2.0.29 – a rather less attractive number. Typically, in most cases, we should abbreviate the version number before presenting it to the user. You certainly won't see a press release announcing the completion of version 2.0.29!

Often, a three part version number is insufficient. It's more common to choose a number of the form 2.2.3.67, interpreted as: major version 2, minor version 2, patch number 3, build number 67.

There is a convention (which I believe originates in the Linux community [4]) to use odd minor version numbers for ongoing development and even version numbers for released versions. Thus, while the release team are knocking 2.0.x into shape, the development team continue with 2.1.x on the trunk.

Scripts should be written to perform all the common operations: to create release branches, to promote release candidates into full releases, and so on. It's important to have these scripts, and indeed the whole process, in place well before the countdown to the final release.

Concluding thoughts

Even with the version numbers sorted out in the manner described, this can be a painful way to develop software. In this article I've described 'big

Libris Unity

Ian Brunlett talks us through some unofficial interfaces.

OK, what is an unofficial interface? For my example, I think it is easier to describe rather than define. Imagine you've got an application that you want to sell lots of copies of while also supporting customer specific extensions. And you want to do it with one source code base.

The application I'll discuss is (was) the Libris Opac / Unity / BNB. Libris started up in 1993 and went broke in 1998. The final copies of the source code CDRs and dataset DAT tapes are probably still sitting in a fireproof safe in Talis, a competitor of Libris that bought the intellectual rights of the system for £5000. To make discussion easier, I'll refer to the family of search engines as Unity, the flagship product of Libris.

Another of Libris' programmers wrote a module to handle windows style `.ini` settings files. To handle the multi-user aspects of Unity, some functions were written on top of this module. In particular, I wanted to have defaults specifiable by Unity when handling the `.ini` files. I just wanted to say the C equivalent of 'read this setting and here's a default value for you to return if the setting is missing'. I was particularly concerned about getting support calls from Libris customer services staff on site wondering why Unity was behaving the way it was. I figured that they'd have to be reasonably familiar with the settings file – but that defaults would make their jobs harder.

So I implemented the function **`GetSettingPutDefault`**. The caller would invoke **`GetSettingPutDefault`** with both:

1. the name of the setting required
2. a default value.

If the setting didn't exist then **`GetSettingPutDefault`** would do two special things:

1. it would write/put the default value into the settings file
2. return the default value.

One of the unexpected bonuses of this was that if you deleted the settings file, running Unity would result in a bare bones default menu system up and running instead of catastrophically halting.

There are certain development projects that merit forking the source code base and most people would say that supporting DOS (actually 32 bit protected mode DOS, using the DOS4GW DOS extender) and the Windows (in this case the WIN32 API as supplied by Windows 3.11, Windows 9x and Windows NT 4 – and documented by Charles Petzold in his book *Programming Windows 95*) would require a fork in the source code base. As things turned out, using link-time polymorphism, clever coding, a fancy makefile meant that the source code did not have to be forked. Quite simply, the entire system sat on top of some unofficial interfaces. There was an unwritten convention that Unity would only use

IAN BRUNTLETT

Ian has been involved in broad spectrum of software systems and languages. He works as a volunteer, teaching people with mental health problems how to use and program computers. He can be reached at ianbruntlett@hotmail.com

The Trouble with Version Numbers (continued)

bang' releases – perhaps a single major release is made every year. In between these releases the main trunk of the codebase is allowed to fall short of release quality: things break and it's only when we create a release branch that we undertake to fix them. There's too much dependency on a team of manual testers to shake out problems, with the effect that the software developers have neglected this duty.

Meanwhile, the sales and marketing team are busily hyping the next release. It's to be announced at an international trade show: How's that for an immovable deadline [5]?

Customers must be persuaded to upgrade but some fail to be excited by the new 2.0 features, seeing instead a bunch of changes they don't particularly want. Instead they'd like to continue with version 1.0, which is the version they purchased – though naturally they demand support and bug-fixes.

Eventually the code base spreads over several branches which have become increasingly laden with patches merged in all directions. The version control system can handle it but can the developers? Although we have several active versions of the software, it's moving irrevocably into maintenance mode.

It seems to me that the emphasis on version numbers and features is wrong, and may indeed take some of the blame for this grim situation. It's *wrong* to risk the stability of the main development trunk and reserve branches for stable code. Instead we should aim for a pristine trunk and use branches for work-in-progress and experiments, so that at any point during development, the tip of the trunk represents the best code we have.

It's also wrong to create an environment in which customers cling to old versions of software. Instead, we should consider upgrades from the outset, and allow our software to migrate softly from version to version. ■

Notes and references

1. In other words, any part of the system which needs the version number must derive it, either at build- or run-time, from this single file. This typically includes the user interface, the documentation, the licensing system. By enforcing a single point of version information, we at least ensure consistency.
2. I also strongly recommend that keyword expansion is disabled for all files in the repository, for the reasons described in this article and also in [4].
3. 'Keyword Substitution – Just Say No', Thomas Guest http://blog.wordaligned.org/articles/2006/08/02/keyword-substitution-just-say-no

   A brief article which argues that keyword expansion is a version control mis-feature.
4. 'Software versioning', Wikipedia http://en.wikipedia.org/wiki/Software_versioning

   A discussion of version numbers including details of Linux kernel version numbers.
5. 'The Other Road Ahead', Paul Graham, http://paulgraham.com/road.html

   An excellent essay in which Graham eloquently describes the advantages of supplying a web-based software service. Here's what he has to say about a typical desktop software release process:

   In the desktop software business, doing a release is a huge trauma, in which the whole company sweats and strains to push out a single, giant piece of code. Obvious comparisons suggest themselves, both to the process and the resulting product.

DISPLAY.C and DISPLAY.H for handling screen display work. There was another unwritten convention that DISPLAY.C would rely on a more platform specific module such as DOSDISP.C to do the gritty displaying of information.

C++ can enforce interfaces that are provided by base classes and their offspring. I'm still coming to grips with the design ramifications of this part of the language.

C, however, is a bit more tolerant of, say, laissez-faire interfaces. On the other hand you can do this laissez-faire stuff in C++ too. When you're dealing with multi-platform code, you use the makefile to pass on flags to the preprocessor and to decide which modules to compile and the preprocessor to handle platform specific stuff like deciding which implementation of a key function will get included in the application.

Unity initially ran on 33MHz 486s, DOS 6.2 and 4MB RAM. As time progressed PCs became faster and memory capacities increased. The earliest versions (1–3) of the search engines (let's call them Opac as this pre-dates Unity) ran in text mode and, to display anything they did a **system()** call to do a **type filename** command where **filename** was an ANSI text file with embedded control codes to change colours. And they were real-mode DOS applications that were compiled using Borland's Turbo C++. And, if I recall this correctly, the Opac used a variety of ways to display things on screen.

Opac V3 was where I came in. In December of 1993, the source code was given to me with a copy of Watcom C++ and I was asked to port the Opac to Watcom C++ and make use of the extra (extended) memory available by DOS4GW. It was a mammoth task. No sooner had I got one module compiling when another reared its head. The source modules were named LIBRISX.CPP where X was the first name of what the module was meant to do. When I got Opac V3 compiled and running – after a lot of hard work – the next change request was to get it to compile using Watcom C, in order to save about £5000 in Watcom licensing fees. After that, bit mapped graphics hit town. Another Libris programmer working on a different family of search engines, Libris Community Information, developed a unique visual look to take advantage of a bitmapped display (256 bit colours, 640x480 pixels). So the next task was to rewrite the Opac to have a glitzy user interface identical in a similar style suitable for the Opac. So all the code that wrote things on the screen had to be changed to use only one way of displaying things on screen and to be done in a fashion that encouraged porting to other operating systems. So DISPLAY.C was born – it provided a 'higher level' of support, doing things like clearing the screen, displaying text, changing colours and displaying images of keys that were provided on special keyboards used by the OPAC. It in turn called upon DOSDISP.C to do the technical stuff.

Later on, perhaps in late 1997, I was called upon to port Unity/Opac to Windows. At the time, Windows programming meant MFC or WIN32. I looked at both and figured WIN32 would give me more flexibility. So a whole load of WIN32 specific code was written, to name WINDISP.C and WINMICK.C (mouse handling) as part of the work involved. As it turned out, the DOS4GW and WIN32 versions looked and ran more or less identically.

How did I pull it off? Well, I turned Petzold's WinMain on its head. Instead of the Windows version of Unity having a message pump that waited for messages (**WM_CLOSE**, **WM_PAINT**), I had Unity run pretty much like a conventional procedural application. Basically, Unity wrote things on screen (in this case a window) and, when it wanted keyboard or mouse input, then it would call **GetValidKCNUM** which would 1) save a bit map image of its own screen (to support **WM_PAINT** requests) and 2) look at what Windows was telling it through the WIN32 **GetMessage** / **PeekMessage** API calls (not sure if I remember the API names completely correctly).

I explain link-time polymorphism like this: unofficially decide on a convention, say that certain functions / structs

1. will act in a certain way
2. will be provided by certain modules (.C or .H files).

without compiler intervention. ■

Influences

1. The C standard library – made me think of things where something could be coded to provide a standard API for different platforms. Instead of having to rewrite everything.
2. *Object Oriented Analysis* by Coad & Yourdon. This, and some hard work, enabled me to get a lot of benefits from structs.
3. The GNU C, YACC and AWK source files when I encountered them for the first time, on the Sinclair QL.

You've Gotta Get On To Get Down

Peter Pilgrim introduces a new series and the Java User Group.

Welcome to the first in a new regular series of articles that I am writing for the ACCU, actually I am using **Writely** (or rather **Google Docs**). I hope to expand of news going on a Java World from a server-side point of view. I am the founder and the leader of a JUG, a Java User Group, called **JAVAWUG**, which stands for Java Web User Group. I started the group way back in May 2004 when I sent out an electronic mail message to the **Struts** user mailing list. I asked first, if there were any developers heading over to **JavaONE** the worldwide developer conference organised by Sun Microsystems in San Francisco, California. Second, I asked, if there were any Struts users ready to meet up regularly and form a user group in London. Our first meeting took place at the infamous Waxy O'Connors pub near Piccadilly. Since then I had the pleasure (and sometimes displeasure) of organising 25 of these so-called birds-of-a-feathers. We tend to meet up about every two months or so. We had some very kind hosts like **Oracle**, **Sun**, and **Skills Matter** provide a room with audio/visual facilities so that JAVAWUG could host professional-style presentations. It has been a learning curve, indeed, learning how to lead a group of disparate people. My hard efforts were rewarded when I was confirmed in February into the Sun Java Champions program, and that is now enough about me.

prediction of the future is always uncertain, but you have to be ready to move, and move fast, to catch the next wave

As a new Java Champion, my reign began with a troubling approach from a fellow JAVAWUG member, who shall remain nameless, as he admitted struggling to find that next IT Job. He asked me to call him at home at some time. I did. We had an busy evening telephone call and I found out that he had been out of work for while, but had considerable experience from previous companies. I critiqued his CV. It had fallen into some common traps: it was way too long and needed re-ordering to place the most recent roles first, highlighting relevant experience. I was flummoxed as how I could help him other than just give my advice. I realised that everyone had some experiences of difficulty being on the so-called *substitute's bench*. It can be quite hard to accept this aspect of modern working life, especially if you do not expect it and suddenly find out yourself there in that situation (again). Think of all the financial, family and pride worries that most of you will have had or will have. An idea dawned on me, and I pushed his approach to the other JAVAWUG members through the mailing list, whilst keeping his personal details confidential. We got a quite few responses and we should have had a lot more. I was disappointed that all developers did not chime in with helpful hints. I guess it must be quite hard to contribute to the community. I concluded that the feeling of being jobless and without contract is far too strong for some people to admit to or respond to, let alone get embarrassed into. A long standing JAVAWUG member posted a hyperlink as a ray of hope about the prospect of good calibre individuals and the proverbial skills shortage (http://www.javaworld.com/javaworld/jw-03-2007/jw-0305-jobs.html):

> As new technologies continue to enter the business world, the demand for these skills will continue to grow this year and beyond. Therefore, wages will continue to increase as hiring managers put a premium on candidates with these maturing skill sets.

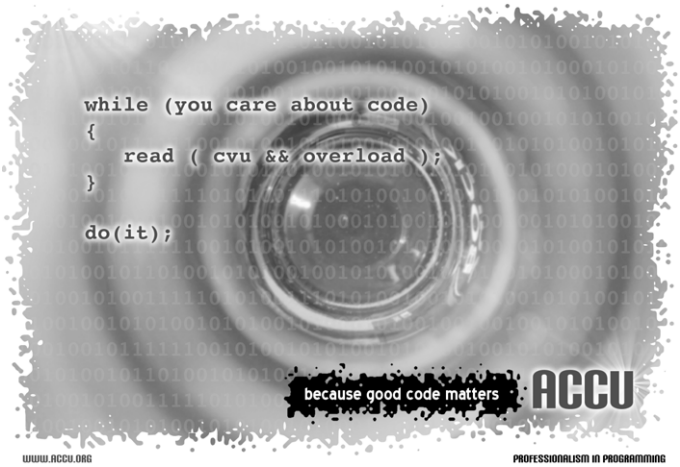
This is fabulously great! Especially if you have those highly desired skills already. If you don't then you could say, 'Woe is me!', but you ought to put your finger in the air and think about the skills that might be relevant in the next year. However, almost everyone agrees it is very uncertain to predict the future. Which relevant Java technologies will be winners? (Java SDK 7, Groovy, JRuby, other Java compatible scripting languages, Java EE 6, REST, SOAP, Grid Computing, EJB 4, AJAX, Dojo, Struts 2, JSF, the JVM, the open sourcing of Java, etc.) The list of *might-be-relevant-technologies* will never be exhausted.

The year 1994 was an interesting year in the IT world. It is best remembered by football sports fans for when Brazil won the FIFA World Cup in the USA, but here is an interesting statistic. At the very beginning of that year there were only about 500 websites on the WWW, but because of a release of browser called **NCSA Mosaic** (the forerunner to Netscape Navigator), that figure jumped to 10000 websites by the end of the year. Suddenly the world stood up and listened. Web hosting and design were born, and a whole first generation of our relatively new industry began.

There are no easy answers, the moral of the story is not that the prediction of the future is always uncertain, but you have to be ready to move, and move fast, to catch the next wave in order to not be left behind. Being a member of the Java User Group certainly helps you get that news faster and also active participation in local and national forums. It's your career after all, so look after it as you do. See you next time. ■

Here are the URLs:

- http://jroller.com/page/javawug
- http://jroller.com/page/peter_pilgrim
- https://java-champions.dev.java.net
- https://jugs.dev.java.net/objectives.html
- http://struts.apache.org
- http://en.wikipedia.org/wiki/mosaic_(web_browser)
- http://java.sun.com/javaone/
- http://java.sun.com/javaee/



PETER PILGRIM

Peter is a Java EE software developer, architect, and Sun Java Champion from London. By days he works as an independent contractor in the investment banking sector.

Adventures in Autoconfiscation #3

Jez Higgins concludes his series on GNU autotools.

Previously in Adventures in Autoconfiscation

In the preceding two episodes[1], I've described my picaresque journey, taking my XML toolkit Arabica[2] from its wobbly homegrown build system toward GNU Autotools, the magic behind **`./configure; make; make install`**. As I begin this article, I'm at the point where I have a no-frills build going. Here I cover adding some flexibility to the build, so it adapts to the presence or absence of third party libraries. Finally, I'll do what I said I'd do last time and examine whether the change to GNU Autotools really did do what I hoped – let more people build Arabica on more platforms more easily but with less fuss and effort on my part.

This isn't the definitive guide to Autotools, it's the how-I-did-it narrative which I hope will inform and entertain.

Customising the build – config.h

The Arabica library builds on top of a third party parser library – **`expat`**[3], **`libxml2`**[4] or **`Xerces`**[5]. The old build system required people to know which library they had (or which of several they wanted to use), the location of its header and shared object files, and set Makefile flags accordingly. The flags were used to generate a C++ header file, `ArabicaConfig.h`, containing a number of **`#define`**s. Those defines were, in turn, used to pull in the appropriate library binding. There were other flags too, which controlled things like wide character support, and whether to use Winsock or 'proper' BSD sockets. Some were set on a platform basis, some at the user discretion. A typical `ArabicaConfig.h` looked like this:

```
#ifndef ARABICA_ARABICA_CONFIG_H
#define ARABICA_ARABICA_CONFIG_H

#define ARABICA_NO_WCHAR_T

#define ARABICA_NO_CODECVT_SPECIALISATIONS

#define USE_EXPAT

#endif
```

Autotools can produce a similar header file, setting flags for all the various bits and pieces it has probed. All you have to do is ask, and you ask by add the **`AC_CONFIG_HEADERS`** macro in **`configure.ac`**, as shown in Listing 1.

Listing 1

```
configure.ac
  AC_INIT([Arabica], [Jan07], [jez@jezuk.co.uk])

  AM_INIT_AUTOMAKE

  AC_PROG_CXX
  AC_PROG_LIBTOOL

  AC_CONFIG_HEADERS([include/SAX/
ArabicaConfig.h])
  AC_CONFIG_FILES([Makefile])
  ...
  AC_OUTPUT
```

Listing 2

```
  /* include/SAX/ArabicaConfig.h.  Generated
     from ArabicaConfig.h.in by configure.  */
  /* include/SAX/ArabicaConfig.h.in.  Generated
     from configure.ac by autoheader.  */

  /* Define to 1 if you have the <dlfcn.h>
     header file. */
  #define HAVE_DLFCN_H 1

  /* Define to 1 if you have the <inttypes.h>
     header file. */
  #define HAVE_INTTYPES_H 1

  /* Define to 1 if you have the <memory.h>
     header file. */
  #define HAVE_MEMORY_H 1

  ... several other #defines I'm not actually
interested in ...
```

A quick round of **`autoreconf`** and **`./configure`**

```
...
configure: creating ./config.status
config.status: creating Makefile
config.status: creating src/Makefile
config.status: creating test/Makefile
config.status: creating test/Utils/Makefile
config.status: creating include/SAX/
 ArabicaConfig.h
config.status: executing depfiles commands
...
```

gives an **`ArabicaConfig.h`** which looks like Listing 2.

All I need to do is get my Arabica specific defines in there, and I'm done. This is the kind of thing you see configure scripts doing all the time, so I assumed it must be straightforward.

You extend what configure looks for by writing your own Autoconf macros. Autoconf macros are written in a language called M4[6]. Although it's been around since shortly after the dawn of Unix, M4 isn't something you encounter every day. As it turns out, you're actually unlikely to encounter writing Autoconf macros either. Almost every significant thing you need to do in a custom Autoconf macro already exists in Autoconf macro library, so you working at one remove from raw M4. You can write your macros inline in **`configure.ac`**, but it's more common to pop them in an external file and just have the call macro in **`configure.ac`**. By convention macro files go in a subdirectory called `m4`, where they'll be picked up automatically.

`Autoconf` macros have the general form

```
AC_DEFUN([macro-name],
         [macro-body])
```


JEZ HIGGINS

Jez works in his attic, living the life of a journeyman programmer. He is currently teaching himself how to make balloon animals. In April last year, he became ACCU Chair. His website is http://www.jezuk.co.uk/

The body can extend over several lines if necessary. Remember that **configure** is a shell script. Custom macros in **configure.ac** are, therefore, being expanded to shell script fragments, which become part of **configure**. In that sense they are metaprograms, but I'm probably making it sound more difficult that it is. It should be born in mind, however, that the output of an Autoconf macro is shell script which will be run later, possibly on an unknown platform. The Autoconf manual has extensive guidance on writing portable shell script, but for common operations an indepth understanding on quite how **awk** operates on Ultrix-1 isn't necessary. The following example should make this clearer.

Customising the build – finding an XML parser

Arabica builds on the **expat**, **libxml2**, or **Xerces** XML parsers. I need my configure to script to find which of these are available. If there are multiple choices choose amongst them.

The first job was to write a macro to look for **expat**. I started where most programming endeavours start these days, by googling. It turns out that searching for <package name> and m4 is a pretty reliable way of finding an Autoconf macro that does, or at least more or less does, what you want. There's also an extensive collection of macros in the Autoconf Macro Archive[7].

Rather than go through the tedious business of writing the macro, I'm going to leap straight to the finished article (Listing 3) and highlight the important parts.

It's not that bad is it? Could be worse – modern conveniences like subroutines and variable scoping don't exist in shell script, after all.

This is more or less the canonical form for a macro which looks for a library. First it searches for a known header, then if that's found it tries to link the library, by looking for a known function in that library. If the link succeeds we can declare victory, in this case by setting a flag. When configure reachs the end of script and the expat library was found, **HAVE_EXPAT** is set and **EXPAT_CFLAGS** and **EXPAT_LIBS** point to the header and library locations. If not, **HAVE_EXPAT** is not set.

The heavy lifting here is provided by the **AC_ARG_WITH**, **AC_COMPILE_IFELSE**, and **AC_CHECK_LIB** macros.

AC_ARG_WITH(package, help-string, [action-if-given], [action-if-not-given]) describes an argument to the configure script. If the user runs configure with **--with-package** or **--without-package** options, run shell commands **action-if-given**. If neither option was given, run shell commands **action-if-not-given**. The option's argument is available in the shell variable **with_package**. The **--without-package** option is equivalent to **--with-package=no**. In this case, if neither **--with-expat** or **--without-expat** is given, I set the with_expat variable myself.

AC_COMPILE_IFELSE(input, [action-if-true], [action-if-false]) compiles a program, running **action-if-true** if successful, and **running action-if-false** otherwise. In this case I just want to try and compile:

```
#include <expat.h>
```

The '#' character is the shell comment character, so I can't use it directly. Autoconf provides a number of quadrigraphs for special characters. The quadrigraph for '#' is the unpronouncable **@%:@**. If you need to check for something more sophisticated than the mere presence of a header, perhaps its presence and contents the **AC_LANG_SOURCE** and **AC_LANG_PROGRAM** macros are useful here[8]. **AC_COMPILE_IFELSE** doesn't try to link.

AC_CHECK_LIB(library, function, [action-if-found], [action-if-not-found]) tests whether a library is available by trying to link a test program that calls function.

When the compiler and linker are invoked, the **CXXFLAGS** and **LIBS** variables are used to pass the compiler and linker options. This is why the macro keeps copies of the initial values of these variables, and resets them at the end of the script. In a shell script all variables are global, so care must be taken with special variables like these.

Working from this template, I wrote two further macros to check for libxml2 and Xerces. The Xerces macro is slightly more involved because Xerces is a C++ library. **AC_CHECK_LIB** plays rather fast and loose with function declarations and is only suitable for checking functions in C libraries. The C++ equivalent is:

```
xerces_save_LIBS="$LDFLAGS"
CXXFLAGS="$CXXFLAGS $XERCES_CFLAGS"
LIBS="$LIBS $XERCES_LIBS -lxerces-c"
AC_LINK_IFELSE([AC_LANG_PROGRAM(
   [[#include <xercesc/util/PlatformUtils.hpp>]],
   [[XERCES_CPP_NAMESPACE::XMLPlatformUtils
     ::Initialize()]])],
   [ XERCES_LIBS="$XERCES_LIBS -lxerces-c"
     xerces_found=yes],
   [ xerces_found=no])
CXXFLAGS="$xerces_save_CXXFLAGS"
LIBS="$xerces_save_LIBS"
```

Listing 3

```
AC_DEFUN([ARABICA_HAS_EXPAT],
[
  AC_ARG_WITH([expat],
              [  --with-expat=PREFIX
               Specify expat library location],
              [],
              [with_expat=yes])

  EXPAT_CFLAGS=
  EXPAT_LIBS=
  if test $with_expat != no; then
    if test $with_expat != yes; then
      expat_possible_path="$with_expat"
    else
      expat_possible_path="/usr /usr/local /opt
       /var"
    fi
    AC_MSG_CHECKING([for expat headers])
    expat_save_CXXFLAGS="$CXXFLAGS"
    expat_found=no
    for expat_path_tmp in
       $expat_possible_path ; do
    CXXFLAGS="$CXXFLAGS -I$expat_path_tmp/
       include"
      AC_COMPILE_IFELSE(
         [@%:@include <expat.h>)],
         [EXPAT_CFLAGS=
          "-I$expat_path_tmp/include"
         EXPAT_LIBS="-L$expat_path_tmp/lib"
         expat_found=yes],
        [])
      CXXFLAGS="$expat_save_CXXFLAGS"
      if test $expat_found = yes; then
        break;
      fi
    done
    AC_MSG_RESULT($expat_found)
    if test $expat_found = yes; then
      AC_CHECK_LIB([expat],
         [XML_ParserCreate],
         [ EXPAT_LIBS="$EXPAT_LIBS -lexpat"
           expat_found=yes ],
         [ expat_found=no ],
         "$EXPAT_LIBS")
      if test $expat_found = yes; then
        HAVE_EXPAT=1
      fi
    fi
  fi
])
```

Listing 4

```
configure.ac
  AC_INIT([Arabica], [Jan07], [jez@jezuk.co.uk])

  AM_INIT_AUTOMAKE

  AC_PROG_CXX
  AC_PROG_LIBTOOL

  AC_LANG([C++])
  ARABICA_HAS_EXPAT
  ARABICA_HAS_LIBXML2
  ARABICA_HAS_XERCES

  AC_CONFIG_HEADERS([include/SAX/
ArabicaConfig.h])
  AC_CONFIG_FILES([Makefile])
  AC_CONFIG_FILES([src/Makefile])
  AC_CONFIG_FILES([test/Makefile])
  AC_CONFIG_FILES([test/Utils/Makefile])
  AC_OUTPUT
```

Adding the new macros to **configure.ac**, preceded by **AC_LANG([C++])** to indicated that test programs should be compiled and linked as C++ rather than C gives Listing 4, and going through a round of **autoreconf** and **./configure** result in Figure 1.

The configure script can detect which XML parsers are available. Now to communicate that to the build. I need to do two things; set a preprocessor symbol in `ArabicaConfig.h` so I can pull in the approprate driver, and have the Arabica library link to the parser library.

My **ARABICA_HAS_*** has macros will have set any or all of **HAVE_EXPAT**, **HAVE_LIBXML2**, and **HAVE_XERCES**, together with a matching pair of variables containing compiler and linker flags. I can write a simple if ladder to set the outputs. But how to set those outputs?

The **AC_DEFINE** macro adds a symbol to the config header. It has the form **AC_DEFINE(variable, value, [description])** and is just what we need.

To compiler and linker flags clearly need to be passed to the compiler and linker. They are invoked by make, so the flags need to be set in the Makefile. Autoconf **AC_SUBST(variable, [value])** macro performs variable replacement in the output files, substituting instances of **@variable@** with the value. This is the mechanism for getting the flags from the configure script into the Makefiles.

Armed with these two macros, I wrote a further macro to select the parser

```
AC_DEFUN([ARABICA_HAS_XML_PARSER],
[
  if test "$HAVE_EXPAT" == "1"; then
    AC_DEFINE([USE_EXPAT], ,
       [define to build against Expat])
    AC_SUBST([PARSER_HEADERS], $EXPAT_CFLAGS)
    AC_SUBST([PARSER_LIBS], $EXPAT_LIBS)
  elif test "$HAVE_LIBXML2" == "1"; then
    ...
  else
    AC_MSG_ERROR([[Cannot find an XML parser
       library.  Arabica needs one of Expat,
       LibXML2 or Xerces]])
  fi
])
```

and added it to `configure.ac`. I updated `src/Makefile.am`[9] to add placeholders for the compiler and link flags

```
   ...
AM_CPPFLAGS =
   -I$(top_srcdir)/include @PARSER_HEADERS@
...
libarabica_la_LDFLAGS= @PARSER_LIBS@
```

And once more around the **autoreconf** and `./configure` loop. Towards the bottom of `ArabicaConfig.h` we find

```
/* define to build against Expat */
#define USE_EXPAT
```

Builds too (see Figure 2).

Readers with eidetic memories will spot that the long list of **-D** options passed to the compiler have gone[10], replaced by the **ArabicaConfig.h** header. Sharpeyed readers will also spot the **-lexpat** in the linker options, the result of the variable of the **AC_SUBST** macro.

Figure 1

```
$ ./configure --help
`configure' configures Arabica Jan07 to adapt to many kinds of systems.

Usage: ./configure [OPTION]... [VAR=VALUE]...

...

Optional Packages:
...
  --with-expat=PREFIX      Specify expat library location
  --with-libxml2=PREFIX    Specify libxml2 library location
  --with-xerces=PREFIX     Specify xerces library location
  ...

  $ ./configure
  ...
  checking how to hardcode library paths into programs... immediate
  checking for expat headers... yes
  checking for XML_ParserCreate in -lexpat... yes
  checking for libxml2 headers... no
  checking for Xerces headers... yes
  checking for XMLPlatformUtils::Initialize in -lxerces-c... yes
  configure: creating ./config.status
  ...
  config.status: include/SAX/ArabicaConfig.h is unchanged
  config.status: executing depfiles commands
  ...
```

Customising the build – no Boost

The **AC_DEFINE** and **AC_SUBST** macros are the two most common ways to customise and configure your build, but sometime they can't quite get you were you want to be.

Arabica consists of several different pieces, which stack on one another. At the bottom is the a SAX layer, wrapping whichever library configure finds. On top of that is a DOM implementation, and on that is an XPath engine. These different layers also have different dependencies. The XPath engine uses Boost, while the other pieces don't. My configure script should check for Boost, as XPath can't be built without it. Its absence isn't completely critical though, since the other pieces can be built. In a case like this, a preprocessor define, environment variable or text substitution isn't really going help. What we need to be able to say is if this condition applies, recurse the build into these

Figure 2

```
  $make
  Making all in src
  make[1]: Entering directory `/home/jez/work/arabica'
if /bin/sh ../libtool --tag=CXX --mode=compile g++ -DHAVE_CONFIG_H -I. -I. -I../include/SAX  -I../
include -I/usr/include    -g -O2 -MT arabica.lo -MD -MP -MF ".deps/arabica.Tpo" -c -o arabica.lo
arabica.cpp; \
  ...
  /bin/sh ../libtool --tag=CXX --mode=link g++  -g -O2   -o libarabica.la -rpath /usr/local/lib -L/usr/
lib -lexpat arabica.lo InputSourceResolver.lo base64codecvt.lo iso88591_utf8.lo ucs2_utf16.lo
ucs2_utf8.lo iso88591utf8codecvt.lo rot13codecvt.lo ucs2utf8codecvt.lo utf16beucs2codecvt.lo
utf16leucs2codecvt.lo utf16utf8codecvt.lo utf8iso88591codecvt.lo utf8ucs2codecvt.lo
XMLCharacterClasses.lo
  ...
  ranlib .libs/libarabica.a
  creating libarabica.la
  (cd .libs && rm -f libarabica.la && ln -s ../libarabica.la libarabica.la)
  make[1]: Leaving directory `/home/jez/work/arabica'
```

subdirectories, or build these executables. Automake's **AM_CONDITIONAL** macro allows us to do exactly that.

At the end of my macro which checks for Boost[11], I have:

```
AM_CONDITIONAL([WANT_XPATH],
               [test "$want_xpath" = "yes"])
```

and in `tests/Makefile.am` I have:

```
SUBDIRS = Utils SAX DOM
if WANT_XPATH
  SUBDIRS += XPath
endif
```

Now, the build will only walk down into the `XPath` directory if the Boost libraries are found.

Build targets for free – install

In addition to the build targets you specify, Autotools provides a number of additional targets. Among the most useful are **install**, **dist** and **dist-check**.

What is installed where are controlled by the `Makefile.am` primaries[12]. A file named in a primary is installed by copying the built file into the appropriate directory. So

```
bin_PROGRAMS = hello
```

would be installed in **$(bindir)**. By default configure **$(bindir)** is `/usr/local/bin`, but that can be changed with a command line parameter. Autotools can also install libraries and headers. Where the platform requires, the install target will relink and take care of any other platform specific jiggery-pokery.

Arabica is implemented mainly in header files[13], and I'm too pragmatic (or lazy, take your pick) to try and keep `Makefile.am`s up to date with a constantly changing list of files. Consequently, the built-in **install** target won't install my headers, because it doesn't know about them. Autoconf accounts for situations like this by providing hook points in the built in targets. For Arabica, I can provide an **install-data-local** target in my `Makefile.am`, and it gets called at the right point in the install. (Listing 5).

This is a straightforward Makefile fragment. I find all the header files (skipping Subversion directories) and copy them into `$(includedir)`. Autotools arrange for `$(includedir)` to be pointing to the correct location.

Autoconf also provides a matching **uninstall** target. Politeness dictates that if you use the **install** hook, you should also provide an **uninstall** hook in your `Makefile.am` (Listing 6).

Autoconf includes a number of other built-in targets, including rules for running tests, maintaining file dependencies, and packaging source files. They all provide similar hook points.

Listing 5

```
install-data-local:
      @echo "----------------------------------------------------------"
      @echo "Installing include files to $(includedir)"
      @echo "----------------------------------------------------------"
      for inc in `cd $(srcdir)/include && find . -type f -print | grep -v \.svn`; \
        do $(INSTALL_HEADER) -D "$(srcdir)/include/$$inc" "$(includedir)/$$inc"; \
      done
```

Listing 6

```
uninstall-local:
      @echo "----------------------------------------------------------"
      @echo "Removing include files from $(includedir)"
      @echo "----------------------------------------------------------"
      for inc in `cd $(srcdir)/include && find . -type f -print | grep -v \.svn`; \
        do rm -rf "$(includedir)/$$inc"; \
      done
      for dir in `cd $(srcdir)/include && find . -type d -print | grep -v \.svn`; \
        do rm -rf "$(includedir)/$$dir"; \
      done
```

What are all those files?

Packages built with Autotools do tend have a lot of files scattered about the place.

Here's an overview.

File	Description
■ AUTHORS ■ NEWS ■ README ■ ChangeLog	Text files that automake expects to find. The GNU Project has guidelines about their use, but I simply created stubs directing people to my website.
■ COPYING ■ INSTALL	Again text files automake expects to find, and can provide for you. The COPYING it provides contains the text of GNU General Public License, for which you may want to substitute an alternative. The provided INSTALL provides general instructions on using configure, building, and installing.
■ configure.ac	The Autoconf source file used to generate configure.
■ aclocal.m4	A generated copy of the m4 macros used by configure.ac.
■ configure	The (enormous) shell script that examines your system, its header files, libraries, and compiler, before generating a set of system specific Makefiles.
■ config.guess ■ config.sub ■ depcomp ■ missing	Support shell scripts used by configure. config.guess and config.sub provide the canoni al system type - e.g. i686-pc-cygwin. depcomp is a script with compiles a program while also capturing its dependencies. missing acts as a stub for GNU programs which might not be present (e.g. yacc, or the autotools themselves), aiming to keep the build going if possible.
■ install-sh	Script to install a program, library, or datafile.
■ ltmain.sh	Part of libtool, providing general library building services.
■ Makefile.am	Automake source used to generate Makefile.in
■ Makefile.in	Template which configure will use to generate the Makefile

That's before running configure - the files that ship in the source package. Somebody building your package will run configure, and discover a few more as a result ...

File	Description
■ config.log	configure's extensive log of the checks it has run, what it found, and how it has expanded its variables.
■ config.status	A script created by configure which generates the output files, creating Makefiles from Makefile.ins. Usually run by configure itself, but can also be run manually to recreate the output files if necessary.
■ libtool	Another script created by configure, this is a system specific library builder. It is called from the generated Makefiles.
■ Makefile	The Makefile configure created from the Makefile.in

Adding a custom target

In addition to the built-in targets, it's also possible to add your own build targets. You simply add the target and its rules to `Makefile.am`. Arabica includes a target to build HTML class documentation

```
docs:
        doxygen doc/arabica.dox
        @echo "-----------------------------------"
        @echo "Generated documents to ./doc/html"
        @echo "-----------------------------------"
```

Expected benefits

My decision to move to a new build system was driven by the fact that my pile of Makefiles had become unworkable. They were increasingly difficult for me to maintain, and difficult for people to use. I chose Autotools expecting it to be able to

- find Arabica's prerequisites – at least an XML parser and optionally Boost
- identify whether **`wchar_t`** was supported
- detect platform specific file extensions
- track file dependencies
- be at least as easy to maintain as my existing setup
- stand a better than even chance of working on the random machine that somebody has just downloaded my code to system

In short, it can. In the course of this article I've outlined macros that identify Arabica's prerequisites. Writing macros is straightforward, if indeed a quick Google doesn't find one for you. Autotools handles file extensions automatically, along with lots of other platform specific details I hadn't even considered.

Since Arabica is mainly implemented in header files, as a developer I was particular keen to have dependencies tracked automatically. The generated Makefiles track dependencies extremely well. In the time I've been using Autotools I've never been caught with a bad build.

Although there was a bit of ramp up to using Autotools, now that I have the relationshop between `configure.ac`, **`configure`**, **`Makefile.am`** and **`Makefile`**s clear in my head, I've found using it extremely easy. Adding new executables to the build takes only a few minutes. Modifying the build to include or exclude certain pieces I've also found to be straightforward to implement. Since build options can be exposed as configure script options, they are much easier for Arabica users to access. They don't need to fish around in a Makefile, they can just pass an option.

My experience with building Arabica on different platforms, and the reports I've had, tell me that a package that uses Autotools stands an extremely good chance of building on some arbitrary machine. Importantly, the configure script allows problems to be identified and reported before we even attempt to compile a link. A message which says 'Can not find an XML parser library. Arabica nees one of Expat, LibXML2 or Xerces.' is much clearer than a screen full of compiler errors caused by a missing header file, or unresolved link error.

For all the criteria I set myself, switching to Autotools has been a success.

Unexpected benefits

Since my initial release of an autoconfiscated Arabica package last September, I've discovered a number of other benefits that I hadn't expected or even considered.

One thing I hadn't expected is that I'm finding `Makefile.am` files much easier to maintain than **`Makefile`**s. Arabica is under relatively energetic development, and so I'm adding new things to the build reasonably often. Despite years of working with **`Makefile`**s, I could rarely get one right first time. `Makefile.am`s are much more concise, and I get them right more often than not. Once I've added a new source file, say, Autotools also takes care of dependency tracking, installation, and source file packaging for me.

I save time both because updating the `Makefile.am` is easier, and because it does more work for me.

As someone with a history of producing not-quite-correct tarballs, I've found the built-in **`dist`** and **`dist-check`** targets to be invaluable. The **`dist`** target creates source `tar.gz`, `tar.bz2` and `zip` files, using the Makefile dependencies. **`Dist-check`** provides extra peace of mind. It bundles up the source files, then unbundles them and tries to build the package. I love it.

According to Sourceforge's statistics, Arabica is now getting more downloads. A new release always did bring a spike in traffic, but I do seem to be seeing a sustained increase in the number of downloads. I don't know who or where most these new downloaders are or what they're doing, but that's part of the fun.

Autotools makes cross-compilation straightforward. People are building Arabica for embedded platforms, with some success. I'm pretty sure my previous system, if not active hostile to cross-compilation, didn't make things any easier.

When **`build`** fails, the emails I have had show that people tend to blame themselves. They write emails containing phrases like: I'm sure it's something I've done and if you could point me in the right direction. Prior to autoconfiscating Arabica, I received perhaps five emails in six years about getting the package build on new platforms. Subsequent to autoconfiscating Arabica, I've received six in the last six months. What's more, in every case but one getting the build going has been straightforward even without my having access to the platform in question.

What this shows, I think, is that people find the ./configure; make; make install incantation comforting. It sends a message that the package author knows what they're doing, and that sends good messages about the package itself. Arabica's initial impressions were good, rather than off-putting. A reliable build means people can concentrate on finding out whether Arabica can actually help them do what they want. If the build doesn't work for whatever reason, people feel it can be fixed, simply because they see Autoconf build working so often.

So was it worth it? Would I do it again?

Yes and yes, although I wouldn't necessarily recommend it universally or unequivocally. For existing systems that work already, I wouldn't recommend you change simply for the sake of changing. For systems where portability isn't a consideration, or where the build is straightforward, I wouldn't necessarily consider Autotools as my only choice.

For something like Arabica, code intended to be built on a number of different platforms, then I think I would now reach for Autotools first. For systems where the build is fluid, where things are coming in and out of the build often, I'd consider Autotools because I've found it easy to maintian. In cases where the build needs to be customisable, for whatever reason (missing header file, basic vs full options , etc), I'd also Autotools a strong candidate.

And it's goodnight from me

And that's more or less it. In the course of these three articles, I've sprinted through my migration to Autoconf. I've discussed what Autoconf is and what it does. I've given examples of common Autoconf operations. Finally this month, I've looked at examples of the various ways to modify the build through preprocess symbols, variable substitutions in Makefiles, conditional targets, and by customising built-in targets.

My experience with Autotools has been, and continues to be good. Should you choose Autotools in the future, hopefully these articles will help you have a similar experience. ■

Notes and references

1. CVu Vol 18 Number 6 and Vol 19 Number 1
2. Code available from http://www.jezuk.co.uk/arabica
3. http://expat.sourceforge.net/ Expat was originally written by James Clark, a real XML big brain, and is widely used. It's my XML parser of first resort.
4. libxml2, the GNOME XML parser, http://www.xmlsoft.org/
5. Xerces-C, an Apache project initially donated by IBM, http://xml.apache.org/xerces-c/
6. See http://www.gnu.org/software/m4. m4 seems to go back to early in Unix history, but I'm not aware of it being widley used outside of Autoconf. The GNU version is still active though, with the latest release as recently as last November.
7. Autoconf Macro Archive contains over 500 macros, indexed by category, http://autoconf-archive.cryp.to/
8. See the Autoconf manual, http://www.gnu.org/software/autoconf/manual/, particularly the index of macros at http://www.gnu.org/software/autoconf/manual/html_node/Autoconf-Macro-Index.html
9. See last time for an overview of Makefile.am files.
10. Again, see last time.
11. Derived from the AX_BOOST_BASE, from http://autoconf-archive.cryp.to/ax_boost_base.html
12. Makefile.am primaries, and this significant of the prefixes in primary names were discuss last time.
13. Currently over 150 header files.

Code Critique Competition 45

Set and collated by Roger Orr.

Please note that participation in this competition is open to all members, whether novice or expert. Readers are also encouraged to comment on published entries, and to supply their own possible code samples for the competition, in any common programming language, to scc@accu.org.

Last issue's code

My thanks to the donator of this piece, who would like to remain anonymous – but you know who you are.

This is a scaled down version of some production code which was longer and slightly more tortuous.

Feed the program (assume here built as `run.exe` or `run`) with arguments like:

```
run foo bar baz
```

And it prints:

```
f,o,o
b,a,r
b,a,z
```

Please critique this code, suggesting how the writer could improve their coding technique.

```
#include <iostream>
#include <iterator>
#include <string>
#include <sstream>
#include <vector>

using std::cin; using std::cout;
using std::ostream_iterator;
using std::stringstream;
using std::string; using std::vector;

int main(int argc, char *argv[])
{
  vector< string > names(
      argv + 1, argv + argc );

  size_t written = 0;
  while( true )
  {
    stringstream output;
    bool added = false;

    if( written == names.size() )
      break;

    string name = names[ written ];
    for( int index = 0; index != name.size();
         ++index )
    {
      output << name[ index ];
      if( index < name.size() - 1 )
        output << ",";

      if( written > names.size() )
        continue;
      added = true;
    }

    if( !added )
      break;

    cout << output.str() << "\n";
    ++written;
  }

  cin.get();
  return 0;
}
```

Critiques

From Nevin :-] Liber <nevin@eviloverlord.com>

To be frank, I really don't like this code. It's messy. Some issues off the top of my head:

- Too much copying (and pointless memory [de]allocation). The names are copied from an array of C strings (**`argv`**) into a vector of **`std::strings`**. Why? Then each name is comma-fied into a **`std::stringstream`**, which is then converted to a **`std::string`** for output to a **`std::ostream`** (**`std::cout`**). Ugh.
- Too much arbitrary control flow. Two nested loops, four **`if`**s, two **`break`**s and one **`continue`** make it difficult to see the execution path through the code. Which means it is easy for a latent, subtle bug to hide in the code. It is hard to reason about the code and it is a maintenance nightmare.
- No Separation of Concerns. The code which comma-fies a word is intertwined with the code that moves from one word to the next. This functionality needs to be decoupled.
- Too many unnecessary artifacts. Things like using **`std::cin`** and using **`std::ostream_iterator`** are in all likelihood left over from previous attempts at implementing this. It is even hard to see if the statement **`if( written > names.size() ) continue;`** is from an old implementation or if some intended functionality is missing (because it is impossible for written to ever be greater than **`names.size()`**).

Personally, I'd just chuck the code and start over:

```
#include <iostream>
std::ostream& Commafy(const char* line,
std::ostream& out = std::cout)
{
  const char* delimiter("");
  for (; *line; ++line)
```


ROGER ORR

Roger has been programming for 20 years, most recently in C++ and Java for various investment banks in Canary Wharf. He joined ACCU in 1999 and the BSI C++ panel in 2002.
He may be contacted at rogero@howzatt.demon.co.uk

```
  {
    out << delimiter << *line;
    delimiter = ",";
  }
  return out;
}

int main(int argc, char* argv[])
{
  if (argc) for (++argv; *argv; ++argv)
    Commafy(*argv) << '\n';
}
```

Yes, that's it. :-).

A few notes:

- **Commafy**: There are two ways to think about how to comma-fy a string **s** of length **N**: either put a comma after each of the characters in the range [0..N-2], or put a comma before the characters in the range [1..N-1]. Picking the latter model simplifies the code; I'd rather not have a special case for **strlen(s) < 2** if I don't have to.

  In addition, I parameterised the output stream, as I can see this function being used for other streams (files, std::stringstreams, etc.).
- **main**: The C++ Standard (section 3.6.1) makes a number of useful guarantees about **argc** and **argv**:
  - **0 <= argc**
  - **0 == argv[argc]**
  - **0 != argv[a]** (when 0 < **argc**, for all **a** in the range [0..**argc**-1])

I take advantage of that to iterate through all the command line parameters.

From Michal Rotkiewicz <michal_hr@yahoo.pl>

The code seems to be correct from the reliability point of view: there are no memory leaks, memory violations and so on.

I will try to point out what may be changed to make this code more readable and I will improve its performance.

At first I would like to clean this code as a few instructions are not necessary:

1. ostream_iterator is not used so both **#include <iterator>** and **using std::ostream_iterator** may be removed. It pays to remove unnecessary headers in terms of compilation time: less headers = less code to compile = faster compilation.
2. **#include <string>** is unnecessary as long as we have **sstream** included.
3. **if (written > names.size()) continue** may be also removed as this condition never becomes fulfilled. Variable written is increased as the last instruction of the **while** loop and it's compared with **names.size()** at the beginning of the **while** loop. Therefore **while** loop is broken faster than if **(written>names.size())** is true.

It's nice that only necessary classes from **std** namespace are introduced. Global namespace shouldn't be polluted with unnecessary classes.

It's worth mentioning that type of index variable should be **size_t** not **int**. In case of long strings **int** may not be big enough to store its length.

Let's move to the performance:

1. storing the input in the vector is not necessary. All vector's benefits are not used. It serves as a plain table in this case. All we need we have placed in ***argv[]**.
2. **stringstream** object is unnecessary as characters may be printed directly from ***argv[]**.

Taking these two points into account we have code like this:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  int j=0;
  for (int i=1;i<argc;i++)
  {
    j=0;

    while(argv[i][j]!='\0')
    {
      printf("%c",argv[i][j]);
      if(argv[i][j+1]!='\0') printf("%c",',');
      j++;
    }
    if (j!=0) printf("%c",'\n');
  }
  return 0;
}
```

This code is almost twice as fast as the original one (checked under Cygwin). Despite getting rid of **vector** and **stringstream**, I used **printf** as it works faster than **cout**. I have tried to use **cout** and have **sync_with_stdio** turned off but it didn't help as much as **printf**.

Commentary

The original code is somewhat obese – it contains one or more variables that have no real purpose and tests that are unnecessary. Both of the solutions come in at just over 1/3 the number of characters, which shows the amount of redundancy present in the original solution.

I am interested in the effect that a poor algorithm has on code complexity – sometimes a minor reworking of the approach can simplify the code. In this example I find the test before writing the delimiter is unclear:

```
if( index < name.size() - 1 )
```

A recent discussion on accu-general showed there are a number of different approaches to this general problem. I personally like Nevin's approach of always writing a, possibly null, delimiter although some seem to find it counter-intuitive when they first come across it.

One common danger with refactoring code however, is of breaking the existing behaviour. This is especially true of boundary conditions. In the code provided, an empty string is a special case as it results in the added variable remaining **false**, and terminating the loop early. Consider execution of the program where one of the arguments supplied is empty:

```
C:>run abc "" def
a,b,c
```

Neither of the critiques supplied produce this output –but interestingly they produce different output from each other!

Nevin's code produces:

```
C:>run abc "" def
a,b,c

d,e,f
```

Michal's code produces:

```
C:>run abc "" def
a,b,c
d,e,f
```

It is by no means certain whether the behaviour of the original code was actually desired, but the fact that there is variable called added does suggest

that this behaviour, odd though it might seem, might have been as designed. One of the difficulties with modifying poorly written code is trying to decide which of the behaviours of the code are expected, which are bugs, and which are bugs the rest of the code relies on.

The winner of CC 44

Both the entries had slightly different behaviour from the original code, so I was tempted to forgo awarding a prize this time round! However, the competition is for the critique and both entrants made useful comments about the original code. I picked the winner of CC 44 as Nevin, mostly because his solution splits the functionality out into a separate function **Commafy** and I consider that sort of refactoring to be a very helpful example to the original writer of the code.

If Nevin could contact me on scc@accu.org to arrange his prize – unfortunately I have had bounces from the email address as supplied.

Code Critique 45

(Submissions to scc@accu.org by 1st May)

I am building a random sentence generator which will construct a sentence from four arrays containing verbs, nouns, etc. The sentence is built by using a random index for each of the arrays. There is one slight problem –- the three calls in my test program produce exactly same value! If I run the program again I get a different sentence, but again repeated three times. Of course, I want three different sentences within the same run.

I've tried to follow the code through with a debugger, but it does produce different sentences when I single step through the code. Can anyone help me out?

```
#include <string>
#include <vector>

class RandomSentence
{
public:
  RandomSentence() { sentence.resize(6); }
  void createRandomSentence();
  std::vector<std::string> & getSentence()
  { return sentence; }

private:
  std::vector<std::string> sentence;
  static std::string article[5];
  static std::string noun[5];
  static std::string verb[5];
  static std::string preposition[5];
};

#include <time.h>

void RandomSentence::createRandomSentence(){

  int randNum;

  for(int i = 0; i <= 5; i++) {
    srand(time(0));
    randNum = (rand()%5);
    switch(i){

      case 0:
        sentence[i] = article[randNum];
        break;

      case 1:
        sentence[i] = noun[randNum];
        break;

      case 2:
        sentence[i] = verb[randNum];
        break;
      case 3:
        sentence[i] = preposition[randNum];
        break;

      case 4:
        sentence[i] = article[randNum];
        break;

      case 5:
        sentence[i] = noun[randNum];
        break;
    }
  }
}

std::string RandomSentence::article[5] =
  {"the", "a", "my", "your", "his"};

std::string RandomSentence::noun[5] =
  {"pig", "cup", "phone", "TV", "letter"};

std::string RandomSentence::verb[5] =
  {"ate", "sat", "flew", "ran", "lay"};

std::string RandomSentence::preposition[5] =
  {"by", "in", "with", "over", "on"};

#include <iostream>

int main()
{
  RandomSentence rsc;
  for ( int i = 0; i < 3; i++ )
  {
    rsc.createRandomSentence();
    for ( int j = 0; j != 6; j++ )
    {
      std::cout << rsc.getSentence()[j];
      if ( j == 5 ) std::cout << std::endl;
      else std::cout << " ";
    }
  }
}
```

You can also get the current problem from the accu-general mail list (the next entry is posted around the last issue's deadline) or from the ACCU website (http://www.accu.org/journals/). This particularly helps overseas members who typically get the magazine much later than members in the UK and Europe. ■

Prizes provided by Blackwells Bookshops and Addison-Wesley

Standards Report

Lois Goldthwaite brings news from the C standard committee.

Immediately following the ACCU Conference, the international C++ standard committee (WG21) convenes for a week-long meeting at the same Paramount Oxford Hotel. As the hosts, ACCU are grateful for the generous financial assistance provided by sponsors Google and Adobe Systems. If you meet someone from these organisations at the conference, please make a personal expression of appreciation for their support of this important work. And if you want to see in person what happens at a WG21 meeting (in a nutshell: geek-speak to the limit!), please write to standards@accu.org for details.

And following the C++ committee meeting, the C standard committee (WG14) convenes its semi-annual meeting in London, April 23-27. This event is hosted by ACCU member Neil Martin of TFJ Ltd, and sponsored by accounting firm SumIT (UK) Ltd.

The focus of the WG21 meeting is on new features for C++0x – and we have committed that 'x' means '9'. That means there is a lot of work on the agenda. According to the timetable agreed last October, by October 2007 a Final Committee Draft has to be ready for voting. This will be a revised version of the ISO/IEC 14882 C++ Standard with all major features in near-final form. The Registration Document which was submitted for balloting last fall contains several 'placeholder' paragraphs promising a new feature but not listing full details in precise standardese language. By October those features need to be baked into near-final text. There will be comments from National Standards Bodies to resolve, and then the really-final text, called a Final Draft International Standard (FDIS), goes to another ballot, planned to begin after October 2008. Each voting cycle takes several months, and official publication introduces yet more delays at the end. This sounds like a lot of red tape, and it is, but ISO/IEC standards are expected to be valid for a period of years, so quality and broad consensus are important.

In order to meet the timetable commitment, WG21 has called an extra week-long meeting in July, and added a sixth day to the schedule for the October meeting. So committee members are working hard, but even if you do not attend WG21 meetings you can still contribute to make this standard the best one achievable. All working documents are available from the WG21 web page at http://www.open-std.org/jtc1/sc22/wg21/. You can post your comments on the internet news groups comp.lang.c++.moderated or comp.std.c++. Or you can submit them to your national standards body, which in the UK can be reached by writing standards@accu.org. In fact, the UK C++ panel welcomes comments from everyone, even if not resident in the UK.

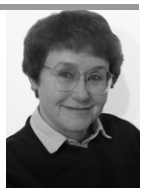
WG14 is not in the middle of an active revision of the ISO/IEC C standard, but on the agenda is a discussion of whether it is time to embark on one.

And, finally, here is another blatant plug for our sponsors:

- www.google.com
- www.adobe.com
- www.sumituk.co.uk

LOIS GOLDTHWAITE

Lois has been a professional programmer for over 20 years. She is convenor of the C++ and Posix standards panels at BSI. One of her hobbies is representing the UK at international standards meetings!
Lois can be contacted at standards@accu.org.uk



Regional Meeting

Frances Buontempo reports from the London regional meeting.

The third monthly meeting of the ACCU in London took place at Lehman Bros in Canary Wharf on the thirtieth floor, providing a spectacular view of the city. Twenty-one people made it this time, including some, soon to be, new joiners which is encouraging.

The talk was a taster of a session to come at this year's conference – 'C++: Why bother?' given by Russel Winder. It was explained that the 'Why bother?' title was intended to be ambiguous for the evening, allowing discussion of the alternatives to C++ and the benefits of C++, though the title of the conference talk may swing in one direction or the other. The aim was to get people thinking and discussing programming language choices.

A variety of languages were considered. Dynamic languages such as Python are coming to the fore now in many areas. Functional programming languages also got a mention on the way, though few present had used them for production code.

The discussion lent towards the view that C++'s importance will continue to diminish in mainstream desktop/server application areas, but there will still be a place for it for some time to come, particularly in systems and the embedded world, though C is an alternative here. The speaker, who didn't really get into the meat of what he'll be presenting at the ACCU conference, personally holds the stronger view that C++0x will probably do more to kill C++ than save it.

Specific downsides of C++ were then explored. It was suggested that, because Linux, and all its important services and libraries, are written in C, trying to write C++ against it (specifically using the STL), there is an 'Impedance mismatch' between containers, strings etc.

Mention was made of overly complicated constructs and particularly long error messages from template code that won't compile, though this was acknowledged as an implementation problem rather than something dictated by the standard. It was also noted that there are a variety of C++ implementations which vary (provocatively put as 'several C++s'), partially due to intimate relation between the language and the hardware it must run on. This has the plus side that OSes and anything hitting the hardware must be written in C or C++.

The bigger picture was briefly taking into account: language choice can sometimes be driven by business decisions. For example, would a business swap to taking on contractors who knew a specific language to get a job done rather than write in the language that legacy code is in and that their current employees know? Mention was made of COBOL still being hidden in the depths of many database calls. How do you decide the best language for a program anyway? How do you allow domain experts to get involved more easily? Are we aware of how many programmer there are (including anyone who has ever written a VBA macro)?

And then the projector shut itself down, clearly signalling we were out of time and should retreat to a nearby waterhole. ■

Bookcase

The latest roundup of book reviews.

If you want to review a book, your first port of call should be the members section of the ACCU website which contains a list of all of the books currently available. If there is something that you want to review, but can't find on there, just ask. It is possible that we can get hold of it.

After you've made your choice, email me and if the book checks out on my database, you can have it. I will instruct you from there. Remember though, if the book review is such a stinker as to be awarded the most un-glamourous "not recommended" rating, you are entitled to another book completely free.

I must thank Blackwells and Computer Bookshop for their continued support in providing us with books.

Software Development

Implementing Lean Software Development

By Mary and Tom Poppendieck, published by Addison Wesley, 276 pp, ISBN 0-321-43738-1

Reviewed by: Pete Goodliffe

Highly recommended.

The Poppendiecks' latest book sits in Addison Wesley's 'Kent Beck Signature Series' which immediately gives it a reputation to live up to. And so it does.

This is their second book on 'lean' software development, and is in many ways a follow-up to its predecessor. There is a small amount of overlap in the two books' material, but only enough to recap the old ground before the authors provide new perspectives on what it means to perform 'lean' software development.

They present a way of looking at 'agile' development in terms of the 'lean' ideas emerging from the 1980s/90s manufacturing and logistics disciplines. In doing do, the authors draw heavily from the pioneers of this movement: Toyota (or, as they were, Toyoda Automatic Loom Works). They provide insights into the lean methodology this company developed, and how it affected their entire company ethos, workflow, and their interactions with outside companies. They dig below simple observations of processes and procedures to understand the underlying motivations that made 'lean' work for Toyota.

The Poppendiecks apply this skillfully to the realm of software development.

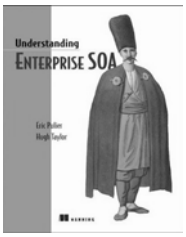
Clearly not all of the manufacturing practices would move over to our world wholesale; they provide a pragmatic application to software development. They explain how the unique mix of qualities in 'lean' development can benefit us, and how to made it all work in practice, not how not to attempt a half-baked version of 'lean' development.

The authors do not present their material as 'this is how to do it'. They provide a set of ways to think about how to apply it. Topics covered include quality issues, building healthy teams (not just 'work groups'), when and how to make the correct decisions and trade-offs, and studies on processes and workflow.

The book is practical, engaging, very well written, and clearly comes from a background of substantial experience. Throughout the book are real world stories from the authors' experience and case studies of real companies doing 'lean' in the Real World.

Each chapter is rounded of with a set of challenging questions to get you thinking about the development process your team currently uses, and how to improve it.

Understanding Enterprise SOA



by Eric Pulier and Hugh Taylor, ISBN: 1932394591, pub Manning Nov 2005

Reviewed by Peter Hammond

According to the back cover blurb, the authors are a technology evangelist and a marketing executive. It shows. There is a brief mention at the beginning that service oriented architecture is not the same as Web Services, and that there are no 'magic bullets'. It then spends the rest of the book telling us how SOA (read Web Services) is going to save us all from the evils of expensive software engineering, and even help to cure cancer (I kid you not, page 82).

Various parts of the book expose either a lack of understanding of the software engineering process, or a degree of simplification that is truly misleading. It seems to suggest that simply by exposing interfaces via a web service, components can be changed at will with no software development effort to integrate them, and that change control and software maintenance are made cheaper. It even suggests that web service components can be deployed on a different platform without porting effort (page 155). It spends several pages talking about benefits of SOA that are actually benefits of open standards, and makes no mention of how SOA is supposed to help if there does not happen to be an open standard for your application yet. Instead there is an implication that using web services standards to pass your messages means that the other side will just understand them, as if by magic.

This book may be useful if you have decided that a Web Services architecture is appropriate, and you need to convince non-technical management. If you are looking for a balanced or detailed introduction to the technology, look elsewhere. Not recommended.

Design

Designing the Obvious

Robert Hoekman Jr, New Riders, pp246, ISBN: 032145345X

Review by: Simon Sebright

The first thing that grabs one about this book is the cover. It looks like a search option on a web page, but cut down to the minimum. The subtitle is *A common sense approach to web application*

Bookshops

The following bookshops actively support ACCU (offering a post free service to UK members – if you ever have a problem with this, please let me know – I can only act on problems that you tell me about). We hope that you will give preference to them. If a bookshop in your area is willing to display ACCU publicity material or otherwise support ACCU, please let us know so they can be added to the list

- **Computer Manuals** (0121 706 6000)
  www.computer-manuals.co.uk
- **Holborn Books Ltd** (020 7831 0022)
  www.holbornbooks.co.uk
- **Blackwell's Bookshop**, Oxford (01865 792792)
  blackwells.extra@blackwell.co.uk

design. It is the result of the author's experiences in working with web applications, that is web sites which actually allow you to do something, as opposed to view people's intimate details or see pictures of their dogs.

The tone of the book is pretty informal and straightforward. It largely boils down to relatively few concepts. At the end of the day, only put in what is absolutely necessary, the idea being that this simplicity is best for the users. They don't want bells and whistles confusing them or getting in the way.

There are several examples of good and bad websites, some from the author's own endeavours, along with little stories about how features got simplified.

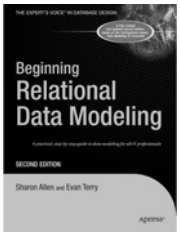
As someone getting into web application development, I found the overall message very informative and inspiring. He advocates a lot of agile-like techniques, and encourages informal processes over documentation. The best piece of advice I found was to cut down the first delivery of a system to bare essentials. Throw away all the other suggestions and then see what users really want as extra features when they use the system.

Over all I would recommend the book as an empowering mechanism for anyone getting into web application development. I am sure the messages could have been delivered in far fewer pages, but it's an easy read. One criticism – there is no technical information. For example, he mentions a 'trick' using divs to implement the expand inline pattern, but doesn't give any help or references about how to do that. Excellent index.

Beginning Relational Data Modeling, 2nd Edition

by Sharon Allen and Evan Tracy (1-59059-463-0), Apress

Reviewed by Paul Thomas



Recommended.

If you already have a technical grounding then you may find this a little light. It's a wordy introduction that goes a long way into the subject without requiring too much attention to the mathematical side. If that's just what you're after, then you could do much worse. The writing has a nice, easy style that will keep you interested. And just because the presentation has a light touch, doesn't mean there is a lack of substance.

There were some occasions when I felt more background was expected of me as certain subjects were introduced. But on the whole, it does a nice job of starting at near zero and building up. The subject matter is broad so you get a good grounding in the basics but you certainly aren't left with only a skin-deep knowledge by the end. The material is pretty much up to date too.

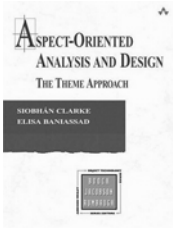
This book has a lot going for it in terms of style, but what impressed me most were the examples. They are real-world and taken to quite some level of detail. There are the usual suspects or sales and order tracking. But the main example – the one taken from concept to physical reality (and then some) – revolves around a card game. Not only does this keep you from zoning out, it helps you think of data-based systems where you wouldn't normally.

Buy it if you want something to read rather than a reference book or course text book.

Aspect-Oriented Analysis and Design: The Theme Approach

by Siobhan Clarke and Elisa Baniassad, Addison-Wesley, Object Technology Series, 366 Pages

Reviewed by Simon Sebright



This is an academic book. The authors are academics, and the contents of the book are distillations of their research. It reads academically in that it is very factual, self consistent and dry. There is a lot of UML.

They present a combination of their work into aspect-oriented techniques, which they call the theme approach. It consists of two parts – Theme/Doc and Theme/UML. They constantly refer to these as 'tools' and that they can regenerate diagrams, and extend UML, but I didn't see anything saying if or where one could acquire these tools, so I assume they are in development, which reinforces the academic nature of this.

The theme approach is to treat all concerns in the requirements as themes. Theme/Doc allows you to associate requirements with themes and identify aspect or cross-cutting themes. Theme/UML is a UML extension allowing you to design the themes in UML and combine them using various notations into a final model.

There are a couple of case studies used in the book, a mobile multi-player game, a mobile phone system and licensing. I didn't really get the feeling that these were really presenting me with full solutions, but they get the point across of how to identify aspect themes.

This is really quite leading edge stuff, and wouldn't be directly useful without the tool support they seem to have. Also, it all ends in a UML model, and then you have to create the code, making it all seem a little pointless. The idea was to have code represent the requirements more closely, but their process ends only with a model, and you have to make decisions about how that gets implemented. True, if you use an aspect-oriented language you can benefit from whatever aspects it supports.

I decided to review this to learn a bit about aspects, and I certainly did. But, I found the book somewhat repetitive. They have a style of telling you something in overview, telling you in detail and then telling you with example.

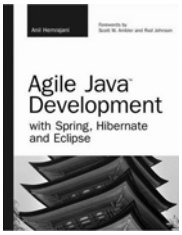
In summary, well written but quite narrow in scope.

Java

Agile Java Development – with Spring, Hibernate and Eclipse

by Anil Hemrajani, Developer's library, ISBN: 0672328968

Reviewed by: James Roberts



This book is impressively ambitious. To give a description of Java development is a full book on its own. To throw in a description of Hibernate, Spring and Eclipse at the same time sounds like a recipe for disaster. Given these reservations, I was pleasantly surprised!

None of the constituent technologies is described in a huge amount of detail. There is enough there to allow one to decide whether it is worthwhile looking further (and also a list of alternatives to each component should you decide to try out other approaches for your development).

The development approach described is XP based, and shows the development of a web-based application, right the way through initial definition of the functional requirements, through design and code and automated testing. Inevitably, given the number of topics in the book each one gets a very lightweight treatment, but this does allow a good overview of how all these technologies might be used together in a real project. The author includes some personal insights and opinions interspersed throughout the main body of the text. These were generally interesting, and added another dimension to the book.

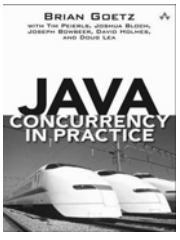
There were a few niggles. For example, why write test code which outputs `println` messages on error, why not `assert`? There was a bit too much of the 'this is the way that we do it, and the result is bound to be a massive success' for my liking, a useful chapter might have been on handling niggles and disasters that happen after the code is written.

However, on the whole I think that this book would be useful for someone who has Java coding experience (which is assumed by the author), but perhaps hasn't used one or more of the other technologies covered. Recommended.

Java Concurrency in Practice

by Bruce Goetz at al, ISBN 0321349601, Addison-Wesley

Reviewed by Paul Thomas



Highly recommended.

There are two ways of looking at this book: one is as a

guide to the `java.util.concurreny.*` packages introduced with Java 1.5, the other is as a treatise on modern multi-threaded programming in Java. The two go hand in hand, of course: the concurrency package was added as a result of JSR-133 – the Java Community Process work on threading and memory model.

If you want to get fully up to date with multithreading in Java, then you really should take the time to read this book. It covers just about every aspect of threading from the general issues of concurrency to the specifics of implementing custom synchronisers using the `AbstractQueuedSynchroniser` base class. In fact, I'd go further and say that if you are interested in concurrency at all, regardless of the language you use, then you should read this book. It's more than just teaching you how to use some Java classes. Topics such as composability of thread safe classes can apply to all modern program design. And I don't think I've come across a book that deals with testing multithreaded classes before.

It wouldn't work as a reference book, but mine will be sitting on the desk for some time. It needs to be read and re-read. If you want to flick through looking for a specific example, the examples of broken code are clearly labelled with a sick-face motif. Just a little detail, but one that could make all the difference!

As well as covering the technical details of threading, locking and memory models, the book is filled with snippets of information about the implementation of JVMs. One example is the discussion of the relative merits of explicit locks vs. implicit locking via 'synchronized'. Not only are we shown profiling data to show how the techniques scale, but we are told how work on Java 6 has drastically changed the situation for the better. The authors don't just state that atomic variables are lighter than locks – they explain in great detail why it might be so. This sort of detail makes the difference between knowing a subject and really understanding.

Spring in Action

Craig Walls and Ryan Breidenbach Manning, ISBN: 1932394354

Reviewed by: James Roberts

To cut to the chase, I am a big fan of this book. I have used it as a reference while coding, while it also clearly explains the topic at a beginner's level.

It starts by explaining how Spring can be used for Inversion of Control and Aspect oriented programming. The code examples used to illustrate the text are short enough to copy and use (although, I would have liked to see links to a web-site which I could have downloaded the examples from – just to save typing effort). The authors include enough details to ensure that the book remains useful as reference material as well as an introductory text – without bogging down the reader who is reading for the first time.

Later chapters cover topics such as integration with other technologies, such as JDBC (or Hibernate or iBatis), transaction control and the Spring MVC (model view controller) framework for the web layer. Although the authors clearly have preferred solutions for object-relational frameworks (Hibernate), presentation layer (Spring MVC) and remoting (RMI), alternatives are given, and their use explained with a fair amount of detail.

I felt that this book was well written, being both easy to understand at an introductory level, and detailed enough to be useful as a reference. The prose style is clear throughout, without skimping on detail.

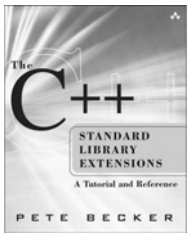
Highly recommended.

C++

The C++ Standard Library Extensions: A Tutorial and Reference

Written by Pete Becker, published by Addison Wesley

Reviewed by Anthony Williams



I had high hopes for this book, as Addison Wesley is a respected publisher of C++ books, and Pete Becker is project editor for the next edition of the C++ Standard, and has contributed a lot of time and effort into the development and publication of the Technical Report described in this book. Unfortunately, I was sorely disappointed. Subtitled *A Tutorial and Reference*, I feel that this book fails to meet either goal.

A key part of a reference book is the index, and for a reference to a library I would expect every function, class and macro to be in the index. This is not the case. For example, many of the type traits classes are not in the index, and the only reference in the index to the unordered (hash) containers by name is to the header synopsis in the appendix rather than the chapter which describes them. Not only that, but some of the entries are just wrong: the only index entry for `result_of` is for page 141, but `result_of` is not mentioned until page 142, and that is only a cross-reference to the real section, which starts on page 148.

Having the reference material interspersed amongst the descriptive text and examples makes it really hard to use this book as a tutorial. It has no narrative flow, which makes it hard to read in a linear fashion. Exercises are included at the end of each chapter, but they often require that you've read and inwardly digested the reference material, rather than following on from a nicely explained tutorial. For many functions, there is just a terse summary of its operation followed by some example code, with little in the way of descriptive text explaining what the function does, and why one might want to use it.

A final, minor note: in my copy, the print quality is really poor. Some of the text is supposed to be black on grey, but in my copy it comes out as grey on slightly-paler grey, which is really hard to read. This varies from page to page, and might just be particular to this copy; I've never had this problem with Addison Wesley books in the past.

Overall, I'm glad I didn't pay the full cover price for this book, and I don't think I'll ever refer to it again. Instead, I'll use the actual TR1 draft (available from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1836.pdf) for reference, and the boost docs/examples as tutorial notes, since just about every component from TR1 is based on a boost library.

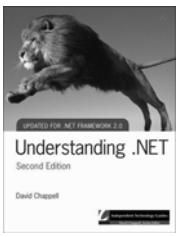
Not recommended.

.NET and Mono

Understanding .NET

by David Chappell, Addison Wesley, ISBN: 0321194047, pp317

Review by: Simon Sebright



Recommended (for the intended audience). This book is indended as a 'big-picture introduction', and largely succeeds. It doesn't go into much detail, and has but a few code snippets.

It's broken into chapters on CLR, Languages, Class Library, ASP, ADO and Distributed Applications. Basically, if you want to do anything regarding .net, be it architect or lead a project, do the programming, etc., you need to be familiar with all the material in this book.

It was well-written and quite readable, and being familiar with .net already, it was a case of nodding along, but with the occasional twigging and new angles on things.

Throughout the book are little extra sidebars where the author considers the pros and cons of .net and other things, mostly java. They are largely unconclusive pieces, saying it's a good thing that there is competition between java and .net.

It would be a good place to start to get to grips with .net, but probably of little value to the already-initiated, although if a copy is available, it might pay to ream through it to keep you in touch.

General Technology

How cool stuff works

by Ted Smart, published by Dorling Kindersley, ISBN 1405308370

Reviewed by Ian Bruntlett.

To coin a phrase, this is a

'popular technology' book. It is hard backed, A4 sized and runs to 256 pages, lushly illustrated. It splits technology in to 6 chapters – Connect, Play, Live, Move, Work and Survive. Each section is split up into two page spreads illustrating a particular invention.

For example the 'Connect' chapter covers the following inventions – Microchip, Mobile Phone, Fibre optics, Digital Radio, LCD TV, Toys Gallery, Voice Recognition, Pet translator, Iris scan, Neon, Links gallery, Internet, Video link, Satellite and 'What Next?'. The other 5 chapters are just as interesting.

To complement the main text, there is a slim reference section with a timeline – showing when certain things were invented, a 'groundbreakers' section which highlights the work of key inventors and a 'techno terms' section which acts as a glossary.

I regard this book as recommended reading for anyone interested in technology. If you like this book then read *Tomorrow's People* by Susan Greenfield.

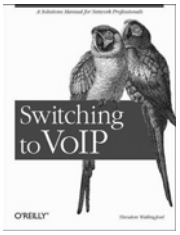
Conclusion: Recommended.

Switching to VoIP

by Ted Wallingford, O'Reilly, ISBN 0-596-00868-6, pp477

Reviewed by Mark Easterbrook

Not recommended.



VoIP is a technology that has been touted as the 'Next Big Thing' for at least 10 years now. All but a few slow cases in the marketing department have now realised that it is a technology with a slow and gradual take-up and is never going to be a gold-rush. There are three main areas where VoIP has made some in-roads: technology savy computer users taking advantage of cheap or even free long distance calls over the Internet, companies replacing ageing voice or data infrastructure using the upgrade as an excuse to converge their communication and data networks, and large telecom companies replacing their core networks. Add to this the companies developing and manufacturing VoIP equipment and we have four diverse domains with very different requirements, constraints, budgets, and understanding.

From this fragmented VoIP world I have failed to identify the target audience for this book. The large telecom companies and equipment manufacturers are too close to the bleeding edge to benefit from printed source material that is at best several years out of date due to the long lead times in book publishing.

Most of the book's hands-on technical content concerns the Linux-based Asterisk PBX (Private Branch Exchange) application and covers installation, configuring and programming. The interface equipment required is only available through specialist suppliers and the small market makes it relatively expensive, limiting its appeal to a few hard-core Linux fans. The technical detail is too low level for most commercial organisations and the sections aimed at this audience, such as RoI estimates, are too few and far between to be worth buying the book for.

The author does provide a good grounding in VoIP concepts, and even has several chapters that are good tutorial material. Unfortunately this is interspersed with the Linux-Asterisk specific detail so that using the book as an introduction to VoIP is difficult. The material is also very US-centric limiting its usefulness to those not working in areas of the world using ANSI telecoms standards, this despite the author claiming otherwise in the introduction.

Overall this book tries to do too many things on too many levels and fails to achieve any of them. The author is obviously very knowledgeable on the subject and the content matter of the book is good. It is just that that it does not all belong together in the same tome. The Linux-Asterisk part needs its own dedicated book, as I believe it has. The tutorial on VoIP needs to consider a non-US audience, and to drop the advocacy, resulting in a much slimmer volume. Finally, there is a need, only just touched on here, for a good guide for the SME thinking of moving to a VoIP and covering the solutions available off-the-shelf and not DIY.

Technology Tidbits by Tim Penhey

I'd like to take just a few words to mention a version control system (VCS) that many of you have probably never heard of. I say many because I know some ACCU folks who are using it. The VCS in question here is called Bazaar [1]. Bazaar is an open source VCS written in python. Bazaar, like most well liked VCSs, has a three letter acronym 'bzr'. The big difference between Bazaar and other popular open source VCS alternatives like CVS and Subversion is that Bazaar is a decentralised VCS[2].

An advantage of having a decentralised VCS is there is no requirement to have a 'central repository', and creating a new bazaar branch is exceedingly simple [3]. Bazaar is also optimised for branching and merging, to the extent that many people create branches for bug fixes.

From a technology point of view, one of my favourite things about Bazaar is that the main developers spent a lot of time working on the underlying model, deciding that it is better to get the model right first, and then work on optimising for speed. By the time that this is in print, the current released version of Bazaar will be 0.15.

Bazaar has a concept of a repository. In this repository is a set of revisions. Revisions are created when the user 'commits' changes. Effectively a revision has one or more parent revisions, and the changes that happened in that revision. Each revision also has a unique identifier. A branch can be thought of as a view into the repository. A branch is a directed acyclic graph (DAG) of revisions, where the left-most traversal of the graph from the latest revision up is called the revision history and gives revision numbers. Other revisions that are part of the branch but not part of the history are considered the ancestry. Working with branches in this manner make branching and merging an exercise in graph theory.

You can work with Bazaar in a number of different ways [4], but the way that we use it at work is like this:

- There is a recognised mainline development branch
- When you are starting a piece of work you branch from this in your local repository
- You work and commit to the local repository
- When you have finished, you push your work to a central server and your code gets reviewed
- When the code has passed review it can go through the automated landing procedure. This is a process that gets your branch, merges it with a copy of the development branch, and if it merges cleanly the full test suite is run. If and only if it passes is your code merged into the mainline development branch.

I find that this works exceedingly well. I hope you decide to check it out. I think it is pretty cool.

References

1. http://bazaar-vcs.org
2. http://bazaar-vcs.org/TheBigPicture
3. http://bazaar-vcs.org/QuickHackingWithBzr
4. http://bazaar-vcs.org/Workflows

View From The Chair

Jez Higgins
chair@accu.org

My first job after leaving university was at another university. It was an interesting and rather fun experience for all kinds of reasons, but particularly because I worked in a research department. Even though I was on the technical rather than academic staff, I was encouraged to pursue my interests, even if not directly applicable to my work, there was a budget to provide the hardware and software tools I needed, I had access to a pretty well-stocked library which also subscribed to a number of journals and magazines, and there were lots of interesting people to talk to. I probably didn't take as much as advantage as I might have, but I learned an awful, awful lot in a very short time.

Then I start doing 'proper' jobs for 'proper' software companies. Initially, I was keen to work for a big consultancy. All those people, doing all that stuff. It must be great! Think what I could learn. The reality was, and you can probably see where this is going, rather different. There were lots of people, doing a certain amount of stuff but not many people seemed to be having much fun, and the work wasn't particularly interesting. More dramatically, there was no culture of learning. Very few people had any books on there desk, and the nearest I got to a technical journal was a week-old copy of Computing.

This is part of the my 'career narrative', and part of the reason why I joined ACCU. It was a place (the conference) and a thing (the journals) where people where interested and interesting. While I won't claim that every article as left me swooning with unalloyed joy or that every conference session leaves me breathless with excitement, ACCU has been and remains an important part of my social and professional programming life. Not every one will feel quite the same, but I hope it's a common sentiment.

In my first View From, I talked about wanting to extend ACCU's reach, that we can't be the only 'programmers who care', and that there must be more of us out there. I've been Chair for nearly a year now, and while there are still no levers[1], I've got a handle on how things work. Assuming the membership re-elects me at the AGM, I want to spend the next year really trying to get on with growing ACCU's membership. It'll be good for us already here, and good for those we find, I have no doubt. Efforts like the London meetings and opening up Overload on the website will really start to take off, I think, but there's clearly more that we can do. Allan and I will be holding a 'growing the membership' BOF session at the conference. Do come along if you can.

[1]In-jokery abounds, see CVu 18-3

Membership Report

David Hodge (retiring)
accumembership@accu.org

I am retiring from this post at the AGM in April, having done it for 10 years.

I leave with the membership at around 900 which is almost the same as when I started. During that time the hishest we had was 1100, the lowest 820.

I wish you all well.

The job of Membership Secretary is being taken over by Mick Brooks who has been doing large parts of the job since November and I feel confident that he will be able to manage the web based system as I did the PC based system.

Mick Brooks
accumembership@accu.org

As David mentions, I've been working with him in the last few months with plans to take over the membership job at the AGM (hopefully I'll be elected – please vote for me!). I want to thank David, not only for the guidance and support he has given to me as I've been getting to grips with the job, but also for all the work he has put in on behalf of the membership in the previous ten years.

David will leave us having specified and helped implement and test the new membership system, which works closely with the new website. The rest of the credit belongs to Tim Pushman, who's built and integrated the system while ensuring that new member sign-ups were working during the busy period just before the conference early-bird discount ran out. The new system means that, when you're logged into the website, you can now access new options on your Account Information page. These allow you to manage your mailing and handbook settings, and will let you renew your membership at the appropriate time. If you haven't logged in recently, please login and have a look. Any member who's forgotten how to access the website, or is having any other trouble with these features shouldn't hesitate to contact me.

Officer Without Portfolio

Allan Kelly

When I first joined the ACCU committee, our meetings seem to ramble on forever. Most of the time it didn't feel like we were actually doing anything. Instead we spent our time trying to work out who had the cheque book, why the journals were late (again) and isn't the website awful. Come the 5.30pm deadline, when we had to run for our trains, we just cut scope and the things further down the meeting agenda were dropped.

Well that was then and this is now. This is the New ACCU. Meetings deal with business, things get done, the website is fixed (and outsourced), we know who has the cheque book, we spend our time discussion things like membership growth and local chapters, and for the first time in memory the accounts will be ready for the AGM. This year the accounts will be prepared by professional accountants and in the near future we will probably outsource the whole job, freeing the treasurer from administrative trivia.

For the last two years I've concentrated on getting the website sorted. The new website has been live for a year now and it still looks fresh. Like any good development changes are now incremental. David Hodge, Mick Brooks and Tim Pushman have done a great job of creating an online membership system that should be live by the time you read this. Tony Barrett-Powell now has Overload available as individual HTML articles (hopefully live by the time you read this) and the book review system has been re-invigorated.

The website is carrying a few GoogleAds at the moment. These do not bring in lots of revenue but they do enough to pay for our site hosting. The book database has not been such a great financial success but with a new push on book reviews we hope to turn this around. Before you next buy a book please think about going to Amazon or Blackwells (UK or USA) via the ACCU website and we'll get a small cut. Jeff Bezos won't miss the money but every little helps the ACCU.

It all starts to look like the website is done. We haven't moved the mailing lists over yet but I will not bore you with the details. We are going to try again, hopefully they will have moved by the summer. Then it will be big red switch time for our trusty server, Brian. (Hopefully the US Chapter will also find the time to merge into the new site soon.)

So what next?

ACCU looks more like a professional organization. Still we suffer from a lack of volunteers. We need people to write journal pieces, to edit our journals, run the committee and fill the empty posts like Advertising and Publicity officer. The solution is quite simple: we need to grow our membership.

If we can grow our membership many of our existing problems will simply go away. With more people to fill the posts the committee will function better, with more authors writing for the journals the material shortage will disappear, with more people reading the journals our

advertising revenue will increase. We don't need to (and probably don't want to) grow very big. Doubling our membership from 850 today to 1700 should be enough.

Personally I don't think this is a difficult problem. This is equivalent of every ACCU member recruiting one new member. Filling the post of publicity officer is going to be key. A few press releases, a few more mentions on The Register, Slashdot and similar will soon bring in the members.

Creating local chapters will help too. Things are happening here too. A London chapter of the ACCU is slowly emerging, all credit to Paul Grenyer here; Bristol ACCU'ers have tried their hand at the odd meeting and the Cambridge boys are thinking about following the lead. For all this we have to thank Reg Charney for his inspiring speech at the ACCU conference last year.

There are plenty of other opportunities for local chapters too. Statistically Reading/Slough should be a prime candidate for a chapter too.

Hopefully we will see some more overseas chapters too. Switzerland, and specifically Zurich, looks like a great opportunity.

The ACCU is looking in great shape just now. I hope to be elected to the ACCU committee for another year at the AGM next month. I also hope you will all join the committee and me in growing the ACCU membership over the coming year.

If you read something in C Vu that you particularly enjoy, you disagree with or that has just made you think, why not put pen to paper (or finger to keyboard) and tell us about it?