overload 92

AUGUST 2009 £3

Quality Matters

We start a new series, considering the idea of software quality

I'll Have To Parse

Parsers are a fundamental programming tool. We see how to write one.

Generation, Handling and Management of Errors

A pattern language for error management, to ensure that your code is robust

No "Concepts" in C++0x

The C++ Standards Committee has voted to remove concepts from C++0x. Bjarne Stroustrup explains the reasons, outlines the controversy, and points out that the sky is not falling on C++.

A magazine of ACCU

ISSN 1354-3172

OVERLOAD 92

August 2009

ISSN 1354-3172

Editor

Ric Parkin
overload@accu.org

Advisors

Phil Bass
phil@stoneymanor.demon.co.uk

Richard Blundell
richard.blundell@gmail.com

Simon Farnsworth
simon@farnz.co.uk

Matthew Jones
m@badcrumble.net

Alistair McDonald
alistair@inrevo.com

Roger Orr
rogero@howzatt.demon.co.uk

Simon Sebright
simon.sebright@ubs.com

Paul Thomas
pthomas@spongelava.com

Anthony Williams
anthony.ajw@gmail.com

Advertising enquiries

ads@accu.org

Cover art and design

Pete Goodliffe
pete@goodliffe.net

Copy deadlines

All articles intended for publication in Overload 93 should be submitted by 1st September 2009 and for Overload 94 by 1st November 2009.

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

ACCU

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

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

Moments in History

Technology shapes our world. Ric Parkin looks back at 40 years of change.

I start to write this on the morning of Thursday 16th July 2009, exactly 40 years after Apollo 11 was launched, and I'll be writing the rest of this over the days of the mission. The TV schedules are filled with documentaries talking about this stunning technical achievement that united the world in awe...but how did the world experience it, and what sort of world was it after all? 40 years is a long time in technology, and it was a very different place.

Some amateurs actually tracked the craft using simple optical equipment (it wasn't that hard to spot the dot of light of the craft, at least until it got too close to the glare of the moon itself) [Keel]. Everyone else had to follow events via the official communications channels – photo and film cameras for bringing back high definition images, and the direct feeds: radio – think of hearing 'The Eagle has landed' in the control room – and television. The thing most people seem to remember was watching it on the television, in a massive global event with a live audience of an estimated 600 million. Interestingly, the transmitted feed was in quite high definition, but was not compatible with the terrestrial networks. In order to convert and broadcast as soon as possible, they did the only thing they could do without powerful computers to do it on the fly – they showed the feed on a compatible monitor, and pointed a TV camera at it! [parkes] This is why the live footage we know is so poor quality. Recently, tapes of the original feed recorded at the Australian down-link have been recovered, and much better quality versions have been shown. There have even been new photographs taken of the landing sites by the Lunar Reconnaissance Orbiter, with such resolution you can actually see the landers and their shadows [LRO]

It wasn't the first such televisual event. In 1953 the coronation of Elizabeth II was shown in the biggest live broadcast by the BBC up until then, an event where many people bought one those newfangled televisions – which cost a lot of money in those days – and the neighbours came round to watch it live, giving an estimated audience of 20 million. And even earlier, broadcasting such a global spectacle was used for more overt propaganda purposes at the 1936 Olympics [earlytelevision].

But why am I mentioning such a disparate group of events? New technologies – in this case, mainly television – had enabled a huge amount of people to share in an event en masse, bringing them all together, often in ways that were unprecedented and unexpected. Such events can even change the way we think of ourselves: consider the impact of such famous photographs as that of Earthrise taken from Apollo 8 [Nasa] – a fragile blue ball lost in a huge dark cosmos.

The technologies in use in Apollo 11 are remarkably different to what we would expect nowadays. Mostly systems were mechanical, hydraulic, or electrical. They did use computers, but of such primitiveness that we would hardly recognise them as much beyond a giveaway calculator nowadays. [Apollo Computer]. Since then the power and complexity of computing technology has increaded greatly, although the same basic architectural ideas are still in use.

It's not the only time that new technologies have changed the way we interact. Even further back we have the cliche of the family sitting around the wireless, whether listening to The Archers, or to get the news during the War; the telephone and before that the telegraph, which allowed really fast communications across large distances, arguably a major influence on creating modern America. And even further back, the invention of the Penny Post (after an analysis by Charles Babbage), newspapers, the printing press, and writing itself.

At around the same time as Apollo, the first early networks were being developed at DARPA. In fact, October 29th will be the 40th anniversary of the first link in ARPANET, which would eventually become the Internet we know today. I think this too was an important moment when things changed: you no longer had to have everything on your computer locally, but could distribute your computing resources in a much more flexible way. And development continues, from the introduction of IPv6, or a new protocol designed to work with the time lags in deep space [DTN]

But of course the Internet is only a conduit – it's what you do with it that makes things interesting, and that tends to fall into two discrete types. The first is a distributed information storage and retrieval service. Early uses including circulating and depositing academic papers, but has grown over the years to include services like Gopher [Gopher] and the HTTP protocol and HTML that made the Web, which

Ric Parkin has been programming professionally for around 20 years, mostly in C++, for a range of companies from tiny startups to international corporations. Since joining ACCU in 2000, he's left a trail of new members behind him. He can be contacted at ric.parkin@gmail.com.

led to a proliferation of services such as Wikipedia, YouTube, photo hosting web sites, and storing documents online 'in the cloud'. I myself use the latter a lot in the role of editor – I have a status spreadsheet to keep track of what articles I have, what stage they're in, and what I need to do. Then I can access it and keep it up to date from any computer, as soon as an article or review feedback comes in.

The second is communication. Email is an obvious application. In fact it predated ARPANET, being created around 1965 as a way for multiple users of a time-sharing mainframe computer to communicate, even if there weren't around at the same time. Apparently it was used a lot when the original Internet protocols were being developed, allowing widely separated parties to collaborate much faster. It is a fantastic tool for non-immediate communication, where you want to send something even if the other party is not currently there, for them to read later. Looked at in this way it's not surprising that mobile phone SMS took off so well – it's a cheap, quick version of email.

If you wanted a real-time conversation, there are a host of chat protocols, such as IRC and the various instant messaging services, which all have the immediacy of a phone call, and of course VoIP services such as Skype reproduce a voice call itself, with added features such as sending files, and video calls.

As well as the different latencies, there are also the various multiplicities of communication. A one-to-one channel is like a phone call or personal email. A one-to-many is like a broadcast, such as publishing on the web, or services such as Twitter, or YouTube. A many-to-many mode is ideal for networks such as a social group where everyone can talk to everyone else, reminicent of a conversion in a pub with a big group of people. Email lists used like that were an early version of a social networking. Newsgroups can also have a similar function, even though they can be used like a notice board. They do look rather old-fashioned now, what with all the web-based networking sites allowing all manner of plugins and media. These are so popular and easy to do stuff, that they allow a group of people to quickly organise and affect the world, whether it was the campaign to get Wispa bars reintroduced, a speed up of word-of-mouth recommendations that can have a real effect [BBC], to the use of Twitter to export news of the Iranian presidential elections, despite curbs on traditional media. The effect of this on governments was acknowledged recently by the Prime Minister at an appearance at the TED Global conference in Oxford [TED]

With such a range of communications channels, it is now possible to organise people into teams where geographic location is much less important. This seems to be really prevalent in the IT industry, perhaps because we have to use the computers that make this work anyway, and we are more aware of the new technologies and so are the early adopters. It is not uncommon for a project to involve people in many locations and timezones, all using various communication technologies to collaborate. This really makes sense when you think of the process of developing things like software – a large amount of the time and effort is to decide what to develop by talking to the relevant people, then deciding how to split the work up so that multiple people can work on it, then having to work out how to integrate the seperate parts, and then verify that what has been produced does in fact meet the requirements. As each of these stages involves people with different skills, knowledge, and opinions, the need for communication is clear. In many failing projects (or just the ones which are horrible to be in), quite often a major issue is when the communication fails.

As for the future, what sort of changes could we look forward to? Some trends are obvious – more mobile 'always on' computers connected to a fast pervasive network, whether it be done by making phones more powerful or netbooks smaller. Location and orientation sensitive phones and software are becoming common, leading to some interesting applications as well as potential privacy concerns. The shift to mobility will highlight the problems of too-small screens and fiddly input devices. There are already systems that are trying to solve these issues, but it is too early to know which will eventually succeed. The future starts today.

References

[Apollo Computer] http://en.wikipedia.org/wiki/Apollo_Guidance_Computer

[BBC] http://www.bbc.co.uk/blogs/technology/2009/07/bruno_and_bonos_box_office_blu.html

[DTN] http://www.nasa.gov/home/hqnews/2008/nov/HQ_08-298_Deep_space_internet.html

[earlytelevision] http://www.earlytelevision.org/1936_olympics.html

[Gopher] http://en.wikipedia.org/wiki/Gopher_(protocol)

[Keel] http://www.astr.ua.edu/keel/space/apollo.html

[LRO] http://www.nasa.gov/mission_pages/LRO/multimedia/lroimages/apollosites.html

[Nasa] http://commons.wikimedia.org/wiki/File:NASA-Apollo8-Dec24-Earthrise.jpg

[parkes] http://www.parkes.atnf.csiro.au/news_events/apollo11/tv_from_moon.html

[TED] http://news.bbc.co.uk/1/hi/technology/8161650.stm

I Think I'll Parse

A parser is a fundamental tool in software development. Stuart Golodetz looks at how you might tackle writing one.

In my opinion, writing a parser manually is something everyone should do once. Any less than that, and you'd be missing out on an interesting bit of coding; any more, and you'd make the efforts put into Lex and Yacc [LexYacc] to save you the time and trouble (and the risk of making silly mistakes) look like a waste. (Lex and Yacc take lexical and grammar rules as input, respectively, and produce code in a language such as C to do the actual lexing and parsing. Modifying the input rules is substantially easier than modifying actual code, hence the reductions in necessary time and errors.)

Accordingly, in this article I'm going to discuss how you go about writing a simple parser for XML in C++ (without worrying too much about adhering 100% to the XML specification, since that's not the primary focus here). To more experienced readers, this may seem quite simple: in that case, I very much welcome comments/bug reports! To anyone who has never done any parsing before, hopefully this article will provide a useful introduction to the topic.

The plan

Our goal here is to turn a source XML file into an abstract syntax tree, or AST (see Listing 1). Once we have the AST, information from it (and thus from the file from which it was created) can be easily extracted for use in other code. The whole process is a (very) well-studied problem in computer science, partly because it forms part of the front-end of the compilation process [DragonBook]. (So this is exactly the sort of thing that happens behind the scenes when you compile your programs.)

```
// Source File:

<article type="short">
  <para text="Not much" font="Arial"/>
</article>

// (Lexical Analysis)
// Token Sequence:

LBRACKET, IDENT("article"), IDENT("type"),
EQUALS, VALUE("short"), RBRACKET, LBRACKET,
IDENT("para"), ..., LSLASH, IDENT("article"),
RBRACKET

// (Parsing)

Abstract Syntax Tree:
XMLElement article [type="short"]
  XMLElement para [text="Not much", font="Arial"]
```

Listing 1

Stuart Golodetz has been programming for 13 years and is studying for a computing doctorate at Oxford University. His current work is on the automatic segmentation of abdominal CT scans. He can be contacted at stuart.golodetz@comlab.ox.ac.uk

As Listing 1 shows, there are two key steps to the process: lexical analysis, and parsing itself. Lexical analysis (lexing, for short) is the process of converting a source file to a sequence of tokens to be consumed by the parser. For instance, the text `<article type="short">` could perhaps be lexed as the token sequence `[LBRACKET, IDENT("article"), EQUALS, VALUE("short"), RBRACKET]`. Note that data can be associated with each token, e.g. an identifier token can carry around the name of the identifier. Parsing is the process of turning a sequence of tokens into an AST.

It is important to note that the two steps can run concurrently. The parser generally doesn't need to see the entire token sequence at once. Rather, a pipeline approach is used. The parser queries the lexer when it wants the next token, and the lexer reads the source file as necessary in order to produce it. The advantage of this approach is that fewer tokens need to be held in memory at any one time; additionally, it makes the local decision-making nature of the parser explicit.

Lexing XML

The first step when building a lexer is to decide what types of token it should output. If you're lucky in having some sort of formal specification for the language to hand, this may already have been done for you; if not, you'll have to invent one by looking at representative example source files and deducing the structure of the language. In this instance, we'll do the latter. The source file in Listing 1 is representative of the XML files we want to support. With a bit of creativity, it doesn't take much effort to come up with the possible token list shown in Table 1.

The next question is how to convert the source into tokens. This is generally done using a finite state machine: we read characters from the source and change states until we're sure we've read a token, at which point we yield it to the parser. A certain amount of lookahead is sometimes required for this: for instance, when we read a `<` here, we don't know whether it's an `LBRACKET` token, or the first character of an `LSLASH`. To find out, we have to read in an additional character from the source file and check whether it's a `/`. If it is, we yield an `LSLASH` token; if not, we

Token Type	Regular Expression
EQUALS	=
IDENT	[A-Za-z0-9.][A-Za-z0-9._]*
LBRACKET	<
LSLASH	</
RBRACKET	>
RSLASH	/>
VALUE	".*"

Table 1

Certain details of the lexer are difficult to show in a finite state machine graph

yield an **LBRACKET** and push whatever character we actually read onto a lookahead list: it's the first character of the next token, so we can't simply discard it.

The design of a finite state machine for the XML lexer is shown in Figure 1. The key things to note are the extra states needed to handle things like **RSLASH** and **VALUE** tokens. For example, when reading an **RSLASH** token, we read the **/** first and end up in the **HALF RSLASH** state. If we then see a **>**, we've read a whole **RSLASH** token and can yield it; if we see anything else, there's an error in the source file. Note that the **"** self-transition from **HALF VALUE** just means 'keep reading characters until we encounter a " or the end of the file, denoted **<eof>**'.

Certain details of the lexer are difficult to show in a finite state machine graph. In particular, the data payloads yielded for **IDENT** and **VALUE** tokens are not shown. Also, what happens when we reach the end of the file whilst reading a token? If the token could be finished (for instance, we encounter **<eof>** whilst in the **IDENT** state), then we yield the token and set the state to **EOF**. If the token definitely isn't finished, we instead transition to the **BAD** state (and thereby throw an exception). The details can be seen in the code in Listing 2.

Writing the parser

Having written the lexical analyser, we can now turn our attentions to parsing proper. The grammar for the language we are trying to parse is shown in Listing 3. It's worth noting that the opening and closing tags for a composite element must have the same identifier, so the grammar is not context-free. Furthermore, as written, the grammar accepts documents with multiple root elements (this is not standard XML, but it will make the parser more flexible).

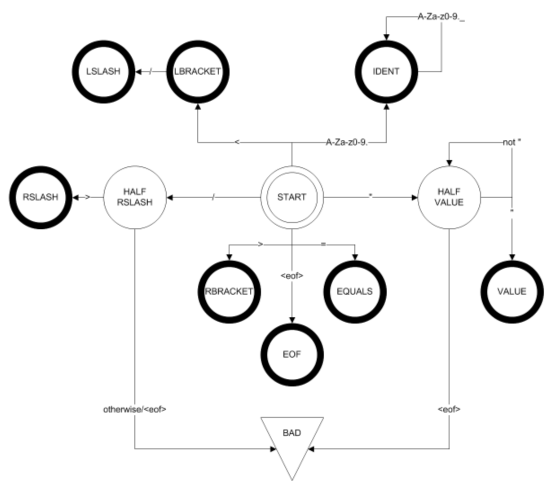

Figure 1

```
XMLToken_Ptr XMLLexer::next_token()
{
  if(m_state == LEX_EOF) return XMLToken_Ptr();
  else m_state = LEX_START;
  std::string value;

  for(;;)
  {
    switch(m_state)
    {
      case LEX_START:
      {
        unsigned char c = next_char();
        if(m_eof) { m_state = LEX_EOF; }
        else if(c == '=') {
           m_state = LEX_EQUALS; }
        else if(c == '/') {
           m_state = LEX_HALF_RSLASH; }
        else if(c == '"') {
           m_state = LEX_HALF_VALUE; }
        else if(c == '<') {
           m_state = LEX_LBRACKET; }
        else if(c == '>') {
           m_state = LEX_RBRACKET; }
        else if(isalpha(c) || isdigit(c) ||
           c == '.') { m_state = LEX_IDENT;
           value += c; }
        break;
      }
      case LEX_EOF:
      {
        return XMLToken_Ptr();
      }
      case LEX_EQUALS:
      {
        return XMLToken_Ptr(
           new XMLToken(XMLT_EQUALS, ""));
      }
      case LEX_HALF_RSLASH:
      {
        unsigned char c = next_char();
        if(m_eof) {
           m_state = LEX_BAD; value = ">"; }
        else if(c == '>') {
           m_state = LEX_RSLASH; }
        else { m_state = LEX_BAD; value = ">"; }
        break;
      }
      case LEX_HALF_VALUE:
      {
        unsigned char c = next_char();
```

Listing 2

```
        if(m_eof) {
          m_state = LEX_BAD; value = "\""; }
        else if(c == '"') { m_state = LEX_VALUE; }
        else { value += c; }
        break;
      }
      case LEX_IDENT:
      {
        unsigned char c = next_char();
        if(m_eof)
        {
          m_state = LEX_EOF;
          return XMLToken_Ptr(new XMLToken(
             XMLT_IDENT,value));
        }
        else if(isalpha(c) || isdigit(c) ||
           c == '.' || c == '_')
        {
          value += c;
        }
        else
        {
          m_lookahead.push_back(c);
          return XMLToken_Ptr(new XMLToken(
             XMLT_IDENT, value));
        }
        break;
      }
      case LEX_LBRACKET:
      {
        unsigned char c = next_char();
        if(m_eof)
        {
          m_state = LEX_EOF;
          return XMLToken_Ptr(new XMLToken(
             XMLT_LBRACKET, ""));
        }
        else if(c == '/')
        {
          m_state = LEX_LSLASH;
        }
        else
        {
          m_lookahead.push_back(c);
          return XMLToken_Ptr(new XMLToken(
             XMLT_LBRACKET, ""));
        }
        break;
      }
      case LEX_LSLASH:
      {
        return XMLToken_Ptr(new XMLToken(
           XMLT_LSLASH, ""));
      }
      case LEX_RBRACKET:
      {
        return XMLToken_Ptr(new XMLToken(
           XMLT_RBRACKET, ""));
      }
      case LEX_RSLASH:
      {
        return XMLToken_Ptr(new XMLToken(
           XMLT_RSLASH, ""));
      }
      case LEX_VALUE:
      {
        return XMLToken_Ptr(new XMLToken(
           XMLT_VALUE, value));
      }
```

Listing 2 (cont'd)

```
      default:  // case LEX_BAD
      {
        throw Exception("Error: Expected " +
value);
      }
    }
  }
}
unsigned char XMLLexer::next_char()
{
  if(m_lookahead.empty())
  {
    unsigned char c = m_is.get();
    if(m_is.eof()) m_eof = true;
    return c;
  }
  else
  {
    unsigned char c = m_lookahead.front();
    m_lookahead.pop_front();
    m_eof = false;
    return c;
  }
}
```

Listing 2 (cont'd)

```
CompositeElement -> LBRACKET IDENT(name) Params
RBRACKET Elements LSLASH IDENT(name) RBRACKET
Document -> Elements
Element -> SimpleElement | CompositeElement
Elements -> <empty> | Element Elements
Param -> IDENT EQUALS VALUE
Params -> <empty> | Param Params
SimpleElement -> LBRACKET IDENT Params RSLASH
```

Listing 3

We'll write a recursive descent-style parser here for simplicity. The top-level parsing function (that for documents) is shown in Listing 4. It creates the root node of the AST and then parses a sequence of XML elements and adds them as children of the root.

The function to parse elements (see Listing 5) reads in elements while there are any remaining, and adds them to a list. This list is then returned.

The remaining parsing function (see Listing 6), which reads in an individual XML element, is the real core of the parser. It starts by trying to read in the next token. If there isn't a next token, then there's no element to parse, so we (effectively) return **`NULL`**; if there is, but it's not **`<`** (the opening token for an element), then we add whatever token we actually did read to the lookahead list and also return **`NULL`**. This happens when we reach the end tag of the enclosing XML element, e.g. if we read **`<article><para/></article>`**, this will happen when we reach the **`LSLASH`** token at the start of **`</article>`**.

```
XMLElement_CPtr XMLParser::parse()
{
  XMLElement_Ptr root(new XMLElement("<root>"));
  std::list<XMLElement_Ptr> children =
     parse_elements();
  for(std::list<XMLElement_Ptr>::
     const_iterator it=children.begin(),
     iend=children.end(); it!=iend; ++it)
  {
    root->add_child(*it);
  }
  return root;
}
```

Listing 4

```
std::list<XMLElement_Ptr>
XMLParser::parse_elements()
{
  std::list<XMLElement_Ptr> elements;
  XMLElement_Ptr element;
  while(element = parse_element())
  {
    elements.push_back(element);
  }
  return elements;
}
```

Listing 5

Assuming we're actually parsing an element, the next step is to read in the element identifier (e.g. article). To read in any parameters, we then read the next token to see whether it's an **IDENT**: if so, we keep reading to find the rest of the parameter and iterate. Note that any erroneous tokens we might encounter are handled using **read_checked_token()**, which will throw an exception if the token we read is of the wrong type.

```
XMLElement_Ptr XMLParser::parse_element()
{
  XMLToken_Ptr token;
  token = read_token();

  if(!token)
  {
    // If there are no tokens left, we're done.
    return XMLElement_Ptr();
  }

  if(token->type() != XMLT_LBRACKET)
  {
    // If the token isn't '<', we're reading
    // something other than an element.
    m_lookahead.push_back(token);
    return XMLElement_Ptr();
  }
  token = read_checked_token(XMLT_IDENT);

  XMLElement_Ptr element(new XMLElement(
     token->value()));
  token = read_token();

  while(token && token->type() == XMLT_IDENT)
  // while there are attributes to be processed
  {
    std::string attribName = token->value();
    read_checked_token(XMLT_EQUALS);
    token = read_checked_token(XMLT_VALUE);
    std::string attribValue = token->value();
    element->set_attribute(attribName,
       attribValue);
    token = read_token();
  }
  if(!token) throw Exception(
     "Token unexpectedly missing");

  switch(token->type())
  {
    case XMLT_RBRACKET:
    {
      // This element has sub-elements, so parse
      // them recursively and add them to the
      // current element.
```

Listing 6

```
      std::list<XMLElement_Ptr> children =
         parse_elements();
     for(std::list<XMLElement_Ptr>::
        const_iterator it=children.begin(),
        iend=children.end(); it!=iend; ++it)
      {
        element->add_child(*it);
      }

      // Read the element closing tag.
      read_checked_token(XMLT_LSLASH);
      token = read_checked_token(XMLT_IDENT);
      if(token->value() != element->name()) throw
         Exception("Mismatched element tags:
         expected " + element->name() + " not "
         + token->value());
      read_checked_token(XMLT_RBRACKET);
      break;
    }
    case XMLT_RSLASH:
    {
      // The element is complete, so just break
      // and return it.
      break;
    }
    default:
    {
      throw Exception("Unexpected token type");
    }
  }
  return element;
}

void XMLParser::check_token_and_type(
   const XMLToken_Ptr& token,
   XMLTokenType expectedType)
{
  if(!token)
  {
    throw Exception("Token unexpectedly missing");
  }
  if(token->type() != expectedType)
  {
    throw Exception("Unexpected token type");
  }
}

XMLToken_Ptr XMLParser::read_checked_token(
   XMLTokenType expectedType)
{
  XMLToken_Ptr token = read_token();
  check_token_and_type(token, expectedType);
  return token;
}

XMLToken_Ptr XMLParser::read_token()
{
  if(m_lookahead.empty())
  {
    return m_lexer->next_token();
  }
  else
  {
    XMLToken_Ptr token = m_lookahead.front();
    m_lookahead.pop_front();
    return token;
  }
}
```

Listing 6 (cont'd)

```
typedef shared_ptr<class XMLElement>
   XMLElement_Ptr;
typedef shared_ptr<const class XMLElement>
   XMLElement_CPtr;

class XMLElement
{
private:
  typedef std::map<std::string,
     std::string> AttribMap;
  typedef AttribMap::const_iterator AttribCIter;
  typedef std::map<std::string,
     std::vector<XMLElement_CPtr> > ChildrenMap;
  typedef ChildrenMap::const_iterator
ChildrenCIter;

private:
  std::string m_name;
  AttribsMap m_attributes;
  ChildrenMap m_children;

public:
  XMLElement(const std::string& name);

public:
  void add_child(const XMLElement_Ptr& child);
  const std::string& attribute(
     const std::string& name) const;
  std::vector<XMLElement_CPtr> find_children(
     const std::string& name) const;
  XMLElement_CPtr find_unique_child(
     const std::string& name) const;
  bool has_attribute(
     const std::string& name) const;
  bool has_child(const std::string& name) const;
  const std::string& name() const;
  void set_attribute(const std::string& name,
     const std::string& value);
};

XMLElement::XMLElement(const std::string& name)
:  m_name(name)
{}

void XMLElement::add_child(
   const XMLElement_Ptr& child)
{
  m_children[child->name()].push_back(child);
}

const std::string& XMLElement::attribute(
   const std::string& name) const
{
  AttribCIter it = m_attributes.find(name);
  if(it != m_attributes.end()) return it->second;
  else throw Exception( "The element does not have
     an attribute named " + name);
}

std::vector<XMLElement_CPtr>
   XMLElement::find_children(
   const std::string& name) const
{
  ChildrenCIter it = m_children.find(name);
  if(it != m_children.end()) return it->second;
  else return std::vector<XMLElement_CPtr>();
}
```

Listing 7

When the next token is no longer an **IDENT** (i.e. when we've reached the end of the parameter list), we check to see whether we're dealing with a simple element or a complex element. If the former, we'll encounter an ending **RSLASH**. If the latter, we'll instead see an **RBRACKET**, at which point we recursively read in any sub-elements, followed by the closing element tag, carefully checking that the names match in the process. Either way, we return the element once it's complete.

Elementary, Watson

The only remaining piece of the puzzle is the design of the XMLElement class: how do we actually represent and use the AST generated by the parser? The structure I came up with is shown in Listing 7. There's nothing particularly complicated about this bit, so I won't dwell on it. What is worth illustrating instead is how easy it is to use (see Listing 8, which shows a snippet of code from a function to load an Ogre mesh file [Ogre]).

Other parsers near you

There's a lot more to parsing than I've been able to show here with a simple example. Parsers in general can be either top-down or bottom-up (the recursive descent parser shown in this article is an example of the former). Top-down parsers try and turn the start symbol of the grammar (e.g. in our example in Listing 3, this would be **Document**) into the token sequence observed, by replacing left-hand sides of productions with right-hand sides. For example, in our case, we start off knowing we have a **Document**, which means we must have a sequence of **Elements**, etc. The alternative

```
XMLElement_CPtr XMLElement::find_unique_child(
   const std::string& name) const
{
  ChildrenCIter it = m_children.find(name);
  if(it != m_children.end())
  {
    const std::vector<XMLElement_CPtr>&
       children = it->second;
    if(children.size() == 1) return children[0];
    else throw Exception( "The element has more
       than one child named " + name);
  }
  else throw Exception(
     "The element has no child named " + name);
}

bool XMLElement::has_attribute(
   const std::string& name) const
{
  return m_attributes.find(name) !=
     m_attributes.end();
}

bool XMLElement::has_child(
   const std::string& name) const
{
  return m_children.find(name) !=
     m_children.end();
}

const std::string& XMLElement::name() const
{
  return m_name;
}

void XMLElement::set_attribute(
   const std::string& name,
   const std::string& value)
{
  m_attributes[name] = value;
}
```

Listing 7 (cont'd)

```
XMLLexer_Ptr lexer(new XMLLexer(filename));
XMLParser parser(lexer);
XMLElement_CPtr root = parser.parse();
XMLElement_CPtr meshElt
   = root->find_unique_child("mesh");

XMLElement_CPtr submeshesElt
   = meshElt->find_unique_child("submeshes");

std::vector<XMLElement_CPtr> submeshElts
   = submeshesElt->find_children("submesh");
size_t submeshCount = submeshElts.size();

std::vector<Submesh_Ptr> submeshes;
for(size_t i=0; i<submeshCount; ++i)
{
  const XMLElement_CPtr& submeshElt
     = submeshElts[i];
  std::string materialName
     = submeshElt->attribute("material");

//...
```

Listing 8

is bottom-up parsing. This starts from the observed token sequence and tries to turn it into the start symbol by applying production rules in reverse. For instance, if we see **IDENT EQUALS VALUE**, we know we're dealing with a **Param**, and so on.

Different parsing algorithms have different strengths and weaknesses: generally speaking, there's a trade-off between how general a grammar a parser can recognise, and the runtime and implementation costs (i.e. the more general parsers tend to be harder to implement and take longer to run). To give you a flavour of some of the algorithms out there, here's a quick description of a few alternatives:

- Table-based LL parser (top-down).

  Calling the left-hand sides of productions 'non-terminals' and the actual tokens like **IDENT**, etc., 'terminals', this sort of parser takes a numbered series of rules and a parse table of type Terminal x Non-Terminal -> Rule Number. It operates using a stack, which initially contains only the starting non-terminal. It iteratively examines the top element of the stack and the next character in the input stream. If the top element of the stack is a terminal, then either it matches the next input character (in which case they are consumed), or it doesn't (there's a syntax error). If the top element is a non-terminal, the correct rule to apply is looked up in the parse table (indexing on the top stack element and the next input character), and the top stack element is replaced with the right-hand side of the appropriate rule (note that the input character is not consumed). If there's no rule for this case in the parse table, there must be a syntax error.Finally, if the stack is empty, then either there's no next input character (we're done parsing), or there is (yet another type of syntax error).

- Shift-reduce parser (bottom-up).

  This works using a stack and the input sequence. At each iteration, it decides whether to shift (push the next input token onto the stack) or reduce (apply a grammar rule to the things on top of the stack). The goal is to empty the input sequence and reduce the stack to the start symbol.

- GLR (Generalized Left-to-right Rightmost derivation) parser.

  A hardcore parser designed for nondeterministic and ambiguous grammars.

If you're interested in parsing, I advise you to have a read through [DragonBook]; it contains a lot of very interesting stuff. It has to be emphasised, however, that these days parsing is considered essentially a solved problem. If you need a parser for real-world use, it's rarely the case that you should roll your own: you're usually much better off using Yacc or something like it [LexYacc]. (You might also want to investigate the Spirit parser framework: indeed I'm reliably informed that there have been a couple of Overload articles [Guest04, Penhey05] about it before!) All the same, it's always a good idea (and indeed fun) to understand what these things are doing under the hood as well.

Conclusion

In this article, I've shown how to construct a basic XML parser (email me if you want the full source code) and given a brief overview of some other types of parser you might like to investigate. It's worth noting that often one of the problems you're likely to have with parsing is not actually in reading text that is syntactically valid: it's dealing with invalid text and outputting suitable error messages. This is a particular problem for compilers. Whilst it's outside the scope of this article, one of the problems is that good error messages have to be readable by humans in order to help them correct the error: this often requires an understanding of intent. Another issue is that stopping at the first parse error you encounter often isn't good enough: real-world parsers need to carry on and attempt to parse the rest of the input, and we'd ideally like them not to output a load of spurious error messages while they're doing it. All in all, it's a fascinating topic, and something I highly recommend investigating further. ■

Acknowledgements

Many thanks to the Overload review team for the many improvements suggested to the original article.

References

[DragonBook] *Compilers: Principles, Techniques and Tools* (1st Ed.), A Aho, R Sethi, J D Ullman, Addison Wesley, January 1985.

[Guest04] 'Mini-project to Decode a Mini-language', Thomas Guest, *Overload* #64, December 2004.

[LexYacc] The Lex and Yacc Page, http://dinosaur.compilertools.net.

[Ogre] Open Source 3D Graphics Engine, http://www.ogre3d.org.

[Penhey05] 'With Spirit', Tim Penhey, *Overload* #69, October 2005.

Quality Matters: Introductions, and Nomenclature

There are many aspects of Software Quality. Matthew Wilson introduces us to some of the concepts.

It's been a few years since I stopped my column-writing (for C/C++ Users Journal and Dr. Dobb's), and I've rather gotten out of the habit of disciplined writing, much to the chagrin of my long-suffering Addison-Wesley editor. (Sorry, Peter, but this stuff *will* help with the coming books, I promise.) Anyway, I plan to get back into the swing and hope to provide material that will cause some of you to ponder further as you read your issue of *Overload* every second month, and I'd like to thank Ric Parkin for giving me the opportunity to foist my opinions on a captive audience once again.

So, what's the deal with 'Quality Matters'? Well, as the cunning linguists among you may have realised, the title is a double-meaning pun, which appeals to me greatly (and no less so to Ric). But the overloading is apposite. In one sense it means that *quality is important*. Which it is. The other is that what is to be discussed will be *issues of quality*.

But what is quality? For sure, when you drive a nice car, stay in a nice hotel, eat a nice meal, watch a nice film (or movie, if you will), read a nice book, or have a nice conversation, you can feel the quality. But defining it is exceeding hard; it takes more than just a subjective 'nice'.

In all my programming related activities – author, consultant, programmer, trainer – quality is crucial, and having spent enough years doing these things I am able to sniff out quality (and its absence). I am comfortable to go into any client's development team and poke around the codebase with confidence, invariably producing useful analyses of what I find. But when asked to pontificate about 'software quality' absent context of a codebase or design documents, I find myself coming up a little dry. At such times I wonder at the abilities of those who have, or at least appear to have, a software dictionary in their heads.

My eldest son and I have recently taken up fencing – the one where you attempt to disembowel your opponent with a sword, rather than the one where you hit lumps of wood into the ground with a hammer (although that would also be fun, I think) – and it has brought into clear focus just how much we rely on training for a great many things. I've been cycling for more than 20 years, and the act of moving about a bike in response to where I want to go seems entirely subconscious. At least I hope my subconscious is handling it, because I know I'm not! Similarly, I've been programming professionally for 15+ years (and unprofessionally for 25+ if we count Vic-20s, ZX-81s, BASIC and 6502 assembler), and by now a huge proportion of what I do is also subconscious. It's only when training other programmers, or (attempting to) write books and articles that I get to glimpse some of the other 90% of the iceberg of software lore that has been accreted into my programming super-ego.

So, in part, this column will be a journey about codifying what my conscious mind has forgotten, in the hope of stringing together a cogent philosophy. Time will tell …

Matthew Wilson is a software development consultant and trainer for Synesis Software who helps clients to build high-performance software that does not break, and an author of articles and books that attempt to do the same. He can be contacted at matthew@synesis.com.au.

Being a practical sort of chap – I'd rather write a software component than write a software component specification – the prognostications in this column will all be based around practical issues, usually, I predict, around some block of code that's offended or inspired me. I intend to examine successful (and some unsuccessful) software libraries and applications, and rip them apart to look at what has been done well, and what could be changed to improve them. I also plan to demonstrate how intrinsic and diagnostic improvements can be applied without damaging or detracting from the existing functionality, robustness or efficiency.

Despite these promises of greasy rags and dirty finger nails, we will need a theoretical framework on which to base the analyses. To that end I'm going to start the journey by identifying three groups of software quality related subjects, which reflect the method I bring to bear in my consulting work.

Most/all of the subjects mentioned in this introductory instalment will be given further treatment in later instalments.

A nomenclature for software quality

There are a multitude of possible ways of slicing and dicing the software quality landscape, and a multitude of software quality metrics offered by different thinkers on the subject. There are terms such as *adaptability*, *cohesiveness*, *consistency*, *correctness*, *coupling*, *efficiency*, *flexibility*, *maintainability*, *modularity*, *portability*, *reliability*, *reusability*, *robustness*, *security*, *testability*, *transparency*, *understandability*, and on and on it goes. What do they all mean? Are they all useful?

I couldn't hope to distil down all these different ideas down into a single set, and I won't pretend to try. What I'm going to talk about in this column are aspects of software quality that I understand and utilise in my consultancy, training and my own software development activities. They break down, more or less neatly, into three groups:

- Intrinsic characteristics
- (Removable) diagnostic measures
- Applied assurance measures

In this instalment I'll flesh out the definitions of the first group, since they'll occupy much of my interest in the next few instalments. I'll also offer brief discussions of the second and third groups now, and go into more detail about the individual items in later instalments when they're relevant (and when there's the space to give them adequate treatment).

Intrinsic software quality characteristics

To be of any use, software quality characteristics have to be definable, even when the definitions involve relativism and subjectivity. To this end, I spent a deal of effort when writing my second book – **`<plug>`***Extended STL, volume 1: Collections and Iterators* [XSTLv1]**`</plug>`** – considering the notion of quality for software libraries. I came up with seven characteristics of quality:

- Correctness/reliability/robustness
- Efficiency

software quality characteristics have to be definable, even when the definitions involve relativism and subjectivity

- Discoverability and transparency
- Modularity
- Expressiveness
- Flexibility
- Portability

Each of these characteristics is innate to a given software entity. Regardless of whether its authors or users know or care about such characteristics, and regardless of whether anyone takes the trouble to measure/assess it in respect of them, every component/library/ sub-system has a level of robustness, efficiency, discoverability and transparency, etc. that can be reasoned about.

For everyone who has not managed to get further than the prologue of *Extended STL, volume 1*, I'll offer definitions of these again now. For those who have, you will probably benefit from reading them, as I've refined some ideas in the last couple of years.

Correctness, reliability and robustness: first pass

Forgetting for the moment all the other issues about how fast it runs, whether it can be easily used/re-used/changed, the contexts it can be used in, and so forth, the *sine qua non* for any piece of software is that it must function according to the expectations of its stakeholders.

Battle-hardened software developers will (hopefully) have bristled at the vagueness of that last phrase 'function according to the expectations of its stakeholders'. But I am being deliberately vague because I believe that this area of software quality is poorly defined, and I hope to come to a better definition than any I've found so far.

Three terms are commonly used when it comes to discussing the expected (or unexpected) behaviour of software: *correctness*, *reliability* and *robustness*. The first of these has an unequivocal definition:

Correctness is the degree to which a software entity's behaviour matches its specification.

Cunningly, the definition is able to avoid equivocation by passing off to the definition of 'specification'. And that's no small thing, to be sure. I am going to skip discussion of what form(s) specifications might take until the next article, for reasons that will become clear then.

I'm also going to skip out on discussing the issues of *robustness* and *reliability*, because there is a lot of equivocation on their definitions in the literature – I'm thinking mainly of McConnell [CC] and Meyer [OOSC] here, but they're not alone – and the only sense I can make of them is when dealing with the specification question.

I will, however, leave you to with something to ponder, which will inform the deliberations of the next instalment: I call it the *Bet-Your-Life? Test* (see sidebar).

The Bet-Your-Life? Test

Assume a perfect operating environment of unfailing hardware and perfect implementations of all layers of software abstraction below the ones at which the following software entities are written. Would you bet your life on them being able to be written to '*function according to the expectations of its stakeholders*'?

(That these are all C is a reflection of the first C in ACCU and also of my need to keep the listings as small as possible. The choice of language is largely, though not completely, irrelevant, since we have already stipulated that the underlying layers of software abstraction are perfectly implemented.)

```
// 1. A boolean inversion: return == !b
bool invert(bool b);

// 2. A string comparison
int strcmp(char const* lhs, char const* rhs);

// 3. A base-64 conversion [B64_ENCODE]
size_t b64_encode(
  void const* src,  size_t srcSize
, char*       dest, size_t destLen
);

// 4. A recursive file-system search
// [RECLS_SEARCH]
RECLS_API Recls_Search(
    char const* searchRoot
,   char const* pattern
,   int         flags
,   hrecls_t*   phSrch
);
```

Well, I'd bet my life, and those of my wife and sons, on my being able to implement (1) perfectly. I'd expect all of you to be comfortable to make a similar compact with your most precious lives.

Conversely, I can tell you that I definitely would not ever be prepared to bet anything of grave importance on the implementation of (4). This is despite my having used my implementation of it, seemingly entirely successfully, probably tens of thousands of time over the last several years.

Beyond those two definitive positions, I'm somewhat up in the air on the other two. Since I'm a programmer, I'm instinctively driven to believe that I could implement perfect implementations of both (2) and (3). And I have, in fact, implemented both before, numerous times in the case of `strcmp()`. And as far as I am aware, both are perfect. But I still wouldn't bet my life on it.

Obviously, the interesting part of this thought experiment is why I hold those different positions, and the criteria I have considered in forming them. The key is in understanding the difference between correctness and robustness and/or reliability, and between contract specification and testing, all of which will be discussed in the next instalment. In the meanwhile, I'd be keen to hear from readers on their positions.

```
DIR* dir = opendir(".");
if(NULL != dir)
{
  struct dirent* de;
  for(; NULL != (de = readdir(dir)); )
  {
    struct stat st;
    if( 0 == stat(de->d_name, &st) &&
        S_IFREG == (st.st_mode & S_IFMT))
   {
     remove(de->d_name);
   }
  }
  closedir(dir);
}
```

Listing 1

Discoverability and transparency

Discoverability and *transparency* are a pair of software quality characteristics that pertain to the somewhat nebulous concept of how '*well-written*' a software entity might be. They are defined as follows [XSTLv1]:

Discoverability is how easy it is to understand a component in order to be able to use it.

Transparency is how easy it is to understand a component in order to be able to change it.

I believe that it's self-evident that these two are hugely significant in the success of software libraries. Particularly so with discoverability, since people have very low tolerance for discomfort in the early stages of adoption of a software library. (This is one of the reasons why I write so many C++ libraries: I cannot *discover* the interface of many existing ones.)

The two characteristics have significant impact on the (uselessly vague and overly general, in my opinion) notion of *maintainability*. If something is hard to change, then the changing of it is going to be (i) unwillingly undertaken and (ii) of high risk. Furthermore, if something is hard to use, then its users will be poorly qualified to guide its evolution.

Consequently, I find mildly astonishing the absence of much concern over these two quality characteristics in commercial developments. In my opinion, discoverability and transparency have one of the biggest cost impacts on software projects, and we should aim to maximise them as much as is possible. The problem with that intent, however, is that, unlike correctness, discoverability and transparency are subjective and non-quantifiable. But there is hope, in the application of the idiom: 'when in Rome …'

Efficiency

Efficiency is the degree to which a software entity executes using the minimum amount of resources. The resources we commonly think of are processing time and memory, but other resources, such as database connections and file handles, can be equally important, depending on application domain.

Efficiency may be primarily concerned with:

- Whether the algorithm chosen to implement the entity's functionality is implemented to use the minimum amount of resources, and
- Whether another algorithm may fulfil the entity's functionality using fewer resources

There are other factors that can influence efficiency, including:

- The compilation environment used to translate and optimise the code
- The execution environment used to execute the code (e.g. choosing one JVM over another, single vs. multi-core hardware)

```
readdir_sequence entries(".",
   readdir_sequence::files);
std::for_each(entries.begin(), entries.end()
   , ::remove);
```

Listing 2

Doubtless we've all heard of Hoare/Knuth's 'premature optimisation is the root of all evil' quotation. The problem is, this has been misunderstood and seized upon by a generation of feckless and witless programmers who don't care about their craft and should instead be spending their days giving someone else's profession a bad name, egged on by commercially-driven mega-companies with giant frameworks and rich consulting services to be foisted on under-informed clients.

I've never worked on a commercial software project where performance was not important, and, frankly, I can't think of a serious software application where it would not be. The actual full quote was 'we should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil', which makes a whole lot more sense. It's not suggesting that programmers blithely ignore performance, rather it's an exhortation to focus on the most significant performance issues first, rather than to 'sweat the small stuff'. Now that makes a lot of sense.

As Herb Sutter has postulated [FREE-LUNCH], we have run out of the performance free-lunch, and it's time to start tightening our belts.

Expressiveness

Expressiveness is 'how much of a given task can be achieved clearly in as few statements as possible' [XSTLv1]. Expressiveness is also known as programming power (or just power), but it's a poorer term for a variety of reasons, and I won't mention it further.

Some code examples will illustrate the point more clearly than words. The first one involves file-system search on UNIX (and is parsimoniously lifted from the section on expressiveness in the prologue of *Extended STL, volume 1* [XSTLv1]):

Contrast Listing 1 with Listing 2.

Here a C++ class – actually an *STL Collection* [XSTLv1] – combined with a standard algorithm is used to provide a significantly more expressive means of tackling the problem of removing all files from the current directory. It achieves this by raising the level of abstraction – all possible files are treated as a single entity, a collection – and by relying on language facilities – deterministic destruction to handle resources, and namespacing rules to define constants with natural names.

I hope it's clear that the expressiveness of class + algorithm engenders a substantial increase in the transparency of the application code. This is not measured solely in the reduction of lines of code, but also in the removal of pointers, the *S_IF???* constants, and explicit resource management, and in the ability to read the second statement as 'for each item in entries, remove [it]'. However, before we get carried away, we must balance such gains in concision with the discoverability (or lack thereof) of the components. The first 'big thing to be known' is the use of STL iterator-pair ranges and algorithms. Certainly, to experienced C++ programmers this is now as straightforward as tying one's shoes. But STL is *not*, in my opinion, intuitive. Of less magnitude (since it's not a global idiom like STL collection + algorithm), but still significant, is the dialecticism of all abstractions, in this case the `readdir_sequence` class.

There's an obviousness to this that's teeth-grindingly painful to state, but it needs to be stated nonetheless. Every meaningful software component provides either a different interface or a different implementation, or both, to all others. If it doesn't, you find yourself the proud owner of a wheel the same size and specification as your neighbour, and there's precious little use in that.

If, as is the more common case, your component has a different interface to existing ones, then by definition you affect its discoverability. Users *must* familiarise themselves with the component's interface to be able to use it. The challenge in this case is to restrict the non-normative aspects to the minimum, without sacrificing other aspects of software quality or

```
IEnumerator en = FileSearcher.Search(directory,
  patterns).GetEnumerator();
while(en.MoveNext())
{
  IEntry entry = (IEntry)en.Current;
  if(!entry.IsReadOnly)
  {
    Console.Out.WriteLine(
        entry.SearchRelativePath);
  }
}
```

Listing 3

functionality. In the case of the `readdir_sequence` class, this is limited to the construction of instances, which involving specifying a search directory and/or search flags, and the collection's value type (which happens to be `char const*`). The rest of the functionality of the class adheres to the requirements of an *STL Collection* [XSTLv1] – it provides access to elements via `begin()` and `end()` iterator ranges – and therefore can be used in the same, idiomatic manner as any other STL collections.

Conversely, if your component provides a new implementation, you will have to reveal something about it to potential users, or they'll have no reason to use it. And like as not you'll have to reveal something more substantive than just saying 'it's faster', so you'll find yourself with a *leaky abstraction* [LEAK, XSTLv1]. Whatever information that must be leaked adds to the sum of knowledge that must be mastered for your component to be used properly – it affects its discoverability. An example of this might be a fast memory allocator that works by using a custom heap that ignores all `free()` calls and simply dumps the memory pool at an established known point. Users will have to abide by the rules of when/where to allocate in order to establish that point.

As if all that wasn't enough, there are even cases where expressiveness can detract from transparency. Consider the following three chunks of C# code, using the new '100%' rewrite of the recls [RECLS-100%] recursive file-system search library (Listing 3).

Thankfully no-one has to write code like that. Even in C# 1.0 you could let the compiler do some of the hard work for you, via `foreach`, as in Listing 4.

Such loops are idiomatic in the programming world, not just to C#, and I can't imagine anyone arguing that the increased expressiveness of the second form incurs a cost to discoverability or transparency over the first.

With C# 3.0, it's possible to condense things even further by using the extension methods provided in the latest recls .NET library in combination with the new language facility of lambda constructs, giving a single statement:

```
FileSearcher.Search(directory, patterns)
  .Filter((entry) => !entry.IsReadOnly)
  .ForEach((entry) => Console.Out.WriteLine(
     entry.SearchRelativePath));
```

I don't think this improvement is quite so unequivocal. Certainly the concision appeals to C# power uses – it does to me – but I don't think anyone can argue, even when the use of lambda becomes second nature to all C# programmers, that such a statement is as transparent as the second loop.

```
foreach(IEntry entry in FileSearcher.Search(
   directory, patterns))
{
  if(!entry.IsReadOnly)
  {
    Console.Out.WriteLine(
        entry.SearchRelativePath);
  }
}
```

Listing 4

I believe that expressiveness is a large factor in the preferences that programmers have for one language over another. It directly impacts productivity because programmers have to type less to express their intent and, importantly, read less when they come back to modify it. It also indirectly affects productivity by reducing defect rates, since many lower level defects simply don't occur. It's not just that housekeeping tasks are obviated: looking back to the file enumeration example, we see that there is no opportunity to forget to release the search handle (via `closedir()`) because `readdir_sequence` does it for us. It's also that the amount of distraction from the main semantic intent of code is reduced: in the C version, the call to `remove()` is that much less discriminated from the boilerplate than in the C++ version, wherein it takes centre stage.

It's no coincidence, therefore, that many of the major languages appear to be making substantial moves to improve their ability to support expressiveness.

Flexibility

Flexibility is 'how easily a [software entity] lets you do what you need to do, with the types with which you need to do it' [FF1].

As you may remember from my recent series of articles on `FastFormat` [FF1, FF2, FF3], flexibility is something I prize very highly in software libraries. To be able to translate your design clearly and correctly into code it is important to be able to express your program logic in terms of the types you deem appropriate to *your* level of abstraction, rather than the types appropriate to the level of abstraction of the component or sub-system in terms of which you're implementing. When you can't do this, you experience what I call *abstraction dissonance*. The following definitions are borrowed from my still-in-preparation book *Breaking Up The Monolith: Advanced C++ Design Without Compromise*, which I'm hoping to finish this year; the web-site [BUTM] contains a slowly growing list of concept/pattern/principle definitions, including:

Unit of Currency: the primary physical type with which client code represents a given conceptual type; the primary physical type by which a component or API communicates a given conceptual type to its client code.

Abstraction Dissonance: the condition whereby client code is written using units of currency that exist at a higher level of abstraction than those used by the libraries/APIs in terms of which the client code is written.

My signal case for abstraction dissonance can be composed from two of the most commonly used and well-understood components from the C++ standard library:

```
std::string path = "data-file";
std::ifstream stm(path); // DOES NOT COMPILE!
```

That this does not compile, and the user is forced to pollute the client code with the damnable `.c_str()`, is nothing less than ridiculous.

```
std::ifstream stm(path.c_str());
```

There are several things that can be done to avoid, or to obviate, situations like this, and I plan to cover them in future instalments. (They're also discussed at length in *Monolith*, should it ever get to the presses.)

Flexibility directly impacts expressiveness and transparency, and indirectly impacts discoverability, efficiency, modularity, and correctness/robustness. I intend to cover many instances of conflict/compromise between these characteristics in the coming articles.

Modularity

Modularity is about dependencies, usually unwanted ones. This tends to have two forms [FF1]:

- What else do I need to do/have in order to work with the library
- What else do I need to do/have in order to use the library to work with other things

There's been a long and inglorious history of poor modularity in the programming pantheon. Windows programmers will remember (or may still be using) the vastness of the MFC libraries. Java and .NET programmers still experience the deployment hassles of their respective virtual support machinery (though many seem not to realise the problems they, or their users, face). But those are all easy pickings. Modularity problems are also to be found in far more subtle, though no less problematic, situations.

Portability

Portability is about how readily you can use a software entity in your chosen *operating environment*. The 'operating environment' may differ in any/all of the following:

- Operating system
- Processor architecture
- Compiler
- Libraries
- Feature modes (e.g. without exceptions and/or RTTI)

C was intended as a portable assembler, and as such it does a great, albeit partial, job of abstracting away disparate physical architectures. But even then, porting compiled C programs to different architectures is impossible, and porting C programs by source often involves troubling aspects, including, but not limited to, architecture differences (e.g. in the sizes of types and byte-ordering) and operating system services. (Anyone who's ported between UNIX and Windows will know the pain to which I allude.)

The same goes for C++, with the considerable additional difficulties resulting from the vastly different interpretations and offerings of the C++ language facilities by the compilers currently available. As I've mentioned in previous writings, I reckon that 90%+ of the time I spend on my open-source C++ libraries is in making the code play nice with all the compilers. It's not cool, and it's not fun.

But things are hardly perfect in the virtual machine languages. I have had my fair share of Java's 'write once, debug everywhere' misery, not to mention the sad irony of C#, a formerly-mediocre/now-good/looking-like-becoming-great language bound to a basket-case family of operating systems. I look forward to some smart people separating the language from Windows *and* the .NET runtime: that would produce something very interesting.

Even when one exits the world of compiled languages entirely, portability is still imperfect, in part due to the leaking up of abstractions of the underlying operating environments. Two obvious ones are the slash/backslash shemozzle, and the lack of globbing in some command-line interpreters. But it can be more profound, such as the relative costs of starting new processes and new threads.

I could go on and on, but I won't. Suffice to say that there is no perfect portability, but there are definitely things that can be done to improve it.

Quantifying quality: relativity and subjectivity

Almost every one of the above characteristics is relative and/or subjective. That does not stop them being useful, but it does mean that we should try to qualify observations about a particular characteristic in terms of the things that we can assess absolutely and objectively.

For example, transparency is highly subjective, but we may attempt to quantify it by enumerating the points of lore and law that must be known to understand a given chunk of code. Similarly, we can make some rudimentary measurement of expressiveness by counting lines of code, and number of sub-expressions in each line. I have my own ideas on these things, and there's plenty of wisdom in the canon, but I'm keen to hear from readers any of their own opinions on the matter.

Characteristics in concert ...

Programmers will always be biased towards one or more intrinsic software quality characteristics, although the particular characteristic(s) may differ in different contexts. When I'm doing C++ it's all about being 'fast', i.e. safe and quick. In Ruby it's all about expressiveness, and discoverabilty and transparency. But it's important to assess maturing components in terms of all relevant intrinsic software quality characteristics.

In many circumstances, just the act of examining a software component in terms of one or more intrinsic software quality characteristics can lead to easy changes that will enhance it in terms of others. It may also, obviously, highlight important deficiencies.

For example, I often find that a first version of a component will be written in terms of some other, useful, component from another library. But if it turns out that just this one piece of reuse incurs coupling to a large library that can cause substantial inconvenience (and hinder acceptance) in terms of modularity and portability, I will be inclined to eschew the third-party component and implement its functionality explicitly within the developed code.

But beyond incidental and independent improvements in respect of particular software quality characteristics, it is often the case that these software quality characteristics can be in conflict. I believe that these conflicts must be explicitly identified and considered, and documented for the benefits of authors (and future maintainers) *and* for its users.

For example, the FastFormat library, described in articles in the last three instalments of this journal, has a bunch of fairly clear design decisions:

1. 100% type-safety, and the highest possible correctness/reliability/robustness
2. Extremely high efficiency – no duplication of measurements or wasted allocations
3. High flexibility (including infinite extensibility)
4. Support for I18N/L10N
5. Highest possible level of expressiveness that does not detract from 1–4.
6. Highest possible levels of discoverability & transparency that do not detract from 1–5.
7. Modular
8. Portable

Users are thus able to make a judgement as to whether they can avail themselves of FastFormat's performance, robustness and flexibility advantages or, if they require the highest possible levels of expressiveness (for width-formatting of numeric types), choose an alternative.

(Removable) diagnostic measures

The foregoing characteristics are intrinsic. They are *of* the software, if you will. As we will shortly discuss, the next group are *of* the programmer(s). They are things done to (measure the) software by human beings, either entirely manually, or with the assistance of computers, or entirely by automated process but still operating as an agency of the programmer(s). In all cases, they are external to the software.

In between these two positions lies a group of measures that are in the software but are of the programmer. They are used for assessing or ensuring the quality of the software. But they have one important characteristic in common: in all cases they are removable. With the terminological assistance of beneficent and sagacious members of the ACCU general mailing list, I now call these *(removable) diagnostic measures*. They include:

- Code coverage constructs
- Contract enforcements
- Diagnostic logging constructs
- Static assertions

The parenthetical inclusion of 'removable' in the name serves as an important reminder of the *principle of removability* (again, from the *Monolith* website [BUTM] in lieu of the book; I have Christopher Diggins to thank for the nice wording):

The Principle of Removability:

When applied to contract enforcement, the principle of removability states: A contract enforcement should be removable from correct software without changing the (well-functioning) behaviour.

When applied to diagnostic logging, the principle of removability states: It must be possible to disable any log statement within correct software without changing the (well-functioning) behaviour.

The same thing goes for code coverage constructs, and for static assertions.

Obviously, there's a bit of circularity here insofar as we've already established correctness as only being definable in terms of contract enforcements or automated testing, and now we've saying that contract enforcements can be removed from correct software. Well, what can I tell you? Somewhere we've got to take a stand.

Applied assurance measures

This group of things are actions that are done by, or on behalf of, software developers, many of which are to be found as primary constituents of established development methodologies. More than half of the list is about testing.

- Automated functional testing
- Performance profiling and testing
- User acceptance testing
- Scratch testing
- Smoke testing
- Code coverage analysis/testing
- Review (manual and automated)
- Coding standards
- Code metrics (automated and manual)
- Multi-target compilation
- … and more …

Most of these should be well known to all competent and experiences programmers, and I don't need to say any more about them at this time.

The one thing I will comment on now is the use of the term *measure*. Just like the title of the column, this meaning of the term is helpfully overloaded: a measure can be a metric/assessment, and also an approach/policy.

Puzzling phenomena

Thanks to the Global Financial Crisis^{TM}, I've recently had to devote serious effort to the business of attracting clients for the first time in a comfortably long while. In updating the company website and my own vitae, I've noted some surprising observations, including, in no particular order, the following:

1. **b64** is popular, and **recls** is not
2. **Pantheios** is popular, and **FastFormat** is not so much
3. No software (sub-)system developed by Synesis Software (my company) has ever had a failure in production. (Caveat: there's been one apoptotic episode, but that was a good thing. Something to examine when we talk about contract programming.)

The third fact is the one with the most commercial bite, but I assure you that my mentioning it is more than mere grandstanding. (Well, there's some grandstanding in there of course, and if any potential clients out there want some magic no-fail pixie dust sprinkled on their codebase, by all means get in contact.) But the main point is that even though I have always prized software quality – even from before I was experienced enough to properly detect or apply it – it was still something of a pleasant shock to realise that we've never had a production failure. Given that the various software (sub-)systems have handled billions of dollars of transactions, that's a pretty comforting thought. And it's nice to be able to trumpet that on the company website. But why should that be of interest to me or you, gentle readers?

Well, it's of direct interest to me because it's quite an improbable achievement, and realising it gives me confidence to attempt the undertaking of writing this column. And I hope it's of interest to you in that it might give you some confidence that some of what I say might be worth a read (assuming you can stomach my grandiloquent loquacity).

Anyway, I won't attempt to offer further convincing on my qualification for the post. If I bodge it, Ric will give me the flick, and rightly so.

Open-source library popularity

What of the relevance of the other two facts, pertaining to the relative 'popularity' of two pairs of my libraries. On the surface, the two popular libraries should be in the shadow of the two less-popular ones, and the consideration of why they're not has raised a number of issues in my mind pertaining to software quality. Let's look at some of the aspects of the puzzle.

Generality of purpose

The b64 library provides Base-64 encoding/decoding. The recls library provides platform-independent recursive file-system search facilities. The latter is surely more generally useful than the former.

Pantheios is a diagnostic logging API library. FastFormat is a formatting library. Although I will argue strongly later that it should not be so, I believe that formatting is to be found far more frequently than logging in C++ codebases.

Available languages

b64 provides a C and a C++ API. recls provides C, Ch, COM, C++ (and STL), C#/.NET, D, Java, Python and Ruby APIs.

Pantheios and FastFormat are both C++ libraries, although Pantheios does provide a C-API for logging C programs.

Promotion

I have not written any articles about b64, and beyond passing a link once or twice I have done nothing to promote it. Conversely, I wrote an extensive series of articles about recls for CUJ/DDJ in 2003-5. Unlike the other three, b64 doesn't even have its own domain, and just has a downloads page that hangs off an unremarkable, barely linked part of the Synesis website. Furthermore, apart from one commercial project, the only thing I've ever used b64 for is to implement the `pantheios::b64` inserter class. And b64 is bundled with Pantheios, only adding to the puzzle of its (relatively) high independent downloads.

I have not (yet) written any articles about Pantheios, whereas I've written a recent series of three articles about FastFormat [FF1, FF2, FF3], where I pretty much prove its superiority over the existing alternatives.

Frequency of release

Although not differing by orders of magnitude, the frequency of releases of recls is greater than that of b64, and FastFormat has been greater than that of Pantheios over the last few months. This serves to further highlight the disparity in ongoing level of downloads (and other activity) of the latter libraries.

Popularity

Over the past couple of years, b64 downloads have been steady at around 2200 per year, whereas the average for recls is around 500 per year. Even though it's not a huge number per se, I find it remarkable, given that Base-64 conversion is a niche area of functionality.

Similarly, Pantheios downloads tends to be several hundred per week, whereas FastFormat is around 50-80. And the SourceForge rankings, based on downloads, web page hits, forum and tracker activity, are similarly different: Pantheios tends to be in the top two hundred, FastFormat around 2000.

What gives?

Despite all these factors pushing in the favour of recls and FastFormat, there are clearly some important effects that are overriding them. I will expound on these in later instalments, but it's worth mentioning some now, I think:

- **Satisfiction**. There are several, very well-established formatting libraries available for C++ programmers, so FastFormat has a *lot* of mindshare to capture. Users of the existing libraries are *satisfied* with what *suffices*, in an effect I've previously called *satisfiction* [XSTLv1].
- **Green pasture**. In contrast, Pantheios has no serious competitors as a logging API library: the existing (and impressively feature-rich) logging libraries have APIs that are manifestly unfit for purpose.
- **Language**. I believe that, all other things being equal, a library implemented in C (such as b64) will be far more popular than one implemented wholly or partly in C++ (such as recls), due to concerns (well-founded or not) of performance, portability, and transparency.
- **Modularity**. b64 does not have any dependencies, not even on the C runtime library. FastFormat, Pantheios and recls all depend on the STLSoft libraries, which cause users more effort (even though, as 100% header-only, the effort involves nothing more than downloading and setting an environment variable).

Doubtless there's more to the situation than I have divined here, but it's enough to inform the analyses of these libraries that will start in the next instalment. I'm keen to hear opinions from readers their thoughts on this issue.

Column format

I don't know about you, but I find it very difficult to understand, or remember, concepts that are presented without examples. Similarly, I find it hard to write about concepts without using examples. So, the instalments of this column – except this first one – are going to be rich with example libraries, programs and code.

For most of these I'll be using my own code, largely because I am able to criticise it as much as is necessary without offending anyone else. The precise material will depend on what is uncovered as the articles progress, but I am confident we'll start in some of my open-source C and C++ libraries, and then move to particular algorithms, components and programs, including those in other languages. For example, I'm currently working with a colleague in updating the Synesis Software .NET libraries (and some .NET forms of several open-source libraries) for C# 2 and 3, and we're cooking. The contrast of what's superior and what's inferior to C++ is very thought-provoking.

In terms of subject areas, you can expect future columns to have diverse subjects, including some/all of the following:

- Correctness, robustness and reliability
- Contract programming: The principles of removability and irrecoverability
- Defining contracts: Identifying and defining software components
- Trade-offs in intrinsic software quality characteristics
- Attracting real users: I*t's the* coupling, *Stupid!*
- Efficiency for real
- Packaging
- The logging conundrum
- Component vs unit-testing: A scratch and sniff approach
- Automated testing
- The evils of the Boolean type(s)
- Overloading vs overriding
- Defining clean methods
- Software quality measures for multithreaded programming
- Cracking the abstraction puzzle
- Conformance: Structural, semantic, explicit, intersecting, and all manner of foul beasts
- … and lots of discussions of the differences in software quality approaches between different application areas, between different languages, between different layers of abstraction

Naturally, some of the material discussed will be a cheap rip-off from that already included in my books. The several in-progress book projects will likely overlap too. But the limited size and broad scope of the column will mean that there's plentiful opportunity to have unique content in each medium.

A quest for quality …

As ably discussed in *Code Complete* [CC], organisations are only able to significantly improve their software quality by a combination of measures. Importantly, the combination has to involve both automated measures and human measures.

I would like to point out that, in my opinion, more important than all the individual measures is a requirement for the people who are writing the software to have the wit and will to seek out quality processes and apply them, against the twin obstacles of business imperatives and the apathy/heroism of 'programmers' who should be in a different career. One of the wonderful things about being a programmer is that it is fun, it is creative, and it can (and should) be beautiful. In this crucial respect, it is a true craft, and it is my aim with this column to help others improve their craftsmanship through the (limited) discussion of quality concepts and the (generous) application of practical quality measures. I invite you to join me. ■

References and asides

[B64_ENCODE] This is one of the API functions from the b64 library, described at http://synesis.com.au/software/b64/doc/b64_8h.html#50a93e4f6a922c5314a9cb50befc2d13

[BUTM] http://breakingupthemonolith.com/

[CC] *Code Complete, 2nd Edition*, Steve McConnell, Microsoft Press, 2004

[FF1] *An Introduction to FastFormat, part 1: The State of the Art*, Matthew Wilson, Overload 89, February 2009

[FF2] *An Introduction to FastFormat, part 2: Custom Argument and Sink Types*, Matthew Wilson, Overload 90, April 2009

[FF3] *An Introduction to FastFormat, part 3: Solving Real Problems, Quickly*, Matthew Wilson, Overload 91, June 2009

[FREE-LUNCH] 'The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software', Herb Sutter, *Dr Dobb's Journal*, March 2005

[LEAK] http://en.wikipedia.org/wiki/Leaky_abstraction

[RECLS_SEARCH] This is one of the API functions from the recls library, described at http://www.recls.org/help/1.6.1/group__group__recls.html#a1

[RECLS-100%] I'm in the process of rewriting the recls library as 'recls 100%', whereby each implementation of recls for a given language will be implemented 100% in that language, rather than recls 1.0–1.8 where each language had a thinnish binding to the underlying C-API. See http://recls.org for progress.

[OOSC] *Object Oriented Software Construction*, 2nd Edition, Bertrand Meyer, Prentice-Hall, 1997

[XSTLv1] *Extended STL, volume 1: Collections and Iterators*, Matthew Wilson, Addison-Wesley, 2007

Code Rot

Maintaining code is vital to keep it working. Tom Guest explores what happens when you neglect it.

Those of us who have to tiptoe around non-standard or ancient compilers will know that template template parameters are off limits.

— Hubert Matthews [Matthews03]

Dvbcodec fail

Long ago, way back in 2004, I wrote an article for Overload [Guest04] describing how to use the Boost Spirit [Spirit] parser framework to generate C++ code which could convert structured binary data to text. I went on to republish this article on my website, where I also included a source distribution.

Much has changed since then. The C++ language hasn't, but compiler and platform support for it has improved considerably. Boost survives — indeed, many of its libraries will feed into the next version of C++. Overload thrives, adapting to an age when print programming magazines are all but extinct. My old website can no longer be found. I've changed hosting company and domain name, I've shuffled things around more than once. But you can still find the article online if you look hard enough, and recently someone did indeed find it. He, let's call him Rick, downloaded the source code archive, dvbcodec-1.0.zip [DVBcodec], extracted it, scanned the README, typed:

```
$ make
```

… and discovered the code didn't even build.

At this point many of us would assume (correctly) the code had not been maintained. We'd delete it and write off the few minutes it took to evaluate it. Rick decided instead to contact me and let me know my code was broken. He even offered a fix for one problem.

Code rot

Sad to say, I wasn't entirely surprised. I no longer use this code. Unused code stops working. It decays.

I'm not talking about a compiled executable, which the compiler has tied to a particular platform, and which therefore progressively degrades as the platform advances. (I've heard stories about device drivers for which the source code has long gone, and which require ever more elaborate emulation layers to keep them alive.) I'm talking about source code. And the decay isn't usually literal, though I suppose you might have a source listing on a mouldy printout, or on an unreadable floppy disk.

No, the code itself is usually a pristine copy of the original. Publishers often attach checksums to source distributions so readers can verify their download is correct. I hadn't taken this precaution with my dvbcodec-1.0.zip but I'm certain the version Rick downloaded was exactly the same as the one I created 5 years ago. Yet in that time it had stopped working. Why?

Standard C++

As already mentioned, this was C++ code. C++ is backed by an ISO standard, ratified in 1998, with corrigenda published in 2003. You might expect C++ code to improve with age, compiling and running more quickly, less likely to run out of resources.

Not so. My favourite counter-example comes from a nice paper 'CheckedInt: A policy-based range-checked integer' published by Hubert Matthews towards the end of 2003 [Matthews03], which discusses how to use C++ templates to implement a range-checked integer. The paper includes a code listing together with some notes to help readers forced to 'tiptoe around non-standard or ancient compilers' (think: MSVC6). Yet when I experimented with this code in 2005 I found myself tripped up by a strict and up-to-date compiler (see Figure 1).

```
$ g++ -Wall -c checked_int.cpp
checked_int.cpp: In constructor
`CheckedInt::CheckedInt(int)':
checked_int.cpp:45: error: there are no arguments
to `RangeCheck' that depend on a template
parameter, so a declaration of `RangeCheck' must
be available
checked_int.cpp:45: error: (if you use
`-fpermissive', G++ will accept your code, but
allowing the use of an undeclared name is
deprecated)
```

Figure 1

I emailed Hubert Matthews using the address included at the top of his paper. He swiftly and kindly put me straight on how to fix the problem.

What's interesting here is that this code is pure C++, just over a page of it. It has no dependencies on third party libraries. Hubert Matthews is a C++ expert and he acknowledges the help of two more experts, Andrei Alexandrescu and Kevlin Henney, in his paper. Yet the code fails to build using both ancient and modern compilers. In its published form it has a brief shelf-life.

Support rot

Code alone is of limited use. What really matters for its ongoing health is that someone cares about it — someone exercises, maintains and supports it. Hubert Matthews included an email address in his paper and I was able to contact him using that address.

How well would my code shape up on this front? Putting myself in Rick's position, I unzipped the source distribution I'd archived 5 years ago. I was pleased to find a README which, at the very top, shows the URL for updates, http://homepage.ntlworld.com/thomas.guest. I was less pleased to find this URL gave me a 404 Not Found error. Similarly, when I tried emailing the project maintainer mentioned in the README, I got a 550

Thomas Guest is an enthusiastic and experienced programmer, who has worked on everything from embedded systems to clustered servers. His website is http://wordaligned.org and he can be contacted at thomas.guest@gmail.com

REQUIREMENTS and PLATFORMS

To build the dvbcodec you will need Version 1.31.0 of Boost, or later.

You will also need a good C++ compiler. The dvbcodec has been built and tested on the Windows operating system using: GCC 3.3.1, MSVC 7.1

Figure 2

Invalid recipient error: the attempted delivery to thomas.guest@ntlworld.com had failed permanently.

Cool URIs don't change [W3C] but my old NTL home was anything but cool; it came for free with a dial-up connection I've happily since abandoned. Looking back, maybe I should have found the code a more stable location. If I'd created (e.g.) a Sourceforge project then my dvbcodec project might still be alive and supported, possibly even by a new maintainer.

How did this ever compile?

These wise hindsights wouldn't fix my code. If I wanted to continue I'd have to go it alone. Figure 2 is what the README had to say about platform requirements.

A 'good C++ compiler', eh? As we've already seen, GCC 3.3.1 may be good but my platform has GCC 4.0.1 installed, which is better. If my records can be believed, this **upperCase()** function (see Listing 1) compiled cleanly using GCC 3.3.1 and MSVC 7.1.

Huh? **Std::string** is a **typedef** for **std::basic_string<char>** and, as GCC 4.0.1 says, there's no such thing as a **std::basic_string<char><char>::iterator**:

```
stringutils.cpp:58: error: 'std::string' is not a
template
```

The simple fix is to write **std::string::iterator** instead of **std::string<char>::iterator**. A better fix, suggested by Rick, is to use **std::transform()**. I wonder why I missed this first time round? (See Listing 2.)

Boost advances

GCC has become stricter about what it accepts even though the formal specification of what it should do (the C++ standard) has stayed put. The Boost C++ libraries have more freedom to evolve, and the next round of build problems I encountered relate to Boost.Spirit's evolution. Whilst it would be possible to require dvbcodec users to build against Boost 1.31 (which can still be downloaded from the Boost website) it wouldn't be reasonable. So I updated my machine (using Macports) to make sure I had an up to date version of Boost, 1.38 at the time of writing.

```
$ sudo port upgrade boost
```

Boost's various dependencies triggered an upgrade of boost-jam, gperf, libiconv, ncursesw, ncurses, gettext, zlib, bzip2, and this single command took over an hour to complete.

I discovered that Boost.Spirit, the C++ parser framework on which **dvbcodec** is based, has gone through an overhaul. According to the change log the flavour of Spirit used by **dvbcodec** is now known as Spirit Classic. A clever use of namespaces and include path forwarding meant my 'classic' client code would at least compile, at the expense of some deprecation warnings (Figure 3).

```
Computing dependencies for decodeout.cpp...
Compiling decodeout.cpp...
In file included from codectypedefs.hpp:11,
                 from decodecontext.hpp:10,
                 from decodeout.cpp:8:
/opt/local/include/boost/spirit/tree/
ast.hpp:18:4: warning: #warning "This header is
deprecated. Please use: boost/spirit/include/
classic_ast.hpp"
In file included from codectypedefs.hpp:12,
                 from decodecontext.hpp:10,
                 from decodeout.cpp:8:
```

Figure 3

To suppress these warnings I included the preferred header. I also had to change namespace directives from **boost::spirit** to **boost::spirit::classic**. I fleetingly considered porting my code to Spirit V2, but decided against it: even after this first round of changes, I still had a build problem.

Changing behaviour

Actually, this was a second level build problem. The **dvbcodec** build has multiple phases (Figure 4):

1. it builds a program to generate code. This generator can parse binary format syntax descriptions and emit C++ code which will convert data formatted according to these descriptions
2. it runs this generator with the available syntax descriptions as inputs
3. it compiles the emitted C++ code into a final **dvbcodec** executable

I ran into a problem during the second phase of this process. The **dvbcodec** generator no longer parsed all of the supplied syntax descriptions. Specifically, I was seeing this conditional test raise an exception when trying to parse section format syntax descriptions.

```
std::string
upperCase(std::string const & lower)
{
  std::string upper = lower;
  for (std::string<char>::iterator cc =
     upper.begin();
  cc != upper.end(); ++cc)
  {
    * cc = std::toupper(* cc);
  }
  return upper;
}
```

Listing 1

```
std::string
upperCase(std::string const & lower)
{
  std::string upper = lower;
  std::transform(upper.begin(), upper.end(),
     upper.begin(), ::toupper);
  return upper;
}
```

Listing 2

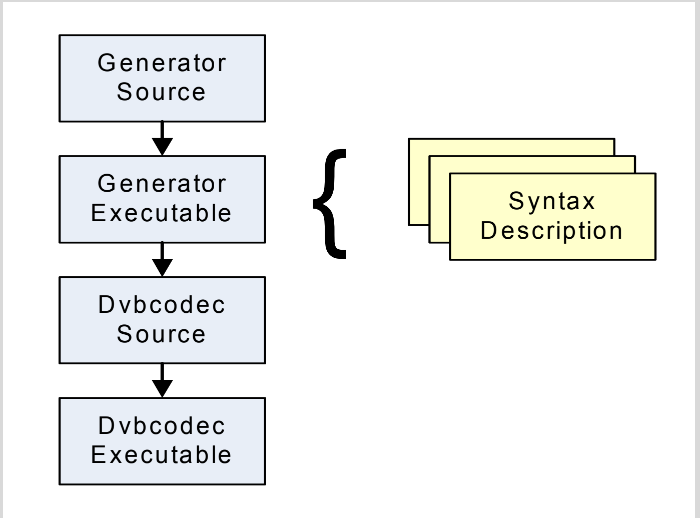

Figure 4

```
if (!parse(section_format,
           section_grammar,
           space_p).full)
{
  throw SectionFormatParseException(
     section_format);
}
```

Here, parse is **`boost::spirit::classic::parse`**, which parses something – the section format syntax description, passed as a string in this case – according to the supplied grammar. The third parameter, **`boost::spirit::classic::space_p`**, is a skip parser which tells parse to skip whitespace between tokens. **`Parse`** returns a **`parse_info`** struct whose **`full`** field is a boolean which will be set to **`true`** if the input section format has been fully consumed.

I soon figured out that the parse call was failing to fully consume binary syntax descriptions with trailing spaces, such as the the one shown below.

```
" program_association_section() {"
"    table_id                   8"
"    section_syntax_indicator   1"
"    '0'                        1"
....
"    CRC_32                    32"
" }                              "
```

If I stripped the trailing whitespace after the closing brace before calling **`parse()`** all would be fine. I wasn't fine about this fix though. The Spirit documentation is very good but it had been a while since I'd read it and, as already mentioned, my code used the 'classic' version of Spirit, in danger of becoming the 'legacy' then 'deprecated' and eventually the 'dead' version. Re-reading the documentation it wasn't clear to me exactly what the correct behaviour of **`parse()`** should be in this case. Should it fully consume trailing space? Had my program ever worked?

I went back in time, downloading and building against Boost 1.31, and satisfied myself that my code used to work, though maybe it worked due to a bug in the old version of Spirit. Stripping trailing spaces before parsing allowed my code to work with Spirit past and present, so I curtailed my investigation and made the fix.

(Interestingly, Boost 1.31 found a way to warn me I was using a compiler it didn't know about.

```
boost_1_31_0/boost/config/compiler/gcc.hpp:92:7:
warning:
#warning "Unknown compiler version - please run the
configure tests and report the results"
```

I ignored this warning.)

Code inaction

Apologies for the lengthy explanation in the previous section. The point is that few software projects stand alone, and that changes in any dependencies, **including bug fixes**, can have knock on effects. In this instance, I consider myself lucky; **`dvbcodec`**'s unusual three phase build enabled me to catch a runtime error. Of course, to actually catch that error, I needed to at least try building my code.

Put more simply: if you don't use your code, it rots.

Rotten artefacts

It wasn't just the code which had gone off. My source distribution included documentation – the plain text version of the article I'd written for Overload – and the Makefile had a build target to generate an HTML version of this documentation. This target depended on Quickbook, another Boost tool. Quickbook generates Docbook XML from plain text source, and Docbook is a good starting point for HTML, PDF and other standard output formats.

This is quite a sophisticated toolchain. It's also one I no longer use. Most of what I write goes straight to the web and I don't need such a fiddly process just to produce HTML. So I decided to freshen up dead links, leave the original documentation as a record, and simply cut the documentation target from the Makefile.

Stopping the rot

As we've seen, software, like other soft organic things, breaks down over time. How can we stop the rot?

Freezing software to a particular executable built against a fixed set of dependencies to run on a single platform is one way – and maybe some of us still have an aging Windows 95 machine, kept alive purely to run some such frozen program.

A better solution is to actively tend the software and ensure it stays in shape. Exercise it daily on a build server. Record test results. Fix faults as and when they appear. Review the architecture. Upgrade the platform and dependencies. Prune unused features, splice in new ones. This is the path taken by the Boost project, though certainly the growth far outpaces any pruning (the Boost 1.39 download is 5 times bigger than its 1.31 ancestor). Boost takes forwards and backwards compatibility seriously, hence the ongoing support for Spirit classic and the compiler version certification headers. Maintaining compatibility can be at odds with simplicity.

There is another way too. Although the **`dvbcodec`** project has collapsed into disrepair the idea behind it certainly hasn't. I've taken this same idea – of parsing formal syntax descriptions to generate code which handles binary formatted data – and enhanced it to work more flexibly and with a wider range of inputs. Whenever I come across a new binary data structure, I paste its syntax into a text file, regenerate the code, and I can work with this structure. Unfortunately I can't show you any code (it's proprietary) but I hope I've shown you the idea. Effectively, the old C++ code has been left to rot but the idea within it remains green, recoded in Python. Maybe I should find a way to humanely destroy the C++ and all links to it, but for now I'll let it degrade, an illustration of its time.

Is it possible that software is not like anything else, that it is meant to be discarded: that the whole point is to see it as a soap bubble?

Alan J. Perlis

Thanks

I would like to thank to Rick Engelbrecht for reporting and helping to fix the bugs discussed in this article. My thanks also to the team at Overload for their expert help. ■

References

[DVBcodec] Download of the DVBcodec is available from: http://wordaligned.org/docs/dvbcodec/dvbcodec-1.0.zip

[Guest04] Thomas Guest, 'A Mini-project to Decode a Mini-language - Part One', *Overload* #63, October 2004. Available from: http://accu.org/index.php/journals/241

[Matthews03] Hubert Matthews, 'CheckedInt: A Policy-Based Range-Checked Integer', *Overload* #58, December 2003. Available from: http://accu.org/index.php/journals/324

[Spirit] 'Spirit User's Guide' Available from: http://www.boost.org/doc/libs/1_39_0/libs/spirit/classic/index.html

[W3C] 'Cool URIs don't change' Available from: http://www.w3.org/Provider/Style/URI

The Model Student: A Primal Skyline (Part 1)

Prime numbers are the ‘building blocks’ of the integers. Richard Harris investigates how they’re combined.

The prime numbers, those positive integers wholly divisible by only themselves or by one, are perhaps the most studied numbers in all of history. Evidently a breed apart from their more mundane neighbours on the number line they are, depending upon how much number theory you have been subjected to, their noble elite, their rugged individualists, or their psychopathic loners.

Every integer can be represented as the product of a set of primes, known as the prime factors. For example the number 42 has the prime factors 2, 3, and 7 since $42 = 2 \times 3 \times 7$.

Numbers with more than one prime factor (i.e. non-primes) are known as composite numbers and the number 1 is technically known as the identity and is neither prime nor composite.

Now, in general, the prime factors of a number may contain multiple copies of each given prime. We can capture this by raising each factor to a power representing how many times it shows up in the factorisation. For example

$$252 = 2 \times 2 \times 3 \times 3 \times 7 = 2^2 \times 3^2 \times 7^1$$

Noting that raising a number to the power of 0 results in 1, we can propose an alternative notation for the integers. Identifying the first, second, third and so on entries in a list as the powers to which the first, second and third and so on primes should be raised in the product, and in much the same way as we truncate trailing zeros in decimal notation, truncating trailing zeros in our list of prime powers, we have a unique representation for every integer. For example

$$252 = 2^2 \times 3^2 \times 5^0 \times 7^1 \rightarrow 2, 2, 0, 1$$

since 2, 3, 5 and 7 are the 1st, 2nd, 3rd and 4th primes.

Whilst this happens to be a supremely convenient notation in which to perform multiplication, it is an atrocious notation for addition, which perhaps explains why we don’t use it.

Compounding the lack of usefulness of this notation is the fact that it is actually rather difficult to identify the n’th prime. Generally, the prime numbers are notoriously difficult to find, which is unfortunate since they lie at the heart of many of the great unanswered mathematical questions of the 21st century. Such as, for example, whether the Riemann Hypothesis is true [duSautoy04], or whether it is possible to efficiently decompose numbers into their constituent prime factors [Menezes97].

Euclid’s proof of the infinity of the primes

It was Euclid who took the first timid steps towards subjugating these aristocrats cum robber barons of the integers by demonstrating, over two thousand years ago, that there are infinitely many of them.

His proof is elegant and simple, and as such is a rare and precious gem of number theory. It is an example of proof by contradiction in which we assume that there are only a finite number of primes and then demonstrate that this leads to a contradiction.

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

So, assuming there are only n primes for some undetermined value of n, we can write the product of all of them as follows:

$$\prod_{i=1}^{n} p_i$$

where p_i is the i’th prime and the capital pi means the result of multiplying together all of the values from p_1 to p_n. In this sense it is much like the capital sigma we use to represent the sum of a set of numbers.

Now, this number is trivially composite since it can be divided by *every* prime. However, consider the result of adding 1 to it:

$$1 + \prod_{i=1}^{n} p_i$$

Now, dividing this number by any of the primes leaves a remainder of 1, since the product is divisible by all of them and the 1 by none of them.

It is, by definition, greater than any of the primes in our set and hence cannot be one of them. Furthermore, since it is not wholly divisible by any of them it must either be a prime itself or have a prime factor that is not in our set. Hence our set is incomplete, no matter what value n takes and there must therefore be an infinite number of primes.

Sweet.

Having concluded that the primes are infinite in number, the next obvious question to ask is how densely packed they are amongst the integers; how many primes are there less than or equal to any given integer n?

The prime number theorem

We have an approximate answer to this question, at least for large n, first guessed at by the mathematical giants Legendre and Gauss in the late 18th century:

$$\pi(n) \approx \frac{n}{\ln n}$$

The drunkenly scribed equals sign means approximately equal to, the lower case pi is the function that returns the number of primes less than or equal to its argument n and the ln is the natural logarithm; the number to which we need to raise the mathematical constant e (approximately 2.718) to recover the argument.

It took about 100 years to raise the status of this formula from conjecture to theorem, when it was tortuously proven by both Vallée-Poussin and Hadamard [Daintith89].

Technically, the theorem states that the ratio between the number of primes and this formula tends to 1 as n grows larger.

$$\lim_{n \to \infty} \frac{\pi(n)}{n / \ln n} = 1$$

it could easily be used to determine the distribution of the primes amongst the positive integers

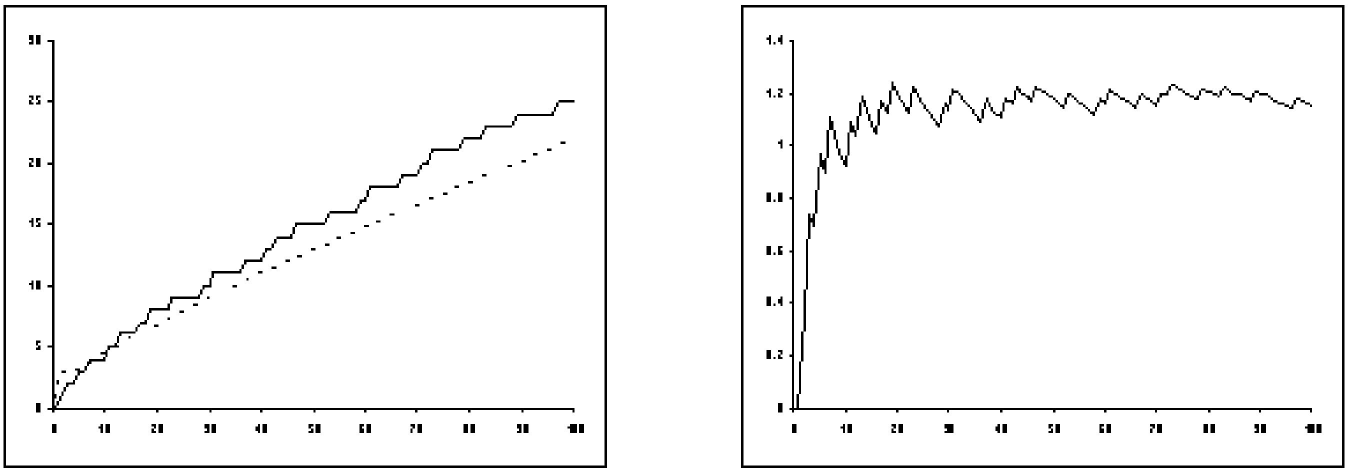

Figure 1

The lim term here indicates that we are describing the limit of the expression that follows it as *n* grows larger and larger, or tends to infinity.

Figure 1 plots the function counting the primes as the solid line in the graph on the left, the approximation as the dotted line and the ratio between them in the graph on the right.

Clearly, the approximation isn't particularly accurate in this range. Which is a shame since it could easily be used to determine the distribution of the primes amongst the positive integers, which is one of the most fundamental puzzles in number theory.

Those of you who have been following this, as it turns out rather formulaic, series of articles will not be in the least bit surprised when I abandon all attempt to address the question at hand and introduce a simpler one which I, with my limited mathematical arsenal, may actually be capable of shedding some light upon.

Ready.

Get set.

The X factors

Rather than attempt to investigate the distribution of the primes, I shall instead propose that we consider looking for a pattern in the prime factorisations of the integers. As mentioned at the start of this article, every positive integer can be represented by the product of a set of primes, with 1 being the special case of multiplying together no primes.

The simplest method of factorising integers, known as trial division, is to iterate through all of the prime numbers less than a given integer, increasing the powers of each whilst they leave no remainder upon dividing it.

Listing 1 illustrates a function that prints out the prime factors of its argument using this algorithm.

Note that if *x* is sufficiently large then repeated multiplication of the **`factor`** variable by the *n*'th prime might exceed the maximum representable value of an **`unsigned long`**.

Fortunately, this is not technically an overflow and hence does not invoke the dreaded undefined behaviour. This is because the C++ standard defines arithmetic with *n* bit unsigned integer types as being modulo 2^n, effectively throwing away any unrepresentable bits in the result of an arithmetic expression and wrapping round into the range of representable values [ANSI].

```
void
print_factors(unsigned long x)
{
  unsigned long n = 0;
  while(nth_prime(n)<=x)
  {
    unsigned long power  = 0;
    unsigned long factor = nth_prime(n);
    while(x%factor==0)
    {
      ++power;
      factor *= nth_prime(n);
    }
    if(power!=0) std::cout << nth_prime(n) <<
       "^" << power << " ";
    ++n;
  }
}
```

Listing 1

we must be careful that we identify any **single prime factor** that may be **larger than the square root** of the number

Unfortunately, this doesn't really help us all that much. For example, if *x* is equal to 2^{n-1}, the `result` variable will eventually wrap around to 0 and the next step of the loop will involve a divide by 0 error during the modulus calculation.

Leaving this problem and the definition of the `nth_prime` function aside for now, we shall instead focus on some performance improvements that we can make to this approach.

The first thing we should note is that the largest repeated factor of a compound number must be no larger than the square root of that number. Indeed, if this were not so, then the product of the least possible number of repeated factors, 2, would exceed our compound number and could clearly not be equal to it.

In exploiting this fact, we must be careful that we identify any single prime factor that may be larger than the square root of the number. We can do this by keeping track of the product of the factors so far identified; if this is not equal to the original number then there must be one more unrepeated prime factor equal to the original number divided by the product of the identified factors.

Listing 2 illustrates the changes we need to make to the `print_factors` function to implement this improvement.

Unfortunately, we have introduced another sensitivity to integer wrap around. If *x* is sufficiently large then the square of the smallest prime strictly greater than it could wrap around. This means that we may very well enter into an infinite loop, never finding a prime whose square, modulo 2 to the power of the number of bits in an `unsigned long`, is greater than *x*.

Ignoring this potential problem too, we finally note that even if we have found all of the factors of the number we will still keep looking until we reach the last prime less than or equal to its square root.

A further improvement is therefore to divide the number by each factor we discover, allowing us to stop when we reach a prime larger than the square root of the product of the remaining factors, if any.

If this remaining product is anything other than 1, it must be a single non-repeated prime factor. To prove this, recall that every compound number must have at least one factor no greater than its square root and since we have already removed all such factors it cannot therefore be a compound number.

This final change to the trial division algorithm is illustrated in listing 3.

The performance improvement from this change is significantly less dramatic than that of the first, as illustrated in figure 2, which gives the

```
void
print_factors(unsigned long x)
{
  unsigned long n = 0;
  unsigned long factors = 1;

  while(nth_prime(n)*nth_prime(n)<=x)
  {
    unsigned long power  = 0;
    unsigned long factor = nth_prime(n);

    while(x%factor==0)
    {
      ++power;
      factor  *= nth_prime(n);
      factors *= nth_prime(n);
    }

    if(power!=0)  std::cout << nth_prime(n)
       << "^" << power << " ";
    ++n;
  }

  if(x!=0 && factors!=x) std::cout << x/factors
     << "^1 ";
}
```

Listing 2

```
void
print_factors(unsigned long x)
{
  unsigned long n = 0;

  while(nth_prime(n)*nth_prime(n)<=x)
  {
    unsigned long power = 0;

    while(x%nth_prime(n)==0)
    {
      ++power;
      x /= nth_prime(n);
    }

    if(power!=0)  std::cout << nth_prime(n)
       << "^" << power << " ";
    ++n;
  }

  if(x>1)  std::cout << x << "^1 ";
}
```

Listing 3

First Attempt	0.53s
Second Attempt	0.14s
Third Attempt	0.12s

Figure 2

we can use the **prime number theorem** to get an estimate of **how large the sequence of primes** might be

time each version of the algorithm takes to factor (but not print) the integers from 2 to 100 using the machine with which I happen to be writing this article 10,000 times each.

That said, it does neatly side step the possible wrap around issue whilst multiplying the **`factor`** variable by the *n*'th prime, although it does not address that of squaring the *n*'th prime.

The n'th prime

So, given that we don't have an exact formula for the *n*'th prime, how are we to go about implementing the **`nth_prime`** function?

To be perfectly honest, we can't; the function I used to test the various implementations of the **`print_factors`** function used a look up table of the primes between 0 and 100, which I'm sure you'll agree isn't particularly scalable.

However, if we are interested in the factorisations of all numbers up to a given upper bound, which I can assure you we shall be, we can build the table of primes as we go.

Instead of providing a function to calculate the primes, we will provide a pair of iterators that range from the first prime, 2, to a prime guaranteed to be no smaller than the square root of the number we seek to factor. Furthermore we change the function to return a **`bool`** to indicate whether or not the number in question remained unchanged throughout the trial division and is itself therefore a prime

Listing 4 illustrates the changes we need to make to the **`print_factors`** function.

We can now supplement this with a second function that iterates from 0 (or strictly speaking a non-negative number less than or equal to 2) up to some upper bound printing the factorisation of each of them.

This second function can use the result of **`print_factors`** to add new primes, up to the square root of the upper bound, to the back of the sequence that the iterators range over.

Note that we can use the prime number theorem to get an estimate of how large the sequence of primes might be. By multiplying this estimate by some constant factor sufficiently greater than 1, we can ensure that it will exceed the number of primes in almost all cases. This, in turn, ensures that in almost all cases we can reserve enough space for all of the primes we'll need in a **`std::vector`**. Of course, this is a ridiculous micro-optimisation, but we shall eventually be desperate for simple ways to squeeze out those last few wasted cycles.

Moving on from that rather vague justification for my apparent performance anxiety, we shall implement this as an overload of the **`print_factors`** function as illustrated in listing 5.

```
template<class FwdIt>
bool
print_factors(unsigned long x, FwdIt first_prime,
   FwdIt last_prime)
{
  const unsigned long x0 = x;

  while(first_prime!=last_prime &&
      (*first_prime)*(*first_prime)<=x)
  {
    unsigned long power = 0;

    while(x%*first_prime==0)
    {
      ++power;
      x /= *first_prime;
    }

    if(power!=0)  std::cout << *first_prime
       << "^" << power << " ";
    ++first_prime;
  }

  if(x>1)  std::cout << x << "^1 ";
  return x0>1 && x==x0;
}
```

Listing 4

```
void
print_factors(unsigned long upper_bound)
{
  std::vector<unsigned long> primes;

  const double pi_upper_bound =
     sqrt(double(upper_bound)) /
     log(sqrt(double(upper_bound)));

  const unsigned long n(1.5*pi_upper_bound);
  primes.reserve(n);

  unsigned long x = 1;
  while(x<upper_bound)
  {
    std::cout << x << ": ";
    bool is_prime =
       print_factors(x,primes.begin(),
       primes.end());
    std::cout << std::endl;

    if(is_prime && x*x<=upper_bound)
       primes.push_back(x);
    ++x;
  }
}
```

Listing 5

analysing the complete factorisations of **ranges of integers** is something of a tall order

```
1:
2: 2^1
3: 3^1
4: 2^2
5: 5^1
6: 2^1 3^1
7: 7^1
8: 2^3
9: 3^2
10: 2^1 5^1
11: 11^1
12: 2^2 3^1
13: 13^1
14: 2^1 7^1
15: 3^1 5^1
16: 2^4
17: 17^1
18: 2^1 3^2
19: 19^1
20: 2^2 5^1
```

Figure 3

Note that this function too suffers from potential integer wrap around whilst squaring prime numbers when we're checking whether to add them to our list.

The output of this function for the integers from 1 to 20 is given in figure 3. Note that since 1 is neither prime nor compound, it has no factors.

Omega? Feh!

Now, analysing the complete factorisations of ranges of integers is something of a tall order, so instead I suggest we simply count how many prime factors each integer has. A cousin of the prime counting function Π, the function that returns the number of prime factors (including repeated factors) of its argument is denoted by Ω.

n	Ω(n)
0	-∞
1	0
2	1
3	1
4	2
5	1
6	2

n	Ω(n)
7	1
8	3
9	2
10	2
11	1
12	3
13	1

n	Ω(n)
14	2
15	2
16	4
17	1
18	3
19	1
20	3

Figure 4

```
template<class FwdIt>
unsigned long
count_factors(unsigned long x, FwdIt first_prime,
   FwdIt last_prime)
{
  unsigned long count = 0;

  while(first_prime!=last_prime &&
     (*first_prime)*(*first_prime)<=x)
  {
    while(x%*first_prime==0)
    {
      ++count;
      x /= *first_prime;
    }

    ++first_prime;
  }

  if(x>1)  ++count;
  return count;
}
```

Listing 6

Using the factorisations we calculated in figure 3, and noting that 0 is the result of dividing 1 by infinitely many factors, we can derive the values of Ω for the integers from 0 to 20, as illustrated in figure 4.

This function has some useful properties, not least of which is that it maps non-negative integers to integers, assuming we're happy to count negative infinity as an integer. Furthermore, every number for which this function evaluates to 1 is, by definition, a prime; it has, after all only 1 prime factor.

It would be extremely tedious to work out the values of Ω from the factorisations we can currently produce. Fortunately, we can simply adapt the functions to count the factors instead. Listing 6 illustrates the function to count the factors of a single integer.

We shall overload this to count the factors of every positive integer up to some upper bound, in much the same way as we did for `print_factors`. The single integer function no longer returns a `bool` to indicate that the argument is a prime, but as noted above if a number has precisely 1 factor it must be prime and we can use this fact as the indication that we should add the number to the back of our sequence of primes. This second function is given in listing 7.

The output of this function for the integers from 1 to 20 is given in figure 5, which you can see is in agreement with our hand derived values for Ω.

The graphs of Ω for the integers from 1 to 20 and from 1 to 100 are given in figure 6. We extend this from the integers to the real numbers by plotting the value of Ω for the integer part of each real number.

It is tempting to look for patterns in these graphs and there is, in fact, a particularly striking one. To see it we need to look at an exponential function of Ω; specifically raising 2 to its power.

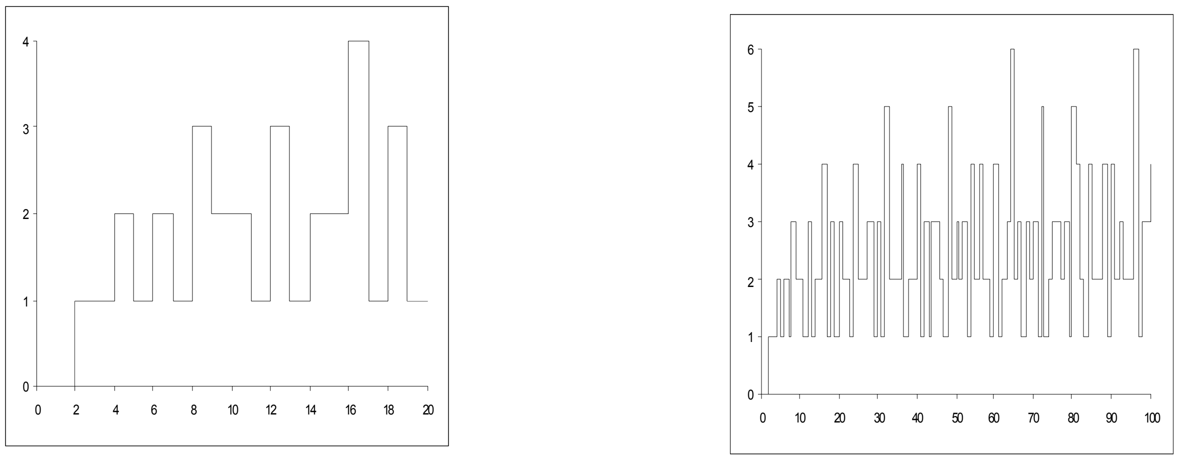

Figure 6

```
void
count_factors(unsigned long upper_bound)
{
  std::vector<unsigned long> primes;

  const double pi_upper_bound =
     sqrt(double(upper_bound)) /
     log(sqrt(double(upper_bound)));
  const unsigned long n(1.5*pi_upper_bound);
  primes.reserve(n);
  unsigned long x = 1;
  while(x<upper_bound)
  {
    const unsigned long count = count_factors(
       x, primes.begin(),primes.end());
    std::cout << x << ": " << count << std::endl;
    if(count==1 && x*x<=upper_bound)
       primes.push_back(x);
    ++x;
  }
}
```

Listing 7

```
1: 0
2: 1
3: 1
4: 2
5: 1
6: 2
7: 1
8: 3
9: 2
10: 2
11: 1
12: 3
13: 1
14: 2
15: 2
16: 4
17: 1
18: 3
19: 1
20: 3
```

Figure 5

In keeping with the time honoured tradition of naming mathematical functions with single letters from non-Latin alphabets, I christen this function $ד_n$.

$$_n(x)= \frac{2^{\Omega(\lfloor 2^n x \rfloor)}}{2^n} \qquad x \in [0, 1]$$

You may recall from previous articles that the odd square brackets surrounding the $2^n x$ terms means the largest integer less than or equal to the value between them and that the expression on the right with the rounded E means that x must be in the range 0 to 1. Figure 7 illustrates the graphs of $ד_5$ and $ד_7$.

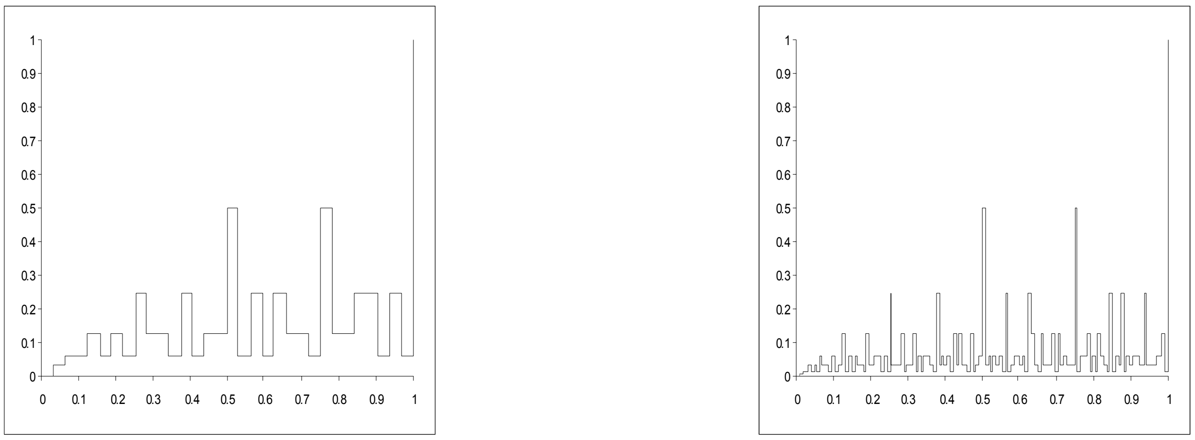

Figure 7

First we assume that the graph *can* rise above the line $y = x$ and show that this leads to a contradiction. This assumption means that for some x

$$\daleth_n(x) > x$$

implying that

$$\frac{2^{\Omega(\lfloor 2^n x \rfloor)}}{2^n} > x$$

$$2^{\Omega(\lfloor 2^n x \rfloor)} > 2^n x$$

$$2^{\Omega(i)} > 2^n \frac{i+\delta}{2^n} \quad \text{for some integer } i \text{ and real number } \delta \geq 0$$

$$2^{\Omega(i)} > i + \delta$$

Now, since the left hand side of the inequality is an integer and δ is greater than or equal to 0, we have

$$2^{\Omega(i)} > i$$

Of course, this is impossible since 2 is the smallest prime number and hence $2^{\Omega(i)}$ must be less than or equal to i and consequently $\daleth_n(x)$ must be less than or equal to x.

Derivation 1

The properties of $\daleth_n$

For any n, $\daleth_n$ is defined for arguments between 0 and 1. Moreover, at 0 it returns a value of 0 and at 1 it returns a value of 1.

To demonstrate this, we note firstly that for x equal to 0, Ω receives an argument of 0 and returns negative infinity. Since any number greater than 1 raised to negative infinity yields 0, $\daleth_n$ returns 0 for an argument of 0.

Secondly, the integer between 0 and 2^n with the most factors is 2^n, since 2 is the smallest prime. This has n factors and hence the top and bottom of the fraction defining $\daleth_n$ are equal when x equals 1, yielding a result of 1.

In fact, the entire graph must never rise above the line from (0,0) to (1,1) as proven in derivation 1.

Another statement that we can make about these curves is that for all n greater than 0, $\daleth_n$ must coincide with $\daleth_{n-1}$ for half of the values of x; specifically, those where the integer part of 2_nx is even, as demonstrated in derivation 2.

The curves $\daleth_n$ and $\daleth_{n-1}$ are further related by the equation

$$\daleth_n(\tfrac{1}{2}x) = 2\daleth_n(\tfrac{1}{2}x)$$

as shown in derivation 3.

If we combine these two properties then we recover the striking pattern that I alluded to above; specifically that, when the integer part of $2^n x$ is even, we have

$$\daleth_n(x) = 2\daleth_n(\tfrac{1}{2}x)$$

Now this may not strike you as being especially striking, but it is strikingly similar to a property exhibited by many of a very well known class of curves; the fractals.

Noting that $\lfloor a \rfloor = 2 \times \left\lfloor \frac{1}{2}a \right\rfloor$ for even $\lfloor a \rfloor$, for even $\lfloor 2_n x \rfloor$ we have

$$\daleth_n(x) = \frac{2^{\Omega(\lfloor 2^n x \rfloor)}}{2^n} = \frac{2^{\Omega(2 \times \lfloor 2^{n-1} x \rfloor)}}{2^n} = \frac{2^{\Omega(\lfloor 2^{n-1} x \rfloor)+1}}{2^n}$$

$$= \frac{2^{\Omega(\lfloor 2^{n-1} x \rfloor)} \times 2}{2^n} = \frac{2^{\Omega(\lfloor 2^{n-1} x \rfloor)}}{2^{n-1}} = \daleth_{n-1}(x)$$

Derivation 2

$$\daleth_n(\tfrac{1}{2}x) = \frac{2^{\Omega(\lfloor 2^n \frac{1}{2} x \rfloor)}}{2^n} = \frac{2^{\Omega(\lfloor 2^{n-1} x \rfloor)}}{2^n}$$

$$= \frac{1}{2}\frac{2^{\Omega(\lfloor 2^{n-1} x \rfloor)}}{2^{n-1}} = \frac{1}{2}\daleth_{n-1}(x)$$

Note that since we haven't exploited the properties of Ω, this would hold if we replaced it with any other function.

Derivation 3

Many fractals are defined in terms of a sequence of curves which serve as closer and closer approximations to the fractal itself. These curves can themselves be approximated by zooming in on sub-sections of them, in much the same way as we can for $\daleth_n$. The major difference is that for the iterative approximations of fractals we can zoom in on many parts of the curve rather than just one specific region.

We can uncover further hints at the relationship between fractals and $\daleth_n$ by considering the limit as n tends to infinity. Waving our hands somewhat vigorously we can take the result that

$$\daleth_n(\tfrac{1}{2}x) = \tfrac{1}{2}\daleth_{n-1}(x)$$

and infer that

$$\daleth_\infty(\tfrac{1}{2}x) = \tfrac{1}{2}\daleth_\infty(x)$$

since $\daleth_{n-1}$ is approximately equal to $\daleth_n$ for arbitrarily accurate interpretations of approximately.

Now, instead of recovering an approximation of the curve by zooming in on a sub-section of it, we can entirely reconstruct it. Specifically, for all values of x, we have

$$\daleth_\infty(x) = 2\daleth_\infty(\tfrac{1}{2}x) = 4\daleth_\infty(\tfrac{1}{4}x) = 8\daleth_\infty(\tfrac{1}{8}x) = \ldots$$

This is suspiciously similar to the self-similarity property of many fractals; the ability to reconstruct the entire curve by zooming in on sub-sections of it.

So is $\daleth_\infty$ itself a fractal?

Well, that happens to be a very interesting question and we shall pursue its answer in the next article.

Until then, dear reader, fare well. ■

Acknowledgements

With thanks John Paul Barjaktarevi? and Lee Jackson for proof reading this article.

References and further reading

[ANSI] *The C++ Standard*, American National Standards Institute

[Daintith89] Daintith, J. & Nelson, R. (ed), *The Penguin Dictionary of Mathematics*, Penguin, 1989

[duSautoy04] du Sautoy, M., *The Music of the Primes*, Harper Perennial, 2004.

[Menezes97] Menezes, A. et al, *Handbook of Applied Cryptography*, CRC Press, 1997

The Generation, Management and Handling of Errors (Part 1)

An error handling strategy is important for robustness. Andy Longshore and Eoin Woods present a pattern language.

In recent years there has been a wider recognition that there are many different stakeholders for a software project. Traditionally, most emphasis has been given to the end user community and their needs and requirements. Somewhere further down the list is the business sponsor; and trailing well down the list are the people who are tasked with deploying, managing, maintaining and evolving the system. This is a shame, since unsuccessful deployment or an unmaintainable system will result in ultimate failure just as certainly as if the system did not meet the functional requirements of the users.

One of the key requirements for any group required to maintain a system is the ability to detect errors when they occur and to obtain sufficient information to diagnose and fix the underlying problems from which those errors spring. If incorrect or inappropriate error information is generated from a system it becomes difficult to maintain. Too much error information is just as much of a problem as too little. Although most modern development environments are well provisioned with mechanisms to indicate and log the occurrence of errors (such as exceptions and logging APIs), such tools must be used with consistency and discipline in order to build a maintainable application. Inconsistent error handling can lead to many problems in a system such as duplicated code, overly-complex algorithms, error logs that are too large to be useful, the absence of error logs and confusion over the meaning of errors. The incorrect handling of errors can also spill over to reduce the usability of the system as unhandled errors presented to the end user can cause confusion and will give the system a reputation for being faulty or unreliable. All of these problems are manifest in software systems targeted at a single machine. For distributed systems, these issues are magnified.

This paper sets out a collection (or possibly a language) of patterns that relate to the use of error generating, handling and logging mechanisms - particularly in distributed systems. These patterns are not about the creation of an error handling mechanism such as [Harrison] or a set of language specific idioms such as [Haase] but rather in the application code that makes use of such underlying functionality. The intention is that these patterns combine to provide a landscape in which sensible and consistent decisions can be made about when to raise errors, what types of error to raise, how to approach error handling and when and where to log errors.

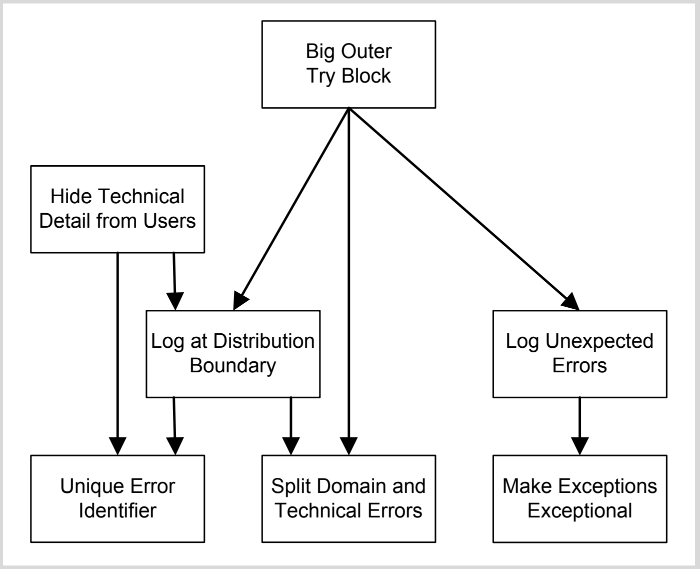
Big Outer Try Block
Hide Technical Detail from Users
Log at Distribution Boundary
Log Unexpected Errors
Unique Error Identifier
Split Domain and Technical Errors
Make Exceptions Exceptional

Figure 1

Overview

The patterns presented in this paper form a pattern collection to guide error handling in multi-tier distributed information systems. Such systems present a variety of challenges with respect to error handling, including the distribution of elements across nodes, the use of different technology platforms in different tiers, a wide variety of possible error conditions and an end-user community that must be shielded from the technical details of errors that are not related to their use of the system. In this context, a software designer must make some key decisions about how errors are generated, handled and managed in their system. The patterns in this paper are intended to help with these system-wide decisions such as whether to handle domain errors (errors in business logic) and technical errors (platform or programming errors) in different ways. This type of far-reaching design decision needs careful thought and the intent of the patterns is to assist in making such decisions.

As mentioned above, the patterns presented here are not detailed design solutions for an error handling framework, but rather, are a set of design principles that a software designer can use to help to ensure that their error handling approach is coherent and consistent across their system. This approach to pattern definition means that the principles should be applicable to a wide variety of information systems, irrespective of their implementation technology. We are convinced of the applicability of these patterns in their defined domain. You may also find that they are applicable to systems in other domains – if so then please let us know.

The patterns in the collection are illustrated in Figure 1.

The boxes in the diagram each represent a pattern in the collection. The arrows indicate dependencies between the patterns, with the arrow running from a pattern to another pattern that it is dependent upon. For example,

Andy Longshaw works for Barclays Bank delivering IT solutions with particular focus on reusability. He has been delivering and explaining technology and system architecture for most of the last decade. He can be contacted at www.blueskyline.com

Eoin Woods is a software architect at Barclays Global Investors, heading the application architecture group.He has been working in software engineering for nearly 20 years and is co-author of the book 'Software Systems Architecture'. He can be contacted at www.eoinwoods.info

technical errors are, by their very nature, **difficult to predict**

Expected vs. unexpected and domain vs. technical errors

This pattern language classifies errors as 'domain' or 'technical' and also as 'expected' and 'unexpected'. To a large degree the relationship between these classifications is orthogonal. You can have an expected domain error (no funds in the account), an unexpected domain error (account not in database), an expected technical error (WAN link down – retry), and an unexpected technical error (missing link library). Having said this, the most common combinations are expected domain errors and unexpected technical errors.

A set of domain error conditions should be defined as part of the logical application model. These form your expected domain errors. Unexpected domain errors should generally only occur due to incorrect processing or mis-configuration of the application.

The sheer number of potential technical errors means that there will be a sizeable number that are unexpected. However, some technical errors will be identified as potentially recoverable as the system is developed and so specific error handling code may be introduced for them. If there is no recovery strategy for a particular error it may as well join the ranks of unexpected errors to avoid confusion in the support department ('why do they catch this and then re-throw it…').

The table below illustrates the relationship between these two dimensions of error classification and the recommended strategy for handling each combination of the two dimensions, based on the strategies contained in this collection of patterns.

	Expected	Unexpected
Domain	■ Handle in the application code ■ Display details to the user ■ Don't log the error	■ Throw an exception ■ Display details to the user ■ Log the error
Technical	■ Handle in the application code ■ Don't display details to the user ■ Don't log the error	■ Throw an exception ■ Don't display details to the user ■ Log the error

to implement Log Unexpected Errors you must first Make Exceptions Exceptional. In turn, the Logging of Unexpected Errors supports a Big Outer Try Block. You can see that to get the most benefit from the set of patterns it is best to use the whole set in concert.

At the end of the paper, a set of proto-patterns is briefly described. These are considered to be important concepts that may or may not become fully fledged patterns as the paper evolves.

Split Domain and Technical Errors

Problem

Applications have to deal with a variety of errors during execution. Some of these errors, that we term 'domain errors', are due to errors in the business logic or business processing (e.g. wrong type of customer for insurance policy). Other errors, that we term 'technical errors', are caused by problems in the underlying platform (e.g. could not connect to database) or by unexpected faults (e.g. divide by zero). These different types of error occur in many parts of the system for a variety of reasons. Most technical errors are, by their very nature, difficult to predict, yet if a technical error could possibly occur during a method call then the calling code must handle it in some way.

Handling technical errors in domain code makes this code more obscure and difficult to maintain.

Context

Domain and technical errors form different 'areas of concern'. Technical errors 'rise up' from the infrastructure – either the (virtual) platform, e.g. database connection failed, or your own artifacts, e.g. distribution facades/proxies. Business errors arise when an attempt is made to perform an incorrect business action. This pattern could apply to any form of application but is particularly relevant for complex distributed applications as there is much more infrastructure to go wrong!

Forces

- If domain code handles technical errors as well as domain ones, it becomes unnecessarily complex and difficult to maintain.
- A technical error can cause domain processing to fail and the system should handle this scenario. However, it can be difficult (or impossible) to predict what types of technical errors will occur within any one piece of domain code.
- It is common practice to handle technical errors at a technical boundary (such as a remote boundary). However, such a boundary should be transparent to domain errors.
- For some technical errors, it may be worth taking certain actions such as retrying (e.g. retry a database connection). However, such an action may not make sense for a domain error (e.g. no funds) where the inputs remain the same.
- As part of the specification of a system component, all of the potential domain errors originating from a domain action should be predictable and testable. However, changes in implementation may vary the number and type of technical errors that may possibly arise from any particular action.
- Technical and domain errors are of interest to different system stakeholders and will be resolved by members of different stakeholder groups.

Solution

Split domain and technical error handling. Create separate exception/error hierarchies and handle at different points and in different ways as appropriate.

Implementation

Errors in the application should be categorized into domain errors (aka. business, application or logical errors) and technical errors. When you

business error handling code will be in a completely different part of the code to the technical error handling

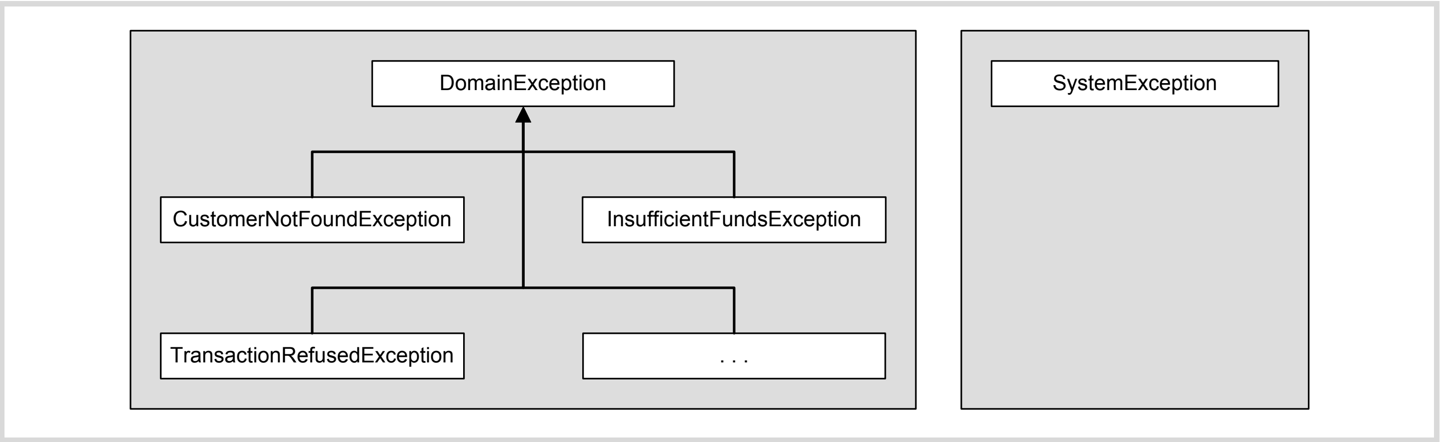
DomainException
CustomerNotFoundException
InsufficientFundsException
TransactionRefusedException
. . .
SystemException

Figure 2

create your exception/error hierachy for your application, you should define your domain errors and a single error type to indicate a technical error, e.g. **`SystemException`** (see Figure 2). The definition and use of a single technical error type simplifies interfaces and prevents calling code needing to understand all of the things that can possibly go wrong in the underlying infrastructure. This is especially useful in environments that use checked exceptions (e.g. Java).

Design and development policies should be defined for domain and technical error handling. These policies should include:

- A technical error should never cause a domain error to be generated (never the twain should meet). When a technical error must cause business processing to fail, it should be wrapped as a **`SystemError`**.
- Domain errors should always start from a domain problem and be handled by domain code.
- Domain errors should pass 'seamlessly' through technical boundaries. It may be that such errors must be serialized and re-constituted for this to happen. Proxies and facades should take responsibility for doing this.
- Technical errors should be handled in particular points in the application, such as boundaries (see LOG AT DISTRIBUTION BOUNDARY).
- The amount of context information passed back with the error will depend on how useful this will be for subsequent diagnosis and handling (figuring out an alternative strategy). You need to question whether the stack trace from a remote machine is wholly useful to the processing of a domain error (although the code location of the error and variable values at that time may be useful).

As an example, consider the exception definitions in Listing 1.

A domain method skeleton could then look like Listing 2.

```
public class DomainException extends Exception
{
  ...
}
Public class InsufficientFundsException
   extends Exception
{
  ...
}
public class SystemException extends Exception
{
  ...
}
```

Listing 1

```
public float withdrawFunds(float amount)
    throws InsufficientFundsException,
    SystemException
{
  try
  {
    // Domain code that could generate various
    // errors both technical and domain
  }
  catch (DomainException ex)
  {
    throw ex;
  }
  catch (Exception ex)
  {
    throw new SystemException(ex);
  }
}
```

Listing 2

technical errors need to be recorded in a log that is **easily accessible from the same place** as the infrastructure's error logs

This method declares two exceptions: a domain error – lack of funds to withdraw – and a generic system error. However, there are many technical exceptions that could occur (connectivity, database, etc.). The implementation of this method passes domain exceptions straight through to the caller. However, any other error is converted to a generic `SystemException` that is used to wrap any other (non-domain) errors that occur. This means that the caller simply has to deal with the two checked exceptions rather than many possible technical errors.

Positive consequences

- The business error handling code will be in a completely different part of the code to the technical error handling and will employ different strategies for coping with errors.
- Business code needs only to handle any business errors that occur during its execution and can ignore technical errors making it easier to understand and more maintainable.
- Business error handling code can be clearer and more deterministic as it only needs to handle the subset of business errors defined in the contract of the business methods it calls.
- All potential technical errors can be handled in surrounding infrastructure (server-side skeleton, remote façade or main application) which can then decide if further business actions are possible.
- Different logging and auditing policies are easily applied due to the clear distinction of error types.

Negative consequences

- Two exception hierarchies need to be maintained and there may be situations where this is artificial or the right location for an exception is not immediately obvious.
- Domain errors need to be passed through infrastructure code - possibly by marshaling and unmarshaling them across the infrastructure boundary (typically a distribution boundary).

Related patterns

- Technical and domain errors should be treated differently at distribution boundaries as defined in LOG AT DISTRIBUTION BOUNDARY
- Unless they are handled elsewhere in the system, both technical and domain errors should be handled by a BIG OUTER TRY BLOCK
- It is common to apply the proto pattern SINGLE TYPE FOR TECHNICAL ERRORS
- A more general form of this pattern is described in EXCEPTION HIERARCHY [Renzel97]
- The use of a domain hierarchy in Java is also discussed in the EXCEPTION HIERARCHY idiom in [Haase]
- The HOMOGENOUS EXCEPTION and EXCEPTION WRAPPING Java idioms in [Haase] show how you might implement `SystemException` in Java.

LOG AT DISTRIBUTION BOUNDARY

Problem

The details of technical errors rarely make sense outside a particular, specialized, environment where specialists with appropriate knowledge can address them. Propagating technical errors between system tiers results in error details ending up in locations (such as end-user PCs) where they are difficult to access and in a context far removed from that of the original error.

Context

Multi-tier systems, particularly those that use a number of distinct technologies in different tiers.

Forces

- You could propagate all the error information back to original calling application where it could be logged by a BIG OUTER TRY BLOCK but the complete set of error information is bulky and may include platform-specific information.
- Technical error information needs to be made easily accessible to the technology specialists (such as operating system administrators and DBAs) who should be able to resolve the underlying problems.
- When administrators come to resolve problems that technical errors reveal, they will need to access the error logs used by other parts of the system infrastructure as well as using the information in the error logged by the application. In order to facilitate this, technical errors need to be recorded in a log that is easily accessible from the same place as the infrastructure's error logs.
- Each technology platform has its own formats and norms for error logging. In order to fit neatly into the technology environment, it is desirable that the new system uses an appropriate logging approach in each environment.
- To correctly diagnose technical errors that occur on a particular system, extra technical information is often required about the current runtime environment (such as number of database connections open) but adding the additional code needed to recover and record this information to the various layers of application code would make such code significantly more complex.
- The handling of errors should not impact the normal behaviour of the system unnecessarily. To reduce any impact it is desirable to avoid passing large quantities of error information around the system.

Solution

When technical errors occur, log them on the system where they occur passing a simpler generic **`SystemError`** back to the caller for reporting at the end-user interface. The generic error lets calling code know that there has been a problem so that they can handle it but reduces the amount of system-specific information that needs to be passed back through the distribution boundary.

Implementation

Implement a common error-handling library that enforces the system error handling policy in each tier of the application. The implementation in each tier should log errors in a form that technology administrators are used to in that environment (e.g. the OS log versus a text file).

The implementation of the library should include both:

- Interfaces to log technical and domain errors separately
- A generic **`SystemError`** class (or data structure) that can be used to pass summary information back to the caller.

The library routine that logs technical errors (e.g. **`technicalError()`**) should:

- log the error with all of its associated detail at the point where it is encountered;
- return a unique but human readable error instance ID (for example, based on the date such as **`"20040302.12"`** for the 12th error on 2nd March 2004); and
- capture runtime environment information in the routine that logs a technical error and add this to the error log (if appropriate).

Whenever a technical error occurs, the application infrastructure code that catches the error should call the **`technicalError`** routine to log the error on its behalf and then create a **`SystemError`** object containing a simple explanation of the failure and the unique error instance ID returned from **`technicalError`**. This error object should be then returned to the caller as shown in Listing 3.

If a technical error can be handled within a tier (including it being 'silently' ignored – see proto-pattern IGNORE IRRELEVANT ERRORS – except that it is always logged) then the **`SystemError`** need not be propagated back to the caller and execution can continue.

```
...
public class AccountRemoteFacade
   implements AccountRemote
{
  SystemError error = null;
  public SystemError withdrawFunds(float amount)
     throws InsufficientFundsException,
     RemoteException
  {
    try
    {
      // Domain code that could generate various
      // errors both technical and domain
    }
    catch (DomainException ex)
    {
      throw ex;
    }
    catch (Exception ex)
    {
      String errorId = technicalError(ex);
      error = new SystemError(ex.getMessage(),
         errorId);
    }
  }
  return error;
}
```

Listing 3

Positive consequences

- Only a required subset of the technical error information is propagated back to the remote caller – just enough for them to work out what to do next (e.g. whether to retry).
- Technical error information is logged in the environment to which it pertains (e.g. a Windows 2000 server) and in which it can be understood and resolved.
- The technical error information is logged in a similar way to (and potentially in the same place as) other system and infrastructure error information. This may make it easier to identify the underlying cause (e.g. if there are lots of related security errors alongside the database access error).
- Using local error logging mechanisms makes the logs much easier for technology administrators to access using their normal tools.
- The logging mechanism for technical errors can decorate the error information with platform-specific information that may assist in the diagnosis of the error.

Negative consequences

- One error can cause multiple log entries on different machines in a distributed environment (see UNIQUE ERROR IDENTIFIERS pattern).
- Using local error logging mechanisms means that the approach used in each tier of the system may be different.

Related patterns

- Implementing SPLIT DOMAIN AND TECHNICAL ERRORS before LOG AT DISTRIBUTION BOUNDARY makes implementation simpler, as it allows the two types of error to be clearly differentiated and handled differently.
- UNIQUE ERROR IDENTIFIERS are needed if you want to tie distributed errors into a SYSTEM OVERVIEW [Dyson04] and to to mitigate the potential confusion arising from one error causing multiple log entries.

UNIQUE ERROR IDENTIFIER

Problem

If an error on one tier in a distributed system causes knock-on errors on other tiers you get a distorted view of the number of errors in the system and their origin.

Context

Multi-tier systems, particularly those that use load balancing at different tiers to improve availability and scalability. Within such an environment you have already decided that as part of your error handling strategy you want to LOG AT DISTRIBUTION BOUNDARY.

Forces

- It is often possible to determine the sequence of knock-on errors across a distributed system just by correlating raw error information and timestamps but this takes a lot of skill in system forensics and usually a lot of time.
- The ability to route calls from a host on one tier to one of a set of load-balanced servers in another tier improves the availability and scalability characteristics but makes it very difficult to trace the path of a particular cross-tier call through the system.
- You can correlate error messages based on their timestamp but this relies on all server times being synchronized and does not help when two errors occur on servers in the same tier within a small time window (basically the time to make a distributed call between tiers).
- Similar timestamps help to associate errors on different tiers but if many errors occur in a short period it becomes far harder to definitively associate an original error with its knock-on errors.

Solution

Generate a UNIQUE ERROR IDENTIFIER when the original error occurs and propagate this back to the caller. Always include the UNIQUE ERROR IDENTIFIER with any error log information so that multiple log entries from the same cause can be associated and the underlying error can be correctly identified.

Known uses

The authors have observed this pattern in use within a number of successful enterprise systems. We do not know of any publicly accessible implementations of it (because most systems available for public inspection are single tier systems and so this pattern is not relevant to them).

Implementation

The two key tenets that underlie this pattern are the uniqueness of the error identifier and the consistency with which it is used in the logs. If either of these are implemented incorrectly then the desired consequences will not result.

The unique error identifier must be unique across all the hosts in the system. This rules out many pseudo-unique identifiers such as those guaranteed to be unique within a particular virtual platform instance (.NET Application Domain or Java Virtual Machine). The obvious solution is to use a platform-generated Universally Unique ID (UUID) or Globally Unique ID (GUID). As these utilize the unique network card number as part of the identifier then this guarantees uniqueness in space (across servers). The only issue is then uniqueness across time (if two errors occur very close in time) but the narrowness of the window (100ns) and the random seed used as part of the UUID/GUID should prevent such problems arising in most scenarios.

It is important to maintain the integrity of the identifier as it is passed between hosts. Problems may arise when passing a 128-bit value between systems and ensuring that the byte order is correctly interpreted. If you suspect that any such problems may arise then you should pass the identifier as a string to guarantee consistent representation.

The mechanism for passing the error identifier will depend on the transport between the systems. In an RPC system, you may pass it as a return value or an [out] parameter whereas in SOAP calls you could pass it back in the SOAP fault part of the response message.

In terms of ensuring that the unique identifier is included whenever an error is logged, the responsibility lies with the developers of the software used. If you do not control all of the software in your system you may need to provide appropriate error handling through a DECORATOR [Gamma95] or as part of a BROKER [Buschmann96]. If you control the error framework you may be able to propagate the error identifier internally in a CONTEXT OBJECT [Fowler].

Positive consequences

- The system administrators can use a unified view of the errors in the system keyed on the unique error identifier to determine which error is the underlying error and which other errors are knock-ons from this one. If the errors in each tier are logged on different hosts it may be necessary to retrieve and amalgamate multiple logs in a SYSTEM OVERVIEW [Dyson04] before such correlation can take place.
- Correlating errors based on the unique error id rather than the hosts on which they occur gives a far clearer picture of error cause and effect across one or more tiers of load-balanced servers.
- Skewed system times on different servers can cause problems with error tracing. If an error occurs when host 1 calls host 2, host 2 will log the error and host 1 will log the failed call. If the system time on host 1 is ahead of host 2 by a few milliseconds, it could appear that the error on host 1 occurred before that on host 2 – hence obscuring the sequence of cause and effect. However, if they both have the same unique error identifier, the two errors are inextricably linked and so the time skew could be identified and allowed for in the forensic examination.
- If lots of errors are generated on the same set of hosts at around the same time it becomes possible to determine if a consistent pattern or patterns of error cascade is occurring.

Negative consequences

- The derivation of a unique error identifier may be relatively complex in some environments and this could be a barrier to the pattern's adoption in some situations.
- The implementation of this pattern implies logging each error a number of times, once in each tier. This additional logging activity means that overall, logs will grow more quickly than in systems that do not implement this approach. This means that the runtime and administration overhead of this additional logging will need to be absorbed in the design of the system.

Related patterns

- Log at DISTRIBUTION BOUNDARY needs errors to have a unique error id in order to correlate the distributed errors.
- You may or may not employ CENTRALIZED ERROR LOGGING [Renzel97] to help assimilate errors.

To be continued...

So far, so good. However this is only part of the story as there are still some fundamental principles to be applied such as determining what is and is not an error. The remaining patterns in this pattern collection (BIG OUTER TRY BLOCK, HIDE TECHNICAL ERROR DETAIL FROM USERS, LOG UNEXPECTED ERRORS and MAKE EXCEPTIONS EXCEPTIONAL) will show how the error handling jigsaw can be completed. These patterns will be explored in the next issue. ■

References

[Buschmann96] *Pattern-Oriented Software Architecture*, John Wiley and Sons, 1996

[Dyson04] *Architecting Enterprise Solutions: Patterns for High-Capability Internet-based Systems*, Paul Dyson and Andy Longshaw, John Wiley and Sons, 2004

[Gamma95] *Design Patterns*, Addison Wesley, 1995.

[Haase] *Java Idioms – Exception Handling*, linked from http://hillside.net/patterns/EuroPLoP2002/papers.html.

[Harrison] *Patterns for Logging Diagnostic Messages*, Neil B. Harrison

[Renzel97] *Error Handling for Business Information Systems*, Eoin Woods, linked from http://hillside.net/patterns/onlinepatterncatalog.htm

No 'Concepts' in C++0x

There have been some major decisions made about the next C++ Standard. Bjarne Stroustrup explains what's changed and why.

At the July 2009 Frankfurt meeting of the ISO C++ Standards Committee (WG21) [ISO], the 'concepts' mechanism for specifying requirements for template arguments was 'decoupled' (my less-diplomatic phrase was 'yanked out'). That is, 'concepts' will not be in C++0x or its standard library. That – in my opinion – is a major setback for C++, but not a disaster; and some alternatives were even worse.

I have worked on 'concepts' for more than seven years and looked at the problems they aim to solve much longer than that. Many have worked on 'concepts' for almost as long. For example, see (listed in chronological order):

- Bjarne Stroustrup and Gabriel Dos Reis: 'Concepts – Design choices for template argument checking'. October 2003. An early discussion of design criteria for 'concepts' for C++. [Stroustrup03a]
- Bjarne Stroustrup: 'Concept checking – A more abstract complement to type checking'. October 2003. A discussion of models of 'concept' checking. [Stroustrup03b]
- Bjarne Stroustrup and Gabriel Dos Reis: 'A concept design' (Rev. 1). April 2005. An attempt to synthesize a 'concept' design based on (among other sources) N1510, N1522, and N1536. [Stroustrup05]
- Jeremy Siek *et al.*: Concepts for C++0x. N1758==05-0018. May 2005. [Siek05]
- Gabriel Dos Reis and Bjarne Stroustrup: 'Specifying C++ Concepts'. POPL06. January 2006. [Reis06]
- Douglas Gregor and Bjarne Stroustrup: Concepts. N2042==06-0012. June 2006. The basis for all further 'concepts' work for C++0x. [Gregor06a]
- Douglas Gregor *et al.*: Concepts: Linguistic Support for Generic Programming in C++. OOPSLA'06, October 2006. An academic paper on the C++0x design and its experimental compiler **`ConceptGCC`**. [Gregor06b]
- Pre-Frankfurt working paper (with 'concepts' in the language and standard library): 'Working Draft, Standard for Programming Language C++'. N2914=09-0104. June 2009. [Frankfurt09]
- B. Stroustrup: Simplifying the use of concepts. N2906=09-0096. June 2009. [Stroustrup09]

It need not be emphasized that I and others are quite disappointed. The fact that some alternatives are worse is cold comfort and I can offer no quick and easy remedies.

Please note that the C++0x improvements to the C++ features that most programmers see and directly use are unaffected. C++0x will still be a more expressive language than C++98, with support for concurrent programming, a better standard library, and many improvements that make it significantly easier to write good (i.e., efficient and maintainable) code. In particular, every example I have ever given of C++0x code (e.g., in 'Evolving a language in and for the real world: C++ 1991–2006' [Stroustrup07] at ACM HOPL-III [HOPL]) that does not use the keywords 'concept' or 'requires' is unaffected. See also my C++0x FAQ [FAQ]. Some people even rejoice that C++0x will now be a simpler language than they had expected.

'Concepts' were to have been the central new feature in C++0x for putting the use of templates on a better theoretical basis, for firming-up the specification of the standard library, and a central part of the drive to make generic programming more accessible for mainstream use. For now, people will have to use 'concepts' without direct language support as a design technique. My best scenario for the future is that we get something better than the current 'concept' design into C++ in about five years. Getting that will take some serious focused work by several people (but not 'design by committee').

What happened?

'Concepts', as developed over the last many years and accepted into the C++0x working paper in 2008, involved some technical compromises (which is natural and necessary). The experimental implementation was sufficient to test the 'conceptualized' standard library, but was not production quality. The latter worried some people, but I personally considered it sufficient as a proof of concept.

My concern was with the design of 'concepts' and in particular with the usability of 'concepts' in the hands of 'average programmers'. That concern was shared by several members. The stated aim of 'concepts' was to make generic programming more accessible to most programmers [Stroustrup03a], but that aim seemed to me to have been seriously compromised: Rather than making generic programming more accessible, 'concepts' were becoming yet another tool in the hands of experts (only). Over the last half year or so, I had been examining C++0x from a user's point of view, and I worried that even use of libraries implemented using 'concepts' would put new burdens on programmers. I felt that the design of 'concepts' and its use in the standard library did not adequately reflect our experience with 'concepts' over the last few years.

Then, a few months ago, Alisdair Meredith (an insightful committee member from the UK) and Howard Hinnant (the head of the standard library working group) asked some good questions relating to who should directly use which parts of the 'concepts' facilities and how. That led to a discussion of usability involving many people with a variety of concerns and points of view; and I eventually – after much confused discussion – published my conclusions [Stroustrup09].

To summarize and somewhat oversimplify, I stated that:

- 'Concepts' as currently defined are too hard to use and will lead to disuse of 'concepts', possibly disuse of templates, and possibly to lack of adoption of C++0x.

Bjarne Stroustrup designed and implemented the C++ programming language. He can be contacted at www.research.att.com/~bs

Unless members are convinced that the risks for doing harm to production code are very low, they must oppose

- A small set of simplifications [Stroustrup09] can render 'concepts' good-enough-to-ship on the current schedule for C++0x or with only a minor slip.

That's pretty strong stuff. Please remember that standards committee discussions are typically quite polite, and since we are aiming for consensus, we tend to avoid direct confrontation. Unfortunately, the resulting further (internal) discussion was massive (hundreds of more and less detailed messages) and confused. No agreement emerged on what problems (if any) needed to be addressed or how. This led me to order the alternatives for a presentation in Frankfurt:

- 'fix and ship'

  Remaining work: remove explicit 'concepts', add explicit refinement, add 'concept'/type matching, handle 'concept' map scope problems
  Risks: no implementation, complexity of description
  Schedule: no change or one meeting
- 'Yank and ship'

  Remaining work: yank (core and standard library)
  Risks: old template problems remain, disappointment in 'progressive' community ('seven year's work down the drain')
  Schedule: five years to 'concepts' (complete redesign needed) or never
- 'Status quo'

  Remaining work: details
  Risks: unacceptable programming model, complexity of description (alternative view: none)
  Schedule: no change

I and others preferred the first alternative ('fix and ship') and considered it feasible. However, a large majority of the committee disagreed and chose the second alternative ('yank and ship', renaming it 'decoupling'). In my opinion, both are better than the third alternative ('status quo'). My interpretation of that vote is that given the disagreement among proponents of 'concepts', the whole idea seemed controversial to some, some were already worried about the ambitious schedule for C++0x (and, unfairly IMO, blamed 'concepts'), and some were never enthusiastic about 'concepts'. Given that, 'fixing concepts' ceased to be a realistic option. Essentially, all expressed support for 'concepts', just 'later' and 'eventually'. I warned that a long delay was inevitable if we removed 'concepts' now because in the absence of schedule pressures, essentially all design decisions will be re-evaluated.

Surprisingly (maybe), there were no technical presentations and discussions about 'concepts' in Frankfurt. The discussion focused on timing and my impression is that the vote was decided primarily on timing concerns.

Please don't condemn the committee for being cautious. This was not a 'Bjarne vs. the committee fight', but a discussion trying to balance a multitude of serious concerns. I and others are disappointed that we didn't take the opportunity of 'fix and ship', but C++ is not an experimental academic language. Unless members are convinced that the risks for doing harm to production code are very low, they must oppose. Collectively, the committee is responsible for billions of lines of code. For example, lack of adoption of C++0x or long-term continued use of unconstrained templates in the presence of 'concepts' would lead to a split of the C++ community into separate sub-communities. Thus, a poor 'concept' design could be worse than no 'concepts'. Given the choice between the two, I too voted for removal. I prefer a setback to a likely disaster.

Technical issues

The unresolved issue about 'concepts' focused on the distinction between explicit and implicit 'concept' maps (see [Stroustrup09]):

1. Should a type that meets the requirements of a 'concept' automatically be accepted where the 'concept' is required (e.g. should a type **`X`** that provides +, -, *, and / with suitable parameters automatically match a 'concept' **`C`** that requires the usual arithmetic operations with suitable parameters) or should an additional explicit statement (a 'concept' map from **`X`** to **`C`**) that a match is intentional be required? (My answer: Use automatic match in almost all cases).
2. Should there be a choice between *automatic* and *explicit* 'concepts' and should a designer of a 'concept' be able to force every user to follow his choice? (My answer: All 'concepts' should be automatic).
3. Should a type X that provides a member operation **`X::begin()`** be considered a match for a 'concept' **`C<T>`** that requires a function **`begin(T)`** or should a user supply a 'concept' map from **`T`** to **`C`**? An example is **`std::vector`** and **`std::Range`**. (My answer: It should match).

The answers 'status quo before Frankfurt' all differ from my suggestions. Obviously, I have had to simplify my explanation here and omit most details and most rationale.

I cannot reenact the whole technical discussion here, but this is my conclusion: In the 'status quo' design, 'concept' maps are used for two things:

1. To map types to 'concepts' by adding/mapping attributes
2. To assert that a type matches a 'concept'.

Somehow, the latter came to be seen an essential function by some people, rather than an unfortunate rare necessity. When two 'concepts' differ semantically, what is needed is not an assertion that a type meets one and not the other 'concept' (this is, at best, a workaround – an indirect and elaborate attack on the fundamental problem), but an assertion that a type has the semantics of the one and not the other 'concept' (fulfills the axiom(s) of the one and not the other 'concept').

For example, the STL **`input`** iterator and **`forward`** iterator have a key semantic difference: you can traverse a sequence defined by **`forward`** iterators twice, but not a sequence defined by **`input`** iterators; e.g.,

When it comes to validating an idea, we hit the traditional dilemma

applying a multi-pass algorithm on an input stream is not a good idea. The solution in 'status quo' is to force every user to say what types match a `forward` iterator and what types match an `input` iterator. My suggested solution adds up to: If (and only if) you want to use semantics that are not common to two 'concepts' and the compiler cannot deduce which 'concept' is a better match for your type, you have to say which semantics your type supplies; e.g., 'my type supports multi-pass semantics'. One might say, 'When all you have is a 'concept' map, everything looks like needing a type/'concept' assertion.'

At the Frankfurt meeting, I summarized:

- Why do we want 'concepts'?

  To make requirement on types used as template arguments explicit
  Precise documentation
  Better error messages
  Overloading

Different people have different views and priorities. However, at this high level, there can be confusion – but little or no controversy. Every half-way reasonable 'concept' design offers that.

- What concerns do people have?

  Programmability
  Complexity of formal specification
  Compile time
  Run time

My personal concerns focus on 'programmability' (ease of use, generality, teachability, scalability) and the complexity of the formal specification (40 pages of standards text) is secondary. Others worry about compile time and run time. However, I think the experimental implementation (ConceptGCC [Gregor06b]) shows that run time for constrained templates (using 'concepts') can be made as good as or better than current unconstrained templates. `ConceptGCC` is indeed very slow, but I don't consider that fundamental.

When it comes to validating an idea, we hit the traditional dilemma. With only minor oversimplification, the horns of the dilemma are:

- 'Don't standardize without commercial implementation'
- 'Major implementers do not implement without a standard'

Somehow, a detailed design and an experimental implementation have to become the basis for a compromise.

My principles for 'concepts' are:

- Duck typing

  The key to the success of templates for GP (compared to OO with interfaces and more).

- Substitutability

  Never call a function with a stronger precondition than is 'guaranteed'.

- 'Accidental match' is a minor problem

  Not in the top 100 problems.

My 'minimal fixes' to 'concepts' as present in the pre-Frankfurt working paper were:

- 'Concepts' are implicit/auto

  To make duck typing the rule.

- Explicit refinement

  To handle substitutability problems.

- General scoping of 'concept' maps

  To minimize 'implementation leakage'.

- Simple type/'concept' matching

  To make vector a range without redundant 'concept' map

For details, see [Stroustrup09].

No C++0x, long live C++1x

Even after cutting 'concepts', the next C++ standard may be delayed. Sadly, there will be no C++0x (unless you count the minor corrections in C++03). We must wait for C++1x, and hope that 'x' will be a low digit. There is hope because C++1x is now feature complete (excepting the possibility of some national standards bodies effectively insisting on some feature present in the formal proposal for the standard). 'All' that is left is the massive work of resolving outstanding technical issues and comments.

A list of features and some discussion can be found on my C++0x FAQ [FAQ]. Here is a subset:

- atomic operations
- auto (type deduction from initializer)
- C99 features
- enum class (scoped and strongly typed enums)
- constant expressions (generalized and guaranteed; constexpr)
- defaulted and deleted functions (control of defaults)
- delegating constructors
- in-class member initializers
- inherited constructors
- initializer lists (uniform and general initialization)
- lambdas
- memory model
- move semantics; see rvalue references
- null pointer (nullptr)
- range for statement
- raw string literals
- template alias

C++1x will be a massive improvement on C++98

- thread-local storage (thread_local)
- unicode characters
- uniform initialization syntax and semantics
- user-defined literals
- variadic templates
- and libraries:
- improvements to algorithms
- containers
- duration and time_point
- function and bind
- forward_list a singly-liked list
- future and promise
- garbage collection ABI
- hash_tables; see unordered_map
- metaprogramming and type traits
- random number generators
- regex a regular expression library
- scoped allocators
- smart pointers; see shared_ptr, weak_ptr, and unique_ptr
- threads
- atomic operations
- tuple

Even without ‘concepts’, C++1x will be a massive improvement on C++98, especially when you consider that these features (and more) are designed to interoperate for maximum expressiveness and flexibility. I hope we will see ‘concepts’ in a revision of C++ in maybe five years. Maybe we could call that C++1y or even ‘C++y!’ ■

References

[FAQ] Bjarne Stroustrup, ‘C++0x - the next ISO C++ standard’ (FAQ), available from: http://www.research.att.com/%7Ebs/C++0xFAQ.html

[Frankfurt09] ‘Working Draft, Standard for Programming Language C++’, a pre-Frankfurt working paper, June 2009, available from: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n2914.pdf

[Gregor06a] Douglas Gregor and Bjarne Stroustrup, June 2006, ‘Concepts’, availabe from: http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2006/n2042.pdf

[Gregor06b] Douglas Gregor, Jaakko Jarvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis and Andrew Lumsdaine, October 2006, ‘Concepts: Linguistic support fro generic programming in C++’, available from: http://www.research.att.com/~bs/oopsla06.pdf

[HOPL] Proceedings of the History of Programming Languages conference 2007, available from: http://portal.acm.org/toc.cfm?id=1238844

[ISO] The C++ Standards Committee – http://www.open-std.org/jtc1/sc22/wg21/

[Reis06] Gabriel Dos Reis and Bjarne Stroustrup, January 2006, ‘Specifying C++ concepts’, available from: http://www.research.att.com/~bs/popl06.pdf

[Siek05] Jeremy Siek, Douglas Gregor, Ronald Garcia, Jeremiah Willcock, Jaakko Jarvi and Andrew Lumsdaine, May 2005, ‘Concepts for C++0x’, available from: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1758.pdf

[Stroustrup03a] Bjarne Stroustrup and Gabriel Dos Reis, October 2003, ‘Concepts – Design choices for template argument checking’, available from: http://www.research.att.com/~bs/N1522-concept-criteria.pdf

[Stroustrup03b] Bjarne Stroustrup, October 2003, ‘Concept checking – A more abstract complement to type checking’, available from: http://www.research.att.com/~bs/n1510-concept-checking.pdf

[Stroustrup05] Bjarne Stroustrup and Gabriel Dos Reis, April 2005, ‘A concept design (Rev. 1)’ available from: http://www.research.att.com/~bs/n1782-concepts-1.pdf

[Stroustrup07] Bjarne Stroustrup, May 2007, ‘Evolving a language in and for the real world: C++ 1991–2006’, available from: http://www.research.att.com/~bs/hopl-almost-final.pdf

[Stroustrup09] Bjarne Stroutstrup ‘Simplifying the use of concepts’. Available from: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n2906.pdf

This article first published in Doctor Dobb’s Journal, July 2009 (http://www.ddj.com/cpp/218600111).
Reprinted here by kind permission.